The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems

The Formal Semantics of a Domain-specific Modeling Language for Semantic Web enabled Multi-agent Systems

Sinem Getir, Moharram Challenger, and Geylani Kardas[1]

International Computer Institute, Ege University, 35100, Bornova, Izmir, Turkey
sinem.getir@ege.edu.tr, moharram.challenger@mail.ege.edu.tr, geylani.kardas@ege.edu.tr

Abstract

Development of agent systems is without question a complex task when autonomous, reactive and proactive characteristics of agents are considered. Furthermore, internal agent behavior model and interaction within the agent organizations become even more complex and hard to implement when new requirements and interactions for new agent environments such as the Semantic Web are taken into account. We believe that the use of both domain specific modeling and a Domain-specific Modeling Language (DSML) may provide the required abstraction and support a more fruitful methodology for the development of Multi-agent Systems (MASs) especially when they are working on the Semantic Web environment. Although syntax definition based on a metamodel is an essential part of a modeling language, an additional and required part would be the determination and implementation of DSML constraints that constitute the (formal) semantics which cannot be defined solely with a metamodel. Hence, in this paper, formal semantics of a MAS DSML called Semantic Web enabled Multi-agent Systems (SEA_ML) is introduced. SEA_ML is a modeling language for agent systems that specifically takes into account the interactions of semantic web agents with semantic web services. What is more, SEA_ML also supports the modeling of semantic agents from their internals to MAS perspective. Based on the defined abstract and concrete syntax definitions, we first give the formal representation of SEA_ML's semantics and then discuss its use on MAS validation. In order to define and implement semantics of SEA_ML, we employ Alloy language which is declarative and has a strong description capability originating from both relational and first-order logic in order to easily define complex structures and behaviors of these systems. Differentiating from similar contributions of other researchers on formal semantics definition for MAS development languages, SEA_ML's semantics, presented in this paper, defines both static and dynamic aspects of the interaction between software agents and semantic web services, in addition to the definition of the semantics already required for agent internals and MAS communication. Implementation with Alloy makes definition of SEA_ML's semantics to include relations and sets with a simple notation for MAS model definitions. We discuss how the automatic analysis and hence checking of SEA_ML models can be realized with the defined semantics. Design of an agent-based electronic barter system is exemplified in order to give some flavor of the use of SEA_ML's formal semantics. Lessons learned during the development of such a MAS DSML semantics are also reported in this paper.

Keywords: Multi-agent System, Semantic Web, Domain Specific Modeling Language, Semantics, Alloy

1. Introduction

Agents can be defined as encapsulated computer systems, mostly software systems, situated in an environment and capable of flexible autonomous action in this environment in order to meet their design objectives (Wooldridge and Jennings, 1995). These autonomous, reactive and proactive agents have also social ability and they constitute systems called Multi-agent Systems (MASs) in which they can interact with other agents in order to accomplish their tasks.

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

[1] Corresponding author. Tel: +90-232-3423232-3223 Fax: +90-232-3887230

The Semantic Web (Shadbolt et al., 2006) improves the current World Wide Web (WWW) such that web page contents can be organized in a more structured way tailored toward specific needs of end-users. The web can be interpreted with ontologies (Berners-Lee et al., 2001) that help machines to understand web content. Within the Semantic Web environment, software agents can be used to collect Web content from diverse sources, process the information and exchange the results. Besides, 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 (Kardas et al., 2009). Semantic web services can be simply defined as web services with semantic interface to be discovered and executed (Sycara et al., 2003). In order to support semantic interoperability and automatic composition of web services, capabilities of web services are defined in service ontologies that provide the required semantic interface. Such interfaces of semantic web services can be discovered by software agents and then the agents may interact with those services to complete their tasks. Engagement and invocation of a semantic web service are also performed according to the service's semantic protocol definitions.

However, agent interactions with semantic web services add more complexity for both design and implementation of MASs. Therefore, it is natural that methodologies are being applied to master the problems of defining such complex systems. One of the possible alternatives represents domain-specific languages (DSLs) (van Deursen et al., 2000; Mernik et al., 2005; Pereira et al., 2008; Fowler, 2011) that have notations and constructs tailored toward a particular application domain (e.g. MAS). The end-users of DSLs have knowledge from the observed problem domain (Sprinkle et al., 2009), but usually they have little programming experience. Domain-specific modeling languages (DSMLs) further raise the abstraction level, expressiveness and ease of use, since models are specified in a visual manner and they represent the main artifacts instead of software codes (Schmidt, 2006; Gray et al., 2007).

We believe that both domain specific modeling and use of a DSML may provide the required abstraction and support in creating a more fruitful methodology for the development of MASs especially when they are working on the Semantic Web environment. Within this context, prior to work discussed in here, we first sketched out the general perspective (Kardas et al., 2010) and defined a metamodel in several viewpoints (Challenger et al., 2011) for a MAS DSML which is called Semantic web Enabled Agent Modeling Language (SEA_ML). Later, we presented the concrete syntax of SEA_ML and provided supporting visual modeling tools (Getir et al., 2011). Furthermore, an interpreter mechanism for SEA_ML has also been defined over model-to-model transformations which pave the way of the code generation for the implementation of SEA_ML agents in various agent platforms (e.g. JADE (Bellifemine et al., 2001), JADEX (Pokahr et al., 2005) or JACK (Howden et al., 2001)).

Although syntax definition based on a metamodel is an essential part of a modeling language, an additional and required part would be the determination and implementation of DSML constraints that constitute the (formal) semantics which cannot be defined solely with a metamodel. Usually, these constraints are given in some dedicated constraint languages (e.g. Object Constraint Language (OCL) (OMG, 2012)). With these constraints, the semantics of a DSML includes some rules that restrict the instance models created according to the language. In other words, the formal semantics presents the meaning of associations and constraints for the language in a formal way. Moreover, formal representation of the semantics helps to identify an unambiguous definition and precise meaning of a program and to have a possibility for more accurate code generation of language-based tools (Bryant et al., 2011). A successful system verification and validation can also be achieved with a proper formal semantics definition. To define the formal semantics of a language, a definition is required by means of mathematics. Unfortunately there is a big gap between model engineering and formal mathematics. Plus, there is no standard formalism to specify the semantics of modeling languages even though the syntax of modeling languages is commonly specified by metamodels. The lack of a formal definition of DSML semantics contributes to several problems (e.g. difficulty in tool generation and analysis, formal language design and composition of modeling language) as listed in (Bryant et al., 2011).

Considering the advantages discussed above, defining the formal semantics of a DSML is one of the crucial tasks of a DSML's development. On that account, in this paper, we present the formal semantics of SEA_ML and discuss the use of the related semantics definitions on MAS model checking and validation. In this way, accurate models, conforming to the predefined specifications and constraints of SEA_ML can be achieved which in turn leads to more feasible code generation for real implementation of SEA_ML models in various MAS platforms in the future. Differentiating from similar contributions of other researchers on formal semantics definition for MAS DSL/DSMLs (e.g. (Hilaire et al., 2000), (Brandao et al., 2004), (Boudiaf et al., 2008), (Hahn and Fischer, 2009)), SEA_ML's semantics presented in this paper defines both static and dynamic aspects of the interaction between software agents and semantic web services, in addition to the definition of the semantics already required for agent internals and MAS communication.

In order to implement the defined formal semantics of SEA_ML, we employ Alloy language (Jackson, 2012) which is based on first order and relational logics. As can be noticed in further sections of the paper, implementation with Alloy makes the definition of SEA_ML's semantics to include relations and sets with a simple notation for MAS model definitions. Moreover, we also discuss how the automatic analysis and hence checking of SEA_ML models can be realized with the defined semantics. Finally, a demonstration of the model checking in question is given with a case study in the paper.

The remainder of the paper is organized as follows: In section 2, a brief discussion of Alloy language is given to warm up for the following discussion of SEA_ML's semantics. Semantics of SEA_ML along with defined language syntax is discussed in section 3. Analysis and checking of SEA_ML instance models by using the defined semantics are discussed and demonstrated in section 4. In section 5, related work is given and finally, the paper concludes in section 6.

2. The Alloy Language

In this paper, we define formal semantics of SEA_ML with Alloy specification language which also has a useful tool, Alloy analyzer, to check defined model and validate instance models according to the constraints. Alloy analyzer can find counter-examples that violate the system constraints. This is fulfilled by using a Satisfiability (SAT) solver (Jackson et al., 2000). In this way, contradictions among rules can be extracted. Alloy constructs yields efficient representations containing static and dynamic semantics for SEA_ML structures. Alloy logic comprises objects, relations and functions which are all based on first order predicate and relational logic. Atoms are primitive entities which constitute sets and relations. The relations can be composed of atoms with various arities (such as unary, binary and ternary).

Inspired from Z language (Spivey, 1988), Alloy (Jackson, 2002) has a strong description capability with presenting a declarative language based on first-order logic to define complex structures and behaviors of systems. Everything is considered as a relation in Alloy and therefore it does not propose a specialized logic for state machines, traces and concurrency to keep simplicity. Alloy is also based on the idea of finding counter-examples that detects the system faults.

SEA_ML semantics benefits from the system constraints by representing Alloy *signatures* and *constraints*. *Signature*s represent meta-elements and their relations as meta-attributes allowing inheritance and subset/superset hierarchy. *Constraint Paragraph*s include *Fact*s, *Predicate*s and *Function*s. *Fact* constraints are always held for metamodel element relations whenever a model is checked. *Predicate*s are reusable constraints to analyze the model during its evolution. On the other hand, *Function*s are reusable expressions to omit recurrent operations in the model. *Assertion*s are conjectures to check the model by considering the facts. Considering *Command*s, one of them is *Run* which runs the predicates and finds some instance models according to defined Alloy model. The other command is *Check* which generates counter-examples for *Assertions* (Jackson, 2012).

While defining the rules of the SEA_ML, we represented all meta-model elements with Signatures and added appropriate relations and attributes as Fields in Signatures. Most static semantic constraints which come from the metamodel are represented with multiplicity properties such as *one*, *some* and *set* which means "exactly one", "at least one" and "zero or more" respectively in *signatures (Sig)* (for meta-elements) and *fields* in signatures (for relations or attributes). Time signature is also added to realize the dynamic semantics.

Additionally, each relation field implicitly defines a relation from a domain set to a co-domain set by using Cartesian product (→). On the other hand, Dot join (e.g.: *p.q*) is handled by taking every combination of a tuple in relation *p* and same for relation *q* and their join if it exists. Transitive closure (*p*^) bases on the transitive operation in mathematics such that every transitive combination of tuples in a relation *p* is added to transitive closure of *p* until there is no combination. Transpose operation (~*p*) replaces the atoms in every tuples such that ~ (*A1*, *B1*) = (*B1*, *A1*). Cardinality (*#p*) gives the number of all elements in a relation (Jackson, 2012).

While choosing a specification language, we considered its semantic complexity and tool possibility among variable languages for the SEA_ML semantics definition. In addition to Alloy's widely-accepted capabilities and tool support, a more enhanced way of describing dynamic semantics contributed in our preference to use Alloy instead of its alternatives such as Z (Spivey, 1992), Object-Z (Duke et al., 1995; Smith, 2000), OCL (OMG, 2012) or Maude (Meseguer, 2000; Clavel et al., 2002). Regarding tool support, Alloy analyzer gives developers the chance to simulate runtime issues and show possible scenarios (instance models) visually. Alloy's easy specification, appropriate kernel semantics and formal specification style within its analyzer tool make it suitable for our DSML's semantics definition.

3. Semantics of SEA_ML

In a Semantic Web enabled MAS, software agents can gather Web contents from various resources, process the information, exchange the results and negotiate with other agents. Within the context of these MASs, autonomous agents can evaluate semantic information and work together with semantically defined entities like semantic web services using content languages.

SEA_ML's abstract syntax which basically describes MAS concepts and their relationships is provided by SEA_ML's platform independent metamodel (PIMM). This PIMM, which will be discussed in this paper, is an extended and updated version of the metamodel introduced in (Challenger et al., 2011). The PIMM is divided into eight viewpoints supporting the modeling of agent internals, MASs architecture and semantic web service interactions. Before going into the depths of their explanations, these viewpoints are listed and briefly described as below:

1. *Agent's Internal Viewpoint*: This viewpoint is related to the internal structures of semantic web agents (SWAs) and defines entities and their relations required for the construction of agents. It covers both reactive and Belief-Desire-Intention (BDI) (Rao and Georgeff, 1995) agent architectures.
2. *Interaction Viewpoint*: This aspect of the metamodel 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 main blocks which compose the complex system as an organization.
4. *Role Viewpoint*: This perspective delves into the complex controlling structure of the agents. All role types such as *OntologyMediatorRole* and *RegistrationRole* are modeled in this viewpoint.
5. *Environmental Viewpoint*: Agents may need to access some resources (e.g. services and knowledgebase covering the facts about the surrounding) in their environment. Use of resources and interaction between agents with their surroundings are considered in this viewpoint.
6. *Plan Viewpoint*: This viewpoint especially deals with an agent's Plan's internal structure. 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 (SWSs) and also some ontological concepts which constitute agent's knowledgebase (such as belief and fact).
8. *Agent - SWS Interaction Viewpoint*: It is probably the most important viewpoint of SEA_ML's metamodel. Interaction of agents with SWSs is described within this viewpoint. Entities and relations for service discovery, agreement and execution are defined. Also the internal structure of SWSs is modeled.

SEA_ML semantics is constituted by defining the system constraints and investigating both static semantics and dynamic semantics (which concentrates on behavioral actions and runtime issues).

During the determination of the static semantics for each viewpoint, some controls are considered such as min-max detection which restricts all multiplicity properties for MAS and SWS entities. Moreover, these controls enable the check on instance creation such as preventing null attribute assignments or setting unique names.

One of the important controls pertaining to SEA_ML's dynamic semantics is to provide the execution ordering among agent Plans. We provide ordering constraints among Plans in two state diagrams that consider both ordering of Plan types' execution during the SWS interactions and transitions of the possible behavior flow for a Plan type. Hence, we provide both internal Plan constraints and intra-Plan constraints. Finally, Time module in our semantic definitions not only contributes to building up a dynamic structure of the elements, but also gives a facility to order relations for the same element or among the elements. Specifically these two features of SEA_ML's semantics cause SEA_ML to be advantageous in MAS design comparing with other alternatives. Remaining controls covered in SEA_ML's dynamic semantics can be listed as: communication control of agents by defining some operations for message passing among agents, mutual execution and resource sharing control and finally providing the consistency between the beliefs of an agent and the facts in the environment within a time period.

Alloy has enabled us to neatly represent the static and dynamic semantics of SEA_ML. As mentioned in section 2, SEA_ML meta-elements are defined as signatures and relations; and attributes are defined as fields in the signatures. Constraints are defined as facts, predictions and functions. In addition, assertions are used to certify the constraints. In order to provide clear understanding and simplicity, defined semantics for SEA_ML is discussed in the following subsections each focusing on a specific viewpoint of the language.

Some transitions among viewpoints are needed during the definition of some semantic rules. Transitions among the viewpoints and meta-elements that play an important role for these transitions are shown in Figure 1. For instance, *SWA* meta-entity, which in fact belongs to Agent's Internal viewpoint of SEA_ML, is imported and used in the description of the semantics for MAS viewpoint. Such transitions are shown in the figure with dotted arrows. Throughout the listing and discussion of the semantics definitions, all Alloy keywords are given in bold. Also, all meta-entities belong to SEA_ML's metamodel and *facts* are given in italic inside the text. Moreover, names of the relations between the meta-entities are used as verbs in the sentences throughout the paper.

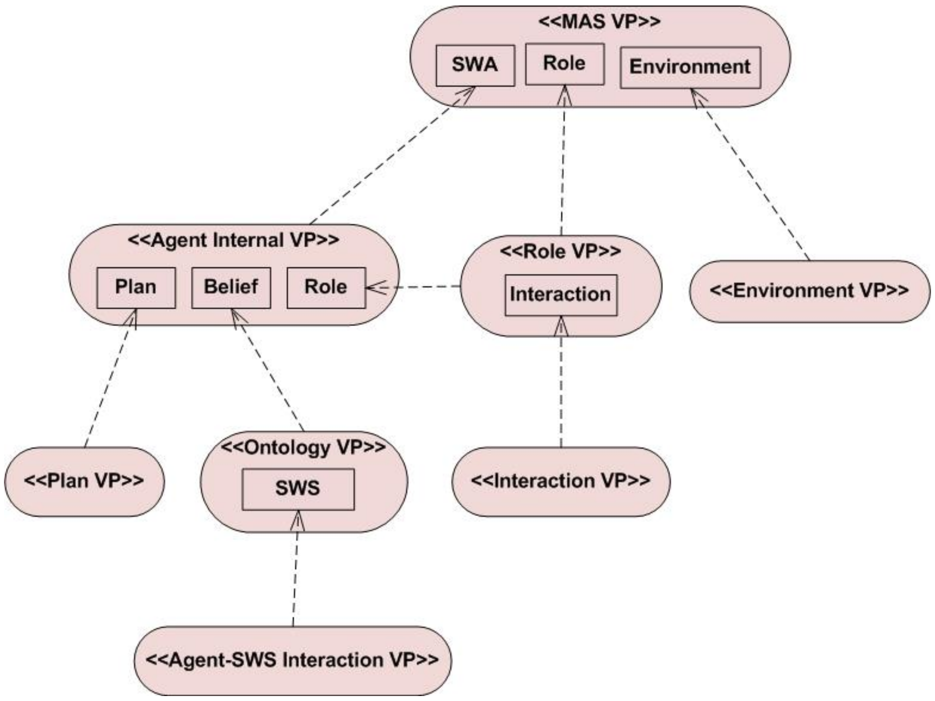


Figure 1: Overview of SEA_ML viewpoints (VPs)

3.1 MAS Viewpoint

SEA_ML's MAS viewpoint solely deals with the construction of a MAS as an overall aspect of the metamodel. It includes main blocks which compose the complex system as an organization (Figure 2). *Semantic Web Organization (SWO)* entity of SEA_ML metamodel is the main element of this viewpoint and includes SWAs which have various goals or duties. *SemanticWebAgent (SWA)* is imported from Agent's Internal viewpoint and *Role* is imported from the Role viewpoint. Alloy signature definitions which belong to MAS viewpoint are presented in Figure 3.

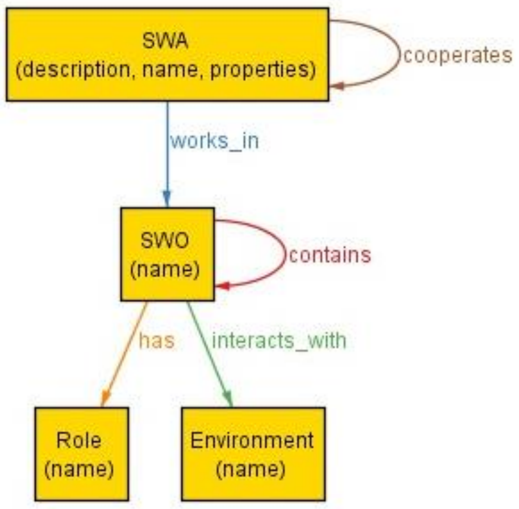


Figure 2: SEA_ML's MAS viewpoint

An agent cooperates with one or more agents inside an organization (Figure 3, Line 6) and it may also reside in more than one organization by playing various roles over time (Figure 3, line 5). *SWO*s include various roles that are to be played by the agents in the organization in accordance with their goals (Line 16). We provide the denotation of this change in an agent's role bound to the change on the MAS organization with "Time" column. More precisely, let $SWA0 \in SWA$; $SWO0, SWO1 \in SWO$ and $T0, T1 \in Time$. Then, combination of atoms can be exemplified in the time T=1 and T=2 such that we can have $(SWA0, SWO0, T0)$ and $(SWA0, SWO1, T1)$.

Moreover, a *SWO* can include several agents at any time and also each organization can be composed of several sub-organizations recursively (Line 15). Each organization interacts with an *Environment* (Line 17) which by itself includes all of the resources, services and non-Agent concepts such as a database. Hence, *SWA*s use the resources of a *SWO* in which they work.

```
01  sig SWA {
02    disj name,description,
03    property,agent_type,
04    agent_state:one Name,
05    works_in:SWO one->Time,
06    cooperates: some SWA
07  }
08  sig Environment{
09    name: one Name
10  }
```

```
13  sig SWO {
14    name:one Name,
15    contains:set SWO,
16    has:some Role,
17    interacts_with: one Environment
18  }
19  sig Role{
20   name:one Name
21  }
```

Figure 3: Signature definitions of MAS viewpoint meta-elements

As a basic rule of a MAS, there should be at least two agents in the system which is given in the *MASInit* fact (Figure 4). The cardinality of *SWA* set is greater than or equal to 2. As it is seen in the metamodel, *SWA* and *SWO* elements have self-relations. Therefore, there is a need for some constraints to handle these relationships. *irreflexive* predicate in Figure 4 controls some relation r (r ∈ univ→univ) not to be reflexive. *asymmetric* predicate controls the relation r not to be symmetric. On the other hand, *acyclic* predicate controls the relation r not to contain a cycle. Therefore, all these constraints are used in the *selfRelationControl fact* for the relation *contains* of *SWO*. That is because no SWO instance can contain itself, which means it cannot be reflexive. In other words, if $SWO1 \in SWO$ then $(SWO1, SWO1) \notin contains$, but *contains* is an asymmetric relation. For instance, let *SWO*1, *SWO*2∈*SWO* then (*SWO*1, *SWO*2) ∈ *contains* and (*SWO*2, *SWO*1) ∉*contains*.

The third operation is added to prevent the cycles from *contains* relation. It is not claimed that *contains* is acyclic just because it is not asymmetric and irreflexive. For example, if (*SWO*1, *SWO*2) ∈ *contains* and (*SWO*2, *SWO*3) ∈ *contains*, then an element like (*SWO*3, *SWO*1) does not break the irreflexive and asymmetric predicates. However, *SWO*1 *contains* *SWO*3 via *SWO*2 (due to transitiveness). Therefore, an opposite relation of (*SWO*3, *SWO*1) is a kind of a contradiction for *contains* relation as it is one directional relation. This rule can also be provided by fulfilling the statement "relations r's transitive closure is asymmetric". Precisely, for a relation r which is not reflexive and symmetric, representation not (^r & iden) and asymmetric [^r] provides that r is acyclic (that means they are equal).

On the contrary, SWA's *cooperates* relation should be irreflexive as a SWA does not cooperate with itself. Hence, irreflexive [cooperates] is added in Figure 4, line 13. For *cooperates* relation, asymmetric or a cyclic constraint cannot be added, since a cooperation can be in different directions and contain different cycles.

```
01  fact MASInıt{ #SWA>=2
02  }
03  pred irreflexive[r: univ -> univ] {
04    no (iden & r)
05  }
06  pred asymmetric[r: univ -> univ] {
07    no (r & ~r)
08  }
09  pred acyclic [r: univ->univ]{
10    no (^r & iden)
11  }
12  fact selfContainment{  irreflexive[contains] &&
13    irreflexive[cooperates] && asymmetric[contains] &&
14    acyclic[contains]
15  }
```
Figure 4: Constraint definitions of MAS viewpoint

3.2 Agent's Internal Viewpoint

This viewpoint, as a part of whole metamodel, focuses on the internal structure of every agent in a MAS organization. As it can be seen in Figure 5, *SWA* in the SEA_ML abstract syntax stands for each agent which is a member of Semantic Web enabled MAS. Hence the main element of this viewpoint is *SWA*. A SWA is an autonomous entity which is capable of interacting with both other agents and semantic web services within the environment. They can play roles and use ontologies to maintain their internal knowledge and infer about the environment based on the known facts.

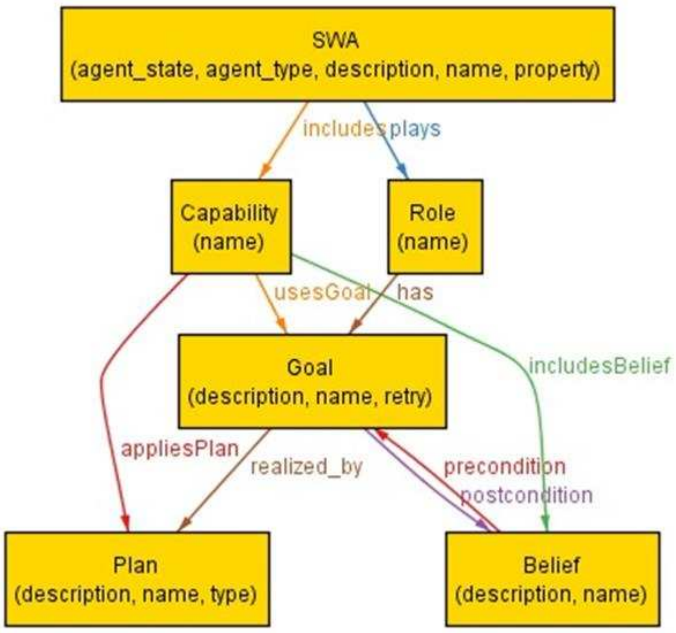


Figure 5: Agent's Internal viewpoint of SEA_ML

SWAs can be associated with more than one *Role* (multiple classifications) and can change these roles over time (dynamic classification). Taking different types of roles into consideration, an agent can play for instance a Manager role, a Broker role or a Customer Role. Signature definitions of meta-elements are presented in Figure 6. As it is mentioned in section 3.1, "Time" column enables the agent to change its role over time. As an example, let $SWA0 \in SWA,\ R0, R1 \in Role$ and $T0, T1 \in Time$, then atom examples $(SWA0, R0, T0)$ and $(SWA0, R1, T1)$ mean that agent plays different roles in the time $T = 0$ and $T = 1$ (Figure 6, Line 5).

"description" and "property" attributes represent the definition and general features of an agent respectively (Figure 6, Lines 2-3). An agent can also have a type (*Agent Type*) during its life based on the application in which it is going to take part, such as buyer agent/shopping bot, user/personal agent, monitoring-and-surveillance agent, or data mining agent (Haag et al., 2003). During the execution, agent state can change in different cases. Therefore agent state attribute is considered in the agent communication (Line 4). An agent can only have one state (*Agent State*) at a time, e.g. waiting state in which the agent is passive and waiting for another agent or resource. Similarly, it can be active while doing the internal or external processes. Therefore, it helps an agent to decide about communication with another agent by considering its state. In addition, an agent can include zero, one or more *Capabilities* (Line 6).
SEA_ML's abstract syntax supports both reactive and BDI agents. As discussed in (Vidal et al., 2001), a reactive agent does not maintain information about the state of its environment but simply reacts to current perceptions. In fact, it is only an automation that receives input, processes it and produces an output (Ferber, 1999). On the other hand, in a BDI architecture (Rao and Georgeff, 1995), an agent decides on which *Goal*s to achieve and how to achieve them. Beliefs represent the information an agent has about its surroundings, while *Desire*s correspond to the things that an agent would like to achieve. *Intentions*, which are deliberative attitudes of agents, include the agent planning mechanism in order to achieve goals.

A *Belief* in a SEA_ML model is a representation of the knowledge of an agent about the environment. "update_type" attribute of *Belief* shows that Belief is updated according to environment variants or Belief is independent from sensors (Figure 6, Line 32). For this reason, update type can be defined as dynamic or static. The update frequency can depend on the update frequency variable.

An agent in a BDI architecture has some goals to realize its final aim. “retry” attribute of *Goal* gets Boolean values in case the *Goal* is unsuccessful to process the *Goal* again. Hence, *Goal*s are reconsidered or given up (Figure 6, Line 24).

Agents execute *Plan*s to achieve their *Goal*s. *Goal* meta-entity should be realized by the *Plan* which is applied for that *Goal* (Figure 6, Line 27). On the other hand, *Goal* meta-element is in an interaction with every “Event” of the agent. According to this interaction, Goal is connected to Belief with precondition before an event (Line 33) and Belief is connected to Goal with post-condition after an event (Line 25). In this case, during an event by SWA, precondition which belongs to the Goal is retrieved by Belief and informed to Belief after the event. The Event column is defined as a signature in the definitions, but it does not belong to the metamodel. It is added as a Time column. Apart from the “Time” column, the Event column enables a dependency between these two meta-elements, *Goal* and *Belief.* For instance, let $G0 \in Goal, B0 \in Belief$ and $E0, E1 \in Event$, then the instances such as $(G0, B0, E0)$ and $(G0, B0, E1)$ mean that same *Goal* and *Belief* instances can depend on each other with different Events.

```
01  sig SWA {                         22  sig Goal {
02    disj name, description,         23    disj name, description:  one
03    property, agent_type,           24    Name, retry: one   Bool,
04    agent_state: one Name,          25    postcondition: set Belief->
05    plays: Role -> Time,            26    one Event,
06    includes: Capability            27    realized_by: some Plan
07  }                                 28  }
08  sig Capability {                  29  sig Belief {
09    disj name: one Name,            30    disj name, description:  one
10    priority: one Int,              31    Name,
11    appliesPlan: some Plan,         32    update_type:one Type,
12    includesBelief: set  Belief,    33    precondition: set
13    usesGoal: set Goal              34    Goal-> one Event
14  }                                 35  }
15  sig Plan {                        36  sig Role {
16    disj name, type,                37    name: one Name,
17    description: one Name,          38    has: Goal
18    priority: one Int,              39  }
19  }                                 40  abstract sig Type {}
20  sig Event{                        41  one sig Dynamic, Static extends
21  }                                 42  Type{}
```
Figure 6: Signature definitions of Agent’s Internal viewpoint

Considering BDI supported agent platforms (e.g. JADEX (Pokahr et al., 2005) and JACK (Howden et al., 2001)), *Capability*, which covers *Plan*s, *Goal*s and *Belief*s, is included in this viewpoint. *Capability* provides reusability by collecting the BDI elements together. *Plan*, *Belief* and *Goal* meta-elements are connected to *Capability* by the relations *appliesPlan*, *includesBelief* and *usesGoal* respectively (Figure 6, Lines 11-13).

In a BDI architecture, a capability which obtains functionality for the “library routines” (Padgham and Winikoff, 2004) should be a well-defined collection of Plans, Beliefs and Goals. *CapabilityComposition* and *CapabilityCoverage* facts in Figure 7 provide related BDI elements inside a *Capability*. This presents modularity of SEA_ML Agent’s Internal viewpoint. Line 3 in Figure 7 states that if a *Goal* is realized by a *Plan*, the *Goal* and the *Plan* should be in the same *Capability*. An example for the left hand side operation is as follows:

Let $P0 \in Plan,\ C0, C1 \in Capability$ and $(C0, P0), (C1,\ P1) \in appliesPlan$ then $\sim appliesPlan = \{(C0, P0), (P1, C1))\}$ and $(P0. \sim appliesPlan) = \{C0\}$.

Right hand side:
Let $G0, G1 \in Goal,\ C0, C1 \in Capability$ and $(C0, G0), (C1,\ G1) \in usesGoal$ then $\sim usesGoal = \{(G0, C0), (G1, C1))\}$ and $(C0. \sim appliesPlan) = \{C0\}$

Hence, $(G0, P0) \in realized\_by$ and $G0$ and $P0$ are in the same $C0$. Therefore, dot join (.) operation here yields to compare *Capabilities*.

Lines 4-5 in Figure 7 provides a similar constraint which means that for all *Goal* and *Belief* elements, if a *Capability* uses a *Goal* element and a *Goal* element is connected to a *Belief* with *postcondition* depending on an "Event", then that *Belief* is in the same *Capability* which the *Goal* is used by. Firstly, dot join operator in Line 5 is used between *Goal* and *postcondition* relation elements (this gives the tuples like *G.(G,B,E) = (B,E)*) then, that operator joins the result with "Event" (*(B,E).E = B*). The final result gives a set of *Belief* to check whether this set of *Belief* is **in** the same Capability with *Goal*.

On the other hand, modeling relationships such as composition and aggregation are not defined in Alloy (Anastasakis et al., 2007). Therefore, *CapabilityComposition* fact controls existence of BDI elements in a *Capability*. Line 9 of Figure 7 holds that for all *Plan*s, a *Capability* which applies the *Plan* cannot be an empty set (!**none**) which means every *Plan* is connected to a *Capability*. For example, $P0 \in Plan,$ $C0, C1 \in Capability$ and $(C0, P0), (C1,\ P1) \in appliesPlan,$ then, $\sim appliesPlan = \{(P0, C0), (P1, C1))\}$. $(P0. \sim appliesPlan)$ is a non-empty set and is equal to $C0$. The same rule is given in Line 9 of Figure 7 for *Belief* elements. However, such a rule is unnecessary for *Goal* elements, because metamodel forces a *Goal* to have at least one *Plan* and Lines 2-3 already forces the *Plan* and the *Goal* to be in the same *Capability*.

Unlike *Beliefs,* both *Plan*s and *Goal*s can be sharable in a MAS since agents can apply various plan codes and have common *Goal*s. Therefore, a fact called *ForbiddingSharing* is added (Figure 7, Lines 11-15) for *Belief* instances. According to this fact, there is no such a *Belief* that it is included by a Capability which is included by a different SWA.

```
01  fact CapabilityCoverage {
02    all g:Goal|some  p:Plan|
03    g.realized_by = p && p.~appliesPlan = g.~usesGoal
04    all b:Belief,g:Goal|some c:Capability,e:Event|c.usesGoal=g
05    && g.postcondition.e=b => b in c.includesBelief
06  }
07  fact CapabilityComposition{
08    all p:Plan,  b:Belief|
09    p.~appliesPlan!=none && b.~includesBelief ! = none
10  }
11  fact ForbiddingSharing{
12    no b:Belief|some disj swa1,swa2:SWA|some
13    c:Capability|c.includesBelief=b &&
14    (swa1.includes=c&&swa2.includes=c)
15  }
```
Figure 7: Semantic constraints of Agent's Internal viewpoint

3.3 Role Viewpoint

SWAs and SWOs (as a whole) can play roles and use ontologies to maintain their internal knowledge and infer about the environment based on the known facts. As discussed in subsection 3.2, agents can also use several roles and can alter these roles over the time. *Role* is a general model entity and should be specialized in the metamodel according to architectural and domain tasks (Figure 8).

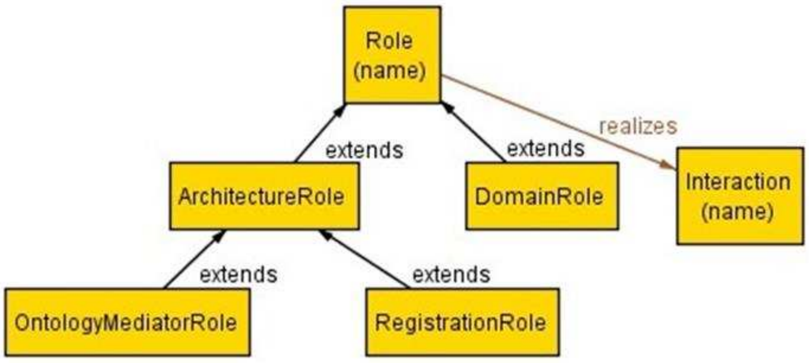


Figure 8: SEA_ML's Role viewpoint

An *ArchitectureRole* defines mandatory roles for a Semantic Web enabled MAS which should be played with at least one agent inside the platform regardless of the organization. On the other hand, a *DomainRole* depends completely on the requirements and task definitions of a specific SWO created for a specific business domain. Since a Role can have various duties, it can have different interactions with different agents. So Roles realize the *Interaction* in which they participate. Two specialization of the *ArchitecturalRole* are also defined in the model: *RegistrationRole* and *OntologyMediatorRole*. *RegistrationRole*s are played by one or more SWAs which store capability advertisements of SWSs. *OntologyMediatorRole* in the metamodel defines basic ontology management functionality that should be supported by some agents in the SWO. Signature definitions for Role viewpoint are given in Figure 9.

```
sig Role {
  name: one Name,
  realizes: some   Interaction,
}
sig Interaction{
  name: one Name,
}
sig RegistrationRole extends
ArchitectureRole{
}
sig ArchitectureRole extends
Role{
}
sig DomainRole extends Role{
}
sig OntologyMediatorRole
extends ArchitectureRole{
}
```

Figure 9: Signature definitions of Role viewpoint

In this viewpoint, it is provided that a SWO *has* Role instances and each role is played by an agent. This control is given with *RoleModularity* fact listed in Figure 10. SWO - Role and SWA - Role relations are added from other viewpoints (see Figure 3, Line 16 and Figure 6, Line 5) to *Role* entity in Alloy model to support this constraint.

```
fact RoleModularity{
  all r:Role|  r.~has!=  none || r.~plays!= none
}
```

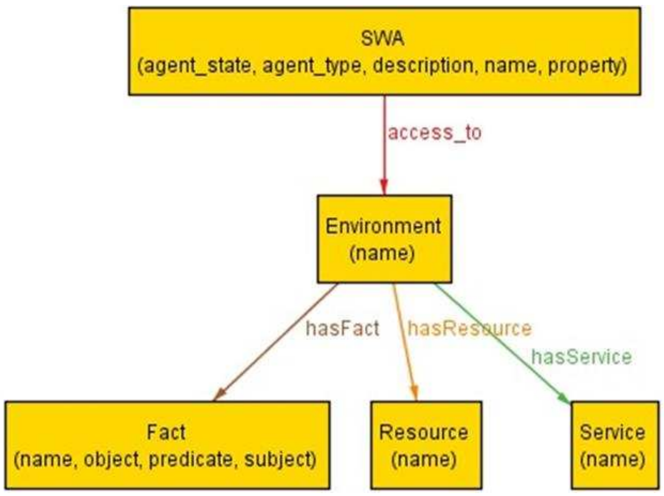
Figure 10: Role Modularity

According to this rule, the dot join of *Role* and the transpose of the relation *has* (*SWO×Role=has*) will be a set of SWO and should be a non-empty set. Or the dot join of *Role* and the transpose of *plays* relation (*SWA×Role = plays)* will be a SWA set and this should be a non-empty set.

3.4 Environment Viewpoint

SEA_ML's Environment viewpoint (Figure 11) focuses on the relations between agents and what they access. *Environment,* in which agents reside, contains all non-agent *Resource*s (e.g. database, network device), *Fact*s and *Service*s. Each service may be a web service or another service with predefined invocation protocol in real-life implementation. Facts are environment-based which means they can change over time, in case the Environment has new knowledge from different resources.

Environment meta-entity, which is the main element of this viewpoint, has a relation to *Fact*, *Service* and *Resource* with *hasFact*, *hasService* and *hasResource* respectively as can be seen in the signature definitions in Figure 12 (Lines 9-11). SWA, which is imported from Agent's Internal viewpoint, has access to Environment in order to use its components (Line 5). Fact meta-entity is extended from ODM OWL Class (which is imported from Object Management Group's (OMG) Ontology Definition Metamodel (ODM) (OMG, 2009)) and has a triple structure. Therefore, it has "subject", "predicate" and "object" attributes forming a Resource Description Framework (RDF) triple structure (Lines 18-20). *Fact* inherits these attributes from ODM OWL Statement, however *ODMOWLStatement* is not included in this viewpoint. The relation of Fact and *ODMOWLStatement* is included in the Ontology viewpoint (see subsection 3.7).


SWA
(agent_state, agent_type, description, name, property)
access_to
Environment
(name)
hasFact
hasResource
hasService
Fact
(name, object, predicate, subject)
Resource
(name)
Service
(name)
Figure 11: SEA_ML's Environment viewpoint

```
01 sig SWA {
02    disj name,description,
03        property,agent_type,
04        agent_state   : one Name,
05    access_to: some  Environment
06 }
07 sig Environment {
08    name: one Name,
09    hasFact: set Fact,
10    hasService: set Service,
11    hasResource: set Resource
12 }
```

```
15 sig Service{ name: one Name }
16 sig Fact {
17    name: one Name,
18    subject: one Name,
19    predicate: one Name,
20    object: one Name
21 }
22 sig Resource{
23    name: one Name
24    IsSharable: Boolean
25 }
```

Figure 12: Signature definitions of Environment viewpoint

To enable *Resource*, *Service* and *Fact* to exist within *Environment*, *EnvironmentComposititon fact* is built (Figure 13). That provides the composition of *Resource*, *Service* and *Fact* in *Environment* as similar to Capability modularity constraint. According to this constraint every Environment set is a non-empty set which is related to *Service*, *Fact* and *Resource* (Figure 13, Lines 2-5).

One of the required constraints is a control for sharing mechanism when agents use Resources. On the other hand, access from an agent to resource is a kind of dynamic behavior. There is no direct relation between an agent and resource in the metamodel. This relation is provided indirectly with the relations "*SWA (SemanticWebAgent)* accesses to *Environment"* and "*Environment* has some *Resource*s". This can be seen in *ResourceAccess* fact in Figure 13. Therefore, in Line 8, the Time column which provides the dynamic behavior is added to the dot join of *access_to* and *hasResource*.

More precisely, let $SWA0, SWA1 \in SWA$; $E0, E1 \in Environment$ and $(SWA0, E0)$, $(SWA1, E0)$, $(SWA1, E1) \in access\_to$. It means that agent $SWA0$ accesses to Environment $E0$, while agent $SWA1$ accesses to both Environments $E0$ and $E1$. Let $R0, R1, R2 \in Resource$ and $(E0, R0), (E0, R1), (E1, R2) \in hasResource$. Since there is no direct relation from *SWA* to *Resource,* dot join of *access_to* and *hasResource* relations gives the relation set *SWA* and *Resource*. Then, *access_to.hasResource* $= \{(SWA0, E0), (SWA1, E0), (SWA1, E1)\}.\{(E0, R0), (E0, R1), (E1, R2)\} = \{(SWA0, R0), (SWA0, R1), (SWA1, R0), (SWA1, R1), (SWA1, R2)\}$.

In this case, $R2$ is not in the intersection set but $R0, R1$ are. Resources *SWA0* and *SWA1* are able to access *R0* and *R1*. This access should happen at different times. When we get the Cartesian ("arrow") product of this set and "Time" column, $\exists \{T0, T1, T2, T3\} \in Time$;
access_to.hasResource →Time = $\{(SWA0, R0, T0), (SWA0, R0, T1), (SWA0, R1, T0), (SWA0, R1, T1), (SWA1, R0, T0), (SWA1, R0, T1), (SWA1, R1, T0), (SWA1, R1, T1), (SWA1, R2, T0), (SWA1, R2, T1)\}$.

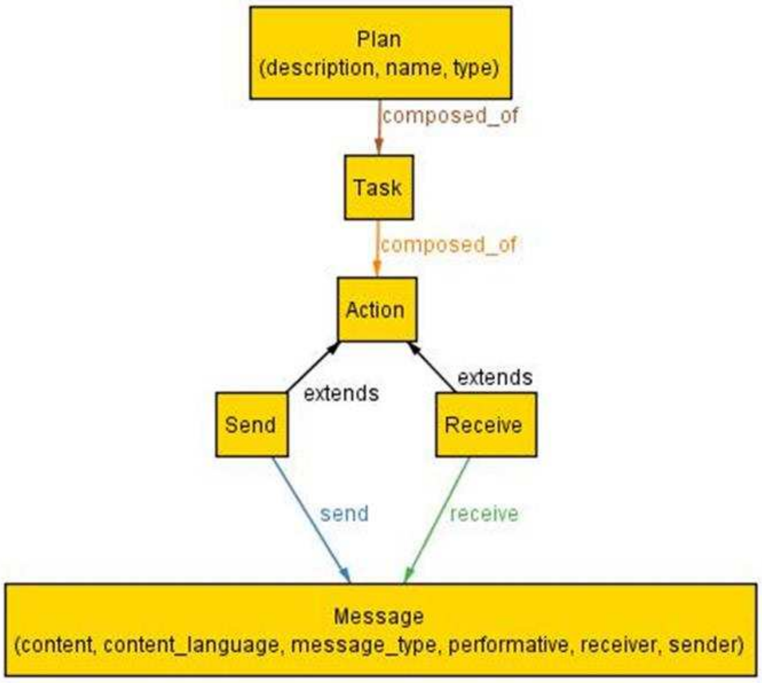
In Line 8, the created set is assigned to *access* set by using **let** keyword. For all *SWA*s, if dot join of *SWA*s and *access* are equal to each other (this operation results like (R,T) $\in Resource \times Time$), then *SWA* instances are equal to each other. For example, one of the elements of $(SWA0, R0, T0)$ and $(SWA1, R0, T0)$, one of the elements of $(SWA0, R0, T1)$ and $(SWA1, R0, T1)$, one of the elements of $(SWA0, R1, T0)$ and $(SWA1, R1, T0)$ or one of the elements $(SWA0, R1, T1)$ and $(SWA1, R1, T1)$ should be removed from *access* set to order this constraint true. As a result, this constraint provides that different agents cannot access the same non-sharable resource at the same time. Note that such complex constraint is provided easily with this language.

One of the semantic rules, which provides transition between viewpoints, is given with *EnvAccess* fact in Figure 13 (Lines 12–16). *SWO* element from MAS viewpoint, *SWA* element from agent internal viewpoint and their relations are added to this constraint. In this manner, for all *SWA*s and such a *SWO* in which these *SWA*s work, *SWA*s can access the *Environment* to which this specific *SWO* interacts at any time.

```
fact EnvironmentComposition {
    all s:Service, f:Fact, r:Resource|
    s.~hasService != none &&
    f.~hasFact != none &&
    r.~hasResource != none
}
fact ResourceAcccess{
  let  access = access_to.hasResource ->Time { if
  all a1,a2:SWA|   a1.access=a2.access => a1=a2
  }
}
fact EnvAccess{
  all swa: SWA | some t:Time, swo: SWO |
  swa.works_in.t =swo  &&
  swa.access_to in swo.interacts_with
}
```
Figure 13: Semantic rules for Environment viewpoint

3.5 Plan Viewpoint

Plan viewpoint defines the internal structure of an agent's plans. Plan entity is the main element of this viewpoint and has some attributes such as name, type, description and priority as illustrated in Figure 14. Plan viewpoint elements are defined with signatures given in Figure 15. When an agent applies a *Plan*, it executes its *Task*s which are composed of the atomic elements called *Action*s. *Send* and *Receive* elements extend *Action* (Figure 15, Lines 14 and 17). These action types are connected with a *Message* entity. Sending a message to another agent or querying an ontology are some examples of Action.


Plan
(description, name, type)
composed_of
Task
composed_of
Action
extends
extends
Send
Receive
send
receive
Message
(content, content_language, message_type, performative, receiver, sender)
Figure 14: SEA_ML's Plan viewpoint

Some constraints are required during the Plan executions according to their priorities. Priority attribute can define the execution order. For this purpose, some functions such as *next* and *prev* from Alloy ordering module are imported and used (Figure 16, Line 2). Ordering module can be used to order sets mostly states, numbers and so on (Jackson, 2012). As Plans and internal components represent states and state transitions in our system, we use ordering module for these components as states. Therefore, we define ordering module such as Util/Ordering[*Plan*], Util/Ordering[*Action*] and Util/Ordering[*Task*]. Function *Next[]* returns the next element of an element and *Prev[]* returns the previous element of the element in the ordering. *Prevs* and *Nexts* return the set which is the previous set and the next set of the element respectively.

```
sig Plan {
      disj name, type,
      description: one Name,
      priority: one Int,
      composed_of: set Task
}
sig Task{
      id: one Int,
      composed_of: set Action,
}
sig Action{
      id: one Int
}
sig Send extends Action{
      send: one Message
}
sig Receive extends Action{
      receive: one Message
}
sig Message{
      content,
      content_language,
      message_type,
      performative: one Name,
      sender: one SWA,
      receiver: some SWA
}
```

Figure 15: Signature definitions of Plan viewpoint

PlanPriority fact in Figure 16 provides that Plans with a smaller priority number execute earlier. The same control is supplied for *Task* and *Action* inside the Plan internal. In Line 5, Task, which has a smaller id, is executed first. It is similar with the control for *Action* elements in Line 6.

The other constraint is about the composition relations. Every *Plan* executes as a composition of *Task*s and every *Task* executes as a composition of *Action*s. Therefore in Lines 9-11, Plan set which belongs to Task and Task set which belongs to Action are non-empty sets.

On the other hand, processing of a message shows that it is either a “send message” or “receive message”. Hence, for all *Message*s, a *Message* is connected to either *Send* or *Receive* entity (Lines 14 and 15).

```
01  fact PlanPriority{
02    all p1, p2:Plan| p1.priority<p2.priority => plan/prev[p2]=p1
03  }
04  fact ActionTaskOrdering{
05    all disj T1,T2: Task| T1.id<=T2.id => task/next [T1] = T2
06    all disj A1,A2: Action| A1.id<=A2.id =>action/next[A1] = A2
07  }
08  fact PlanInternal{
09    all t:Task, a:Action|
10      t.~composed_of != none &&
11      a.~composed_of != none
12  }
13  fact MessageFact{
14    all m:Message| some s:Send, r:Receive | m.~send=s ||
15    m.~receive=r
16  }
17  fact MessageAccess{
18    some rl: Role, g:Goal, t:Task, s:Send, r:Receive,
19    i:Interaction, p:Plan| all m:Message |
20    rl.has = g && g.realized_by = p && p.composed_of = t &&
21    {t.composed_of = r || t.composed_of = s} &&
22    {s.send = m || r.receive=m} => rl.realizes=i && i.includes=m
23  }
```

Figure 16: Semantic rules for Plan viewpoint

MessageAccess constraint which provides the transition between the Plan viewpoint and the other viewpoints is given in Figure 16 (Lines 17-23). The whole constraint, in summary, enables the control of identification and uniqueness of each Message element by accessing the same Message instance over different relationship paths. Interpretation of the constraint is illustrated in Figure 17. *Interaction* set from Interaction viewpoint; *Goal* set from Agent's Internal viewpoint, *Role* set from Role viewpoint are added to the model as *signature*s. This constraint suggests that the Message received by 'Receive' or sent by 'Send' actions (already in the agent's Task contained by the Plan that figured out the Goal is owned by the Role (path 2 in Figure 17)) should be the same with the Message which is contained by the Interaction realized by the same Role (path 1 in Figure 17).

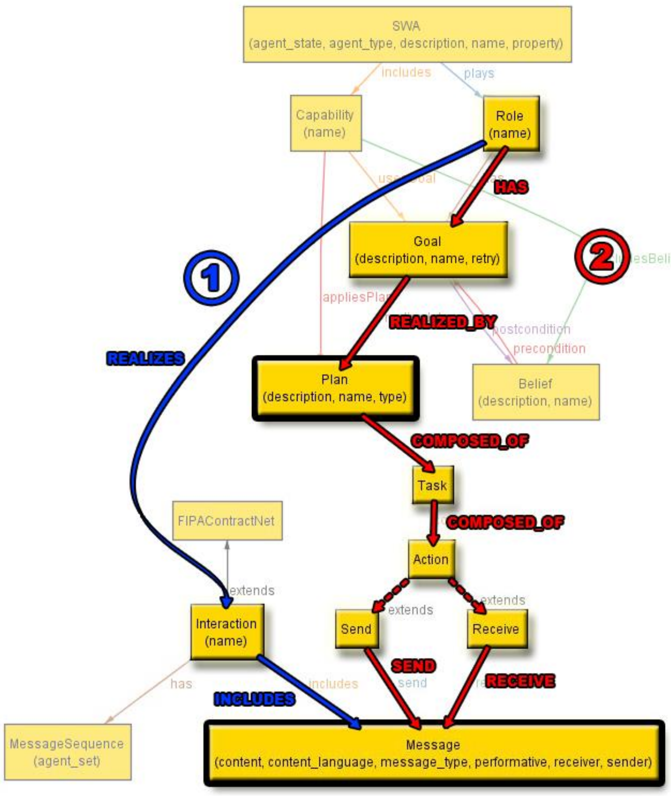
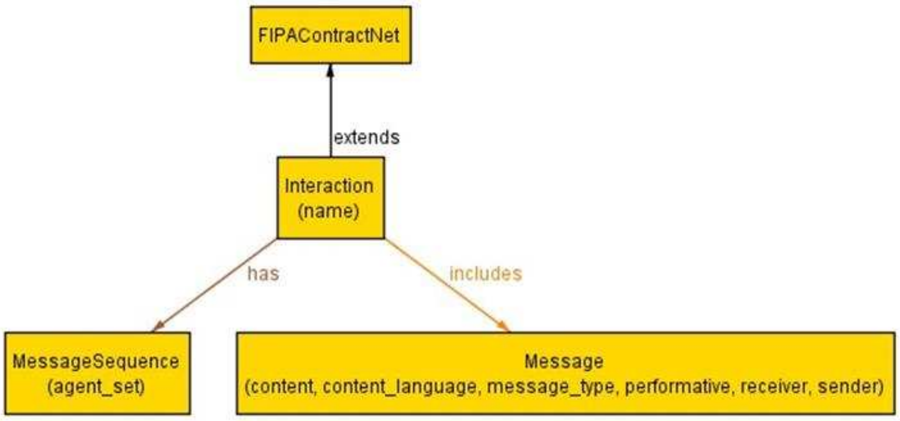


Figure 17: Transition among the viewpoints for the *MessageAccess* rule

3.6 Interaction Viewpoint

This viewpoint focuses on agent communications and interactions in a MAS and defines entities and relations such as *Interaction*, *Message*, and *MessageSequence* (Figure 18). *Interaction* is the main element of this viewpoint (Figure 19, Line 12). Agents interact with each other based on their social abilities. Each interaction, by itself, consists of some Message submissions (Figure 19, Line 17) each of which should have a message type, (Figure 19, Line 3) such as "inform", "request", or "acknowledgement". Specifically, each communication between initiator and participant agents can be modeled with Messages which can also have performative property (e.g. inform, query, or propose) compatible with IEEE FIPA standards (FIPA, 2002a). The content language property of Message entity is used for the communication between agents and can be one of the communication languages such as Knowledge Query and Manipulation Language (KQML) (Finin et al., 1994) or FIPA Agent Communication Language (ACL) (FIPA, 2002b). Interaction element extends *FIPAContractNet* element. *FIPAContractNet* represents IEEE FIPA's specification for the interactions of agents, which applies the well-known Contract Net Protocol (CNP)

(Smith, 1980). In addition, each Interaction should have a *MessageSequence* to control the communication flow (Figure 19, Line 16). Communication of distributed agents can be handled by a sequence diagram or an activity diagram with using this entity.



FIPAContractNet
extends
Interaction (name)
has
includes
MessageSequence (agent_set)
Message (content, content_language, message_type, performative, receiver, sender)

Figure 18: SEA_ML's Environment viewpoint

Agent interaction rules are important for this viewpoint. In Figure 19, Lines 5, 6 and 10, a co-domain set of the relations is defined as *SWA*. Notice that, a SWA actually is not included in this viewpoint; however, a part of *SWA* signature set is defined here again to model this viewpoint and to define the rules (Figure 19, Lines 22-24).

```
01  sig Message{
02     content, content_language,
03        message_type,
04        performative: one   Name,
05     sender: one SWA,
06     receiver: some SWA
07  }
08  sig MessageSequence {
09     id: one Int,
10     agent_set: some SWA
11  }
12  sig Interaction extends
13  FIPAContractNet{
14     name: one Name,
15     has: one
16        MessageSequence,
17     includes: some Message,
18  }
19  sig FIPAContractNet{
20     spec_no: one Int
21  }
22  sig SWA {
23     cooperates: set SWA
24  }
```
Figure 19: Signature definitions of Environment viewpoint

In Figure 20, *AgentTalking* fact provides cooperation for sender and receiver agents. In Line 2, for all *Message*s and for any two *SWAs*, let *swa1* in *SWA*'s *receiver* set and let *swa2* in SWA's *sender* set, either *swa2* should be in the set which *swa1* cooperates with or *swa1* should be in the set which *swa2* cooperates with. Shortly, if two SWAs send messages to each other, they should be in cooperation.

On the other hand, *AgentSet* fact in Figure 20 provides that all sender and receiver SWA sets are in the set which message sequence includes. In Line 7, for all *Interaction*s and *MessageSequence*s and for such a *Message;* a *Message* is in the set "*Interaction includes"* and *MessageSequence* is in the set "*Interaction has"* (Line 8). Therefore the receiver and the sender of the same *Message* should be in a *SWA* set of the same *MessageSequence* (Line 9).

SelfMessage fact provides that sender and receiver of a Message should not be the same agent. Since *Message* concept is considered as a structure for messaging between the agents, messaging between the internal components of the agents are prevented.

```
fact AgentTalking{
  all m: Message| some swa1,swa2: SWA| swa1 in m.receiver
  &&swa2 in m.sender=>swa2 in swa1.cooperates || swa1 in
  swa2.cooperates
}
fact AgentSet{
  all i:Interaction | some m:Message, ms: MessageSequence|
  m in i.includes && ms in i.has &&
  m.receiver in ms.agent_set && m.sender in ms.agent_set
}
fact SelfMessaging{
 all m:Message| m.sender != m.receiver
}
```
Figure 20: Semantic rules for Interaction viewpoint

At the same time, some constraints are supplied to be used during the model analysis such as functions or predicates for reusability especially on message sending and receiving. These constraints can be defined as *pred* or *fun* in Alloy (Taghdiri and Jackson, 2003). *Pred* definition is preferred here to be able to run the cases separately. *MsgReceivePrecondition* in Figure 21 supplies the preconditions for message receiving. Message Receiving is provided with *ReceiveMsg* predicate and sending message is provided with *SendMsg* predicate.

A relation called *getMessage* to associate a Message with "Time" (their Cartesian product with *SWA*) is defined in Line 2 of Figure 21 for *MsgReceivePrecondition* predicate. In Line 3, it is provided that current *Message* is not in the set of received *Message*s before and in Line 4 the *Message* is in the set of sent *Message*s to be able to be received. Precondition operation is used in Line 8 for *ReceiveMsg* predicate and *t'* is the previous time before *t*. Following messages are defined (Line 10) similar to the one in Line 2.

```
01  pred MsgReceivePrecondition (swa: SWA, msg:Message, t:Time){
02     let getMessage = SWA->Time->Message {
03        msg !in swa.getMessage[prev[t]]
04        msg.sentTime in prevs[t]
05    }
06  }
07  pred ReceiveMsg (swa: SWA, t:Time, msg: Message){
08     MsgReceivePrecondition [swa, msg, t]
09     let t' = prev[t] {
10        let getMessage = SWA->Time->Message{
11           swa.getMessage[t] =  swa.getMessage[t'] + msg
12        }
13     }
14     msg.receiver = SWA
15  }
16  pred SendMsg (swa: SWA, t:Time, msg: Message){
17     let t' = prev[t] {
18        let sendMessage = SWA->Time->Message{
19           swa.sendMessage[t] = swa.sendMessage[t'] + msg
20        }
21     }
22     msg.sender = SWA && msg.sentTime =t
23  }
```
Figure 21: Messaging Constraints

Finally, current message set is defined as the union of current message and previous messages (Line 11). Current message is associated with aforementioned SWA's receiver (Line 14). On the other hand, there is no precondition for message sending. *SendMsg* predicate is defined in a similar way to *receiveMsg*. Additionally, current *Message*'s sender is associated with the *SWA* and current time is associated with the *SWA*'s sent time (*sentTime*).

3.7 Ontology Viewpoint

A MAS Organization in Semantic Web is inconceivable without ontologies. An ontology represents any information gathering and reasoning resource for MAS members. SEA_ML's Ontology viewpoint brings all ontology sets and ontological concepts together as shown in Figure 22. Signature definitions for the elements of this viewpoint are shown in Figure 23. ODM OWL Ontology from OMG's ODM (OMG, 2009) is the adopted standard for all of our ontology sets such as *Role*, *Organization* and *Service* Ontologies. Therefore, they extend the ODM OWL Ontology class (in Figure 23, Lines 9, 12 and 15 respectively) which has the attribute description and contains one or more *ODMOWLStatement*s and *ODMOWLClass*es.

According to this viewpoint, all of the ontologies are known by their related elements. Collection of the ontologies creates knowledgebase of the MAS that provides domain context. These ontologies are represented in SEA_ML models as *OrganizationOntology* instances. Inside a domain role, an agent uses a *RoleOntology* which is defined for the related agent role concepts and their relations. Semantic interfaces and capabilities of SWSs are described according to *ServiceOntologies*.

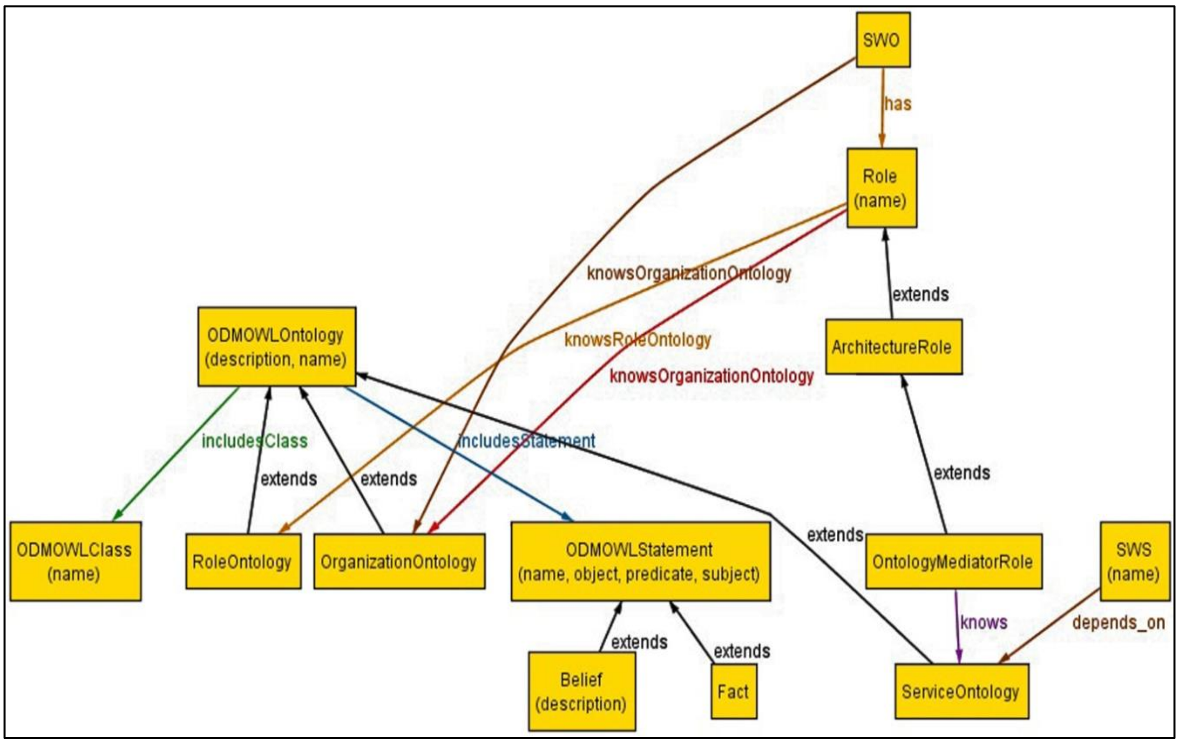


Figure 22: SEA_ML's Ontology viewpoint

Finally, for the Semantic Web environment, each fact or an agent's belief is an ontological entity and they are modeled as an extension of *ODMOWLStatement*. *ODMOWLStatement* has a structure as a triple of RDF in semantic web: "subject", "predicate", "object" (Lines 20-22). Although Belief and Fact elements have the same attributes; they have different interpretations. For instance, a Fact in the Environment keeps the current market value as 1.803 TL (Turkish Liras) for one US dollar. An agent extracts this information and keeps it in its knowledgebase. However, when the value changes to 1.700 TL, agents may not update the information. Therefore, Fact and Belief may keep different values for the same variable. This can result in an agent having inconsistencies in its knowledgebase regarding the real world. Some constraints can provide an updated *Beliefbase* with some frequencies such as the abovementioned example.

```
sig ODMOWLOntology{
   name: one Name,
   description:one Name,
   includesStatement: some
      ODMOWLStatement,
   includesClass: some
      ODMOWLClass
}
sig RoleOntology extends
   ODMOWLOntology{
}
sig OrganizationOntology extends
   ODMOWLOntology{
}
sig ServiceOntology extends
   ODMOWLOntology{
}
sig ODMOWLStatement{
   name: one Name,
   subject: one Name,
   predicate: one Name,
   object: one Name
}
sig ODMOWLClass{
   name: one Name }
sig Fact extends
   ODMOWLStatement{
}
sig Belief extends
ODMOWLStatement{
   description: one Name,
   update_type: one Type
}
sig Role {
  name: one Name,
  knowsOrganizationOntology:
    some OrganizationOntology,
  knowsRoleOntology:
    some RoleOntology,
}
sig SWO {
   has: some Role,
   knowsOrganizationOntology:
     some OrganizationOntology
}
sig SWS{
   name: one Name,
   depends_on:
      some ServiceOntology
}
```

Figure 23: Concepts of Ontology viewpoint

KnowledgeConsistency predicate is written to eliminate the inconsistencies between Belief and Fact by comparing their corresponding attributes (see Figure 24). For this, it is appropriate to compare a *SWA*'s *Belief* and *Fact* which is accessed by the same *SWA*. Therefore SWA, Capability and Environment sets are added with required relations. Exemplarily, this *pred* can run for the triples (weather, is, 15 degrees Celcius) and (weather, is, 30 degrees Celcius) without conflict.

Another constraint is needed to control the relationships between the meta-elements and the ontologies they use. *OntologyDependency* fact in Figure 24 associates a *SWO* and a *Role* which use *OrganizationOntology*. According to this constraint, if a *SWO* knows an *OrganizationOntology* and a Role knows that *OrganizationOntology*, then the *SWO* has that *Role*.

```
pred KnowledgeConsistency (b:Belief, f:Fact){
  all swa:SWA| some e:Environment, c:Capability|
  e in swa.access_to && f in e.hasFact &&
  c in swa.includes && b in c.includesBelief &&
  f.subject = b.subject && f.predicate = b.predicate =>
  f.object = b.object
}
fact OntologyDependency{
  all swo:SWO, r:Role | some OrgOnt:
  OrganizationOntology| swo.knowsOrganizationOntology =
  OrgOnt && r.knowsOrganizationOntology = OrgOnt =>
  swo.has = r
}
```

Figure 24: Ontological constraints

3.8 Agent – Semantic Web Service Interaction Viewpoint

Agent-SWS Interaction viewpoint (Figure 25), models the interaction between agents and SWSs. Concepts and their relations for appropriate service discovery, agreement with the selected service and execution of the service are all defined in this viewpoint. Furthermore, the internal structure of SWS is modeled inside this viewpoint. The preliminary version of the semantics pertaining to this viewpoint is first discussed in (Getir et al., 2012).

Semantic Web Agents apply *Plan*s to perform their tasks. In order to discover, negotiate and execute Semantic Web Services dynamically, the extensions of the *Plan* entity are defined in the metamodel. Semantic Service (*SS)_Finder Plan* is a Plan in which the discovery of candidate semantic web services takes place. *SS_AgreementPlan* involves the negotiation on QoS metrics of the service (e.g. service execution cost, running time or location) and agreement settlement. After service discovery and negotiation, the agent applies the *SS_ExecutorPlan* to execute appropriate semantic web services. As we discussed before, Semantic Service Matchmaker Agents (*SS_MatchmakerAgent*) which are extensions of *SWAs* represent service registry for agents to discover services according to their capabilities. In addition, a *SS_RegisterPlan* can be applied with a *SS_MatchmakerAgent* to register a new *SWS*.

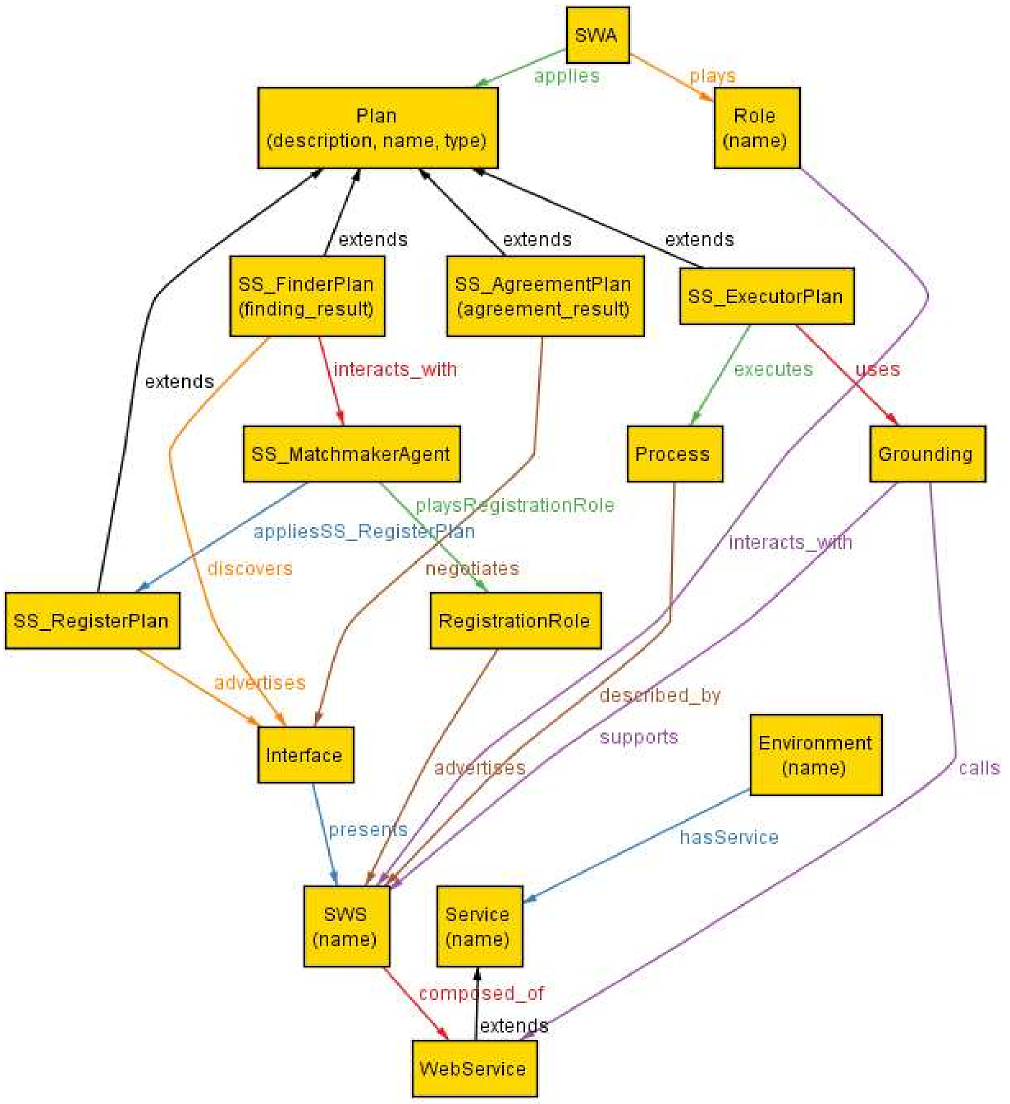


Figure 25: SEA_ML's Agent–SWS Interaction Viewpoint

SWS modeling approaches (e.g. OWL-S (Martin et al., 2004)) mostly cover three important pieces of information about semantically enriched web services which are also modeled in SEA_ML: Service Interface, Process Model and Physical Grounding. Service Interface is the capability representation of the service in which service's inputs, outputs and any other necessary descriptions are listed. Process Model defines service's internal combinations and service execution dynamics. Finally, Physical Grounding defines the service's real execution protocol. Since the operational part of today's semantic services is mostly a web service, Web Service concept is also included in SEA_ML's metamodel associated with the physical grounding mechanism. These meta-entities are shown in Figure 25 with *Interface*, *Process* and *Grounding* entities respectively. These components can use *Input*, *Output*, *Precondition* and *Effect* (a.k.a. IOPE), which model the fundamental properties of a service and extend *OWLClass* from OMG's ODM (OMG, 2009). The meta-elements of this viewpoint are also defined in Alloy signatures which are shown in Figure 26.

```
sig SWA {
    plays: Role some -> Time,
    applies: some Plan,
}
sig SS_MatchmakerAgent extends SWA{
    appliesSS_RegisterPlan:
    SS_RegisterPlan some->one Time,
 playsRegistrationRole:RegistrationRole some->one Time
}
sig Role {
    name: one Name,
    interacts_with: some SWS,
}
sig RegistrationRole extends Role {
    advertises: some SWS
}
sig SWS{
    name: one Name,
    composed_of: set WebService
}
sig Interface{
    presents: some SWS
    hasInputt:Input,
    hasOutput:Output,
    hasEffect:Effect,
    hasPrecondition:Precondition
}
sig Process{
    described_by: some SWS
    hasInputt:Input,
    hasOutput:Output,
    hasEffect:Effect,
    hasPrecondition:Precondition
}
sig Grounding{
    supports: some SWS
    calls: one WebService
}
sig Input extends ODMOWLClass{}
sig Output extends ODMOWLClass {}
sig Effect extends ODMOWLClass {}
sig Precondition extends ODMOWLClass {}
sig Service{
    name: one Name
}
sig WebService extends Service{
}
sig Plan {
    disj name,type,description:one Name,priority:one Int,
}
sig SS_RegisterPlan extends Plan{
    advertises: Interface some ->Time
}
sig SS_FinderPlan extends Plan {
    interacts_with: some SS_MatchmakerAgent
    discovers: set Interface,
}
sig SS_AgreementPlan extends Plan{
    negotiates: some Interface
}
sig SS_ExecutorPlan extends Plan{
    executes: some Process,
    uses: some Grounding
}
```

Figure 26: Concepts of Agent–SWS Interaction

One type of static semantic rules we define for this viewpoint deals with the composition relationships between *Service-Environment* and *SWS–WebService* elements. For instance, *ServiceComposition fact* is provided (in Figure 27, between Lines 1 and 4). According to that fact, every *WebService* should be connected to *SWS* via *composed_of* relation.

On the other hand, SEA_ML metamodel specifically focuses on agent-SWS interaction. As a result of that, *Agent_SWS_Interaction* fact in Figure 27 guaranties that if there is a *WebService* in an Environment, there is at least one interaction between an agent and that web service (over related web service's semantic interface). Line 7 in Figure 27 stipulates that each Environment has a *Web Service*. This provides a *SWS* in the environment since a *WebService* requires at least one *SWS* as a precondition of implication. There are two ways which provide the interaction between an agent (SWA) and a SWS. *sws1* represents the first way which yields that a *SWA* plays a *Role* and this *Role interacts_with* the *SWS.* On the contrary, *sws2* represents the second way for agent-SWS interaction which means a *SS_FinderPlan* is applied by a *SWA* and this plan discovers an *Interface* and the *Interface* presents the *SWS*. Finally cardinality sum of *sws1* and *sws2* should be at least one. The other ways from *SWA* through the plan types to *SWS* are not added as a constraint, because the other plan types cannot be applied without an existence of a *SS_FinderPlan*. In other words, if there is a *SWS* in the environment, a *SWA* should interact with it anyway. In order to make a clear understanding of this semantics, the visualization of this constraint's application is illustrated in Figure 28. Path 1 represents *sws1* variable and path 2 represents *sws2* variable in *Agent_SWS_Interaction* fact.

A SWA can apply all kinds of plan types. However, in this system a SWA focuses on finder, agreement and execution plan types and registration is not its task. But according to inheritance, a SWA can apply *SS_RegisterPlan* as it extends Plan. Therefore, *InheritanceBreak* fact is added to break the effect of this inheritance (see Figure 27). In Line 17, this control is fulfilled. *SS_MatchmakerAgent*'s task is to register the services, advertise them and help SWAs to find them.

```
01 fact ServiceComposition{
02   all s:Service | s.~has != none
03   all wb:WebService | wb.~composed_of != none
04 }
05 fact Agent_SWS_Interaction{
06   all e: Environment| some ws:WebService
07   ws in e.hasService =>
08   {some swa1,swa2:SWA, sws1,sws2:SWS, r:Role,
09   t1,t2,t3,t4:Time, f:SS_FinderPlan, i:Interface, x:Int
10   |swa1.plays.t1= r && r.interacts_with.t2=sws1
11   && swa2.applies.t3 = f && f.discovers.t4 =i &&
12   i.presents= sws2
13   && #sws1 =x && x.plus[#sws2] >=1
14   }
15 }
16 fact InheritanceBreak{
17   no a:SWA,rp:SS_RegisterPlan, t:Time|  a.applies.t= rp
18 }
```

Figure 27: Static semantics control for Agent–SWS Interaction

Another behavioral control is given with the *InterfaceControl* fact in Figure 29. This control restricts meta-elements such as *SS_FinderPlan*, *SS_AgreementPlan* and *SS_ExecutorPlan* to reach an unregistered *Interface*. In other words, a *SS_FinderPlan* should try to discover a new *Interface* which is in the set of *Interface*(s) that is advertised by a *SS_RegisterPlan* earlier (Line 4). Analogously, a *SS_AgreementPlan* should try to negotiate with an *Interface* which is in the set of *Interface*(s) discovered previously (Line 5 in Figure 29). It is also similar for a *SS_ExecutorPlan*'s *Interface* access (Lines 7- 8). For this reason, (**in**) relations on *Interface* subset are held in this fact.

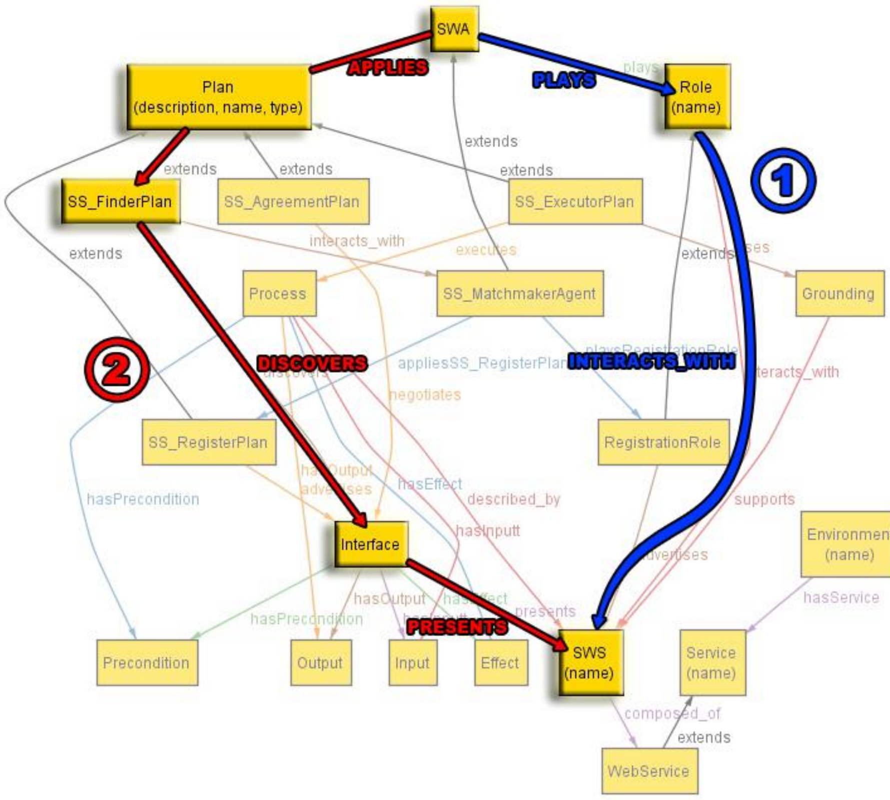


Figure 28: Agent–SWS Interaction Paths

```
01  fact InterfaceControl{
02    all f:SS_FinderPlan, r:SS_RegisterPlan,
03    a:SS_AgreementPlan | some   t1,t2,t3: Time|
04    f.discovers.t3 in  r.advertises.t1 &&
05    a.negotiates.t1 in f.discovers.t2
06    all i:Interface, p:Process, g:Grounding, e:SS_ExecutorPlan |
07    p in e.executes && g in e.uses &&
08    p.described_by in i.presents  && g.supports in i.presents
09  }
```

Figure 29: Behavioral controls

Additionally, the *SWS* which is supported by a Grounding that a *SS_ExecutorPlan* uses and the *SWS* element which is described by a *Process* that a *SS_ExecutorPlan* executes should be in the *SWS* set which is presented by an *Interface* (Lines 6-8 in Figure 29).

In our study, behavioral and dynamic semantics are especially detailed for supporting the execution ordering of SWA *Plan*s. Required state transitions are illustrated with two state diagrams as depicted in Figure 30. Figure 30-a focuses on the sequence of plan types that needs an exact order and Figure 30-b focuses on the execution of all plan types which handle cascading records of SWS discovery, agreement with SWS and execution of SWS processes. It draws the whole procedure of agent-SWS interaction steps within plan types.

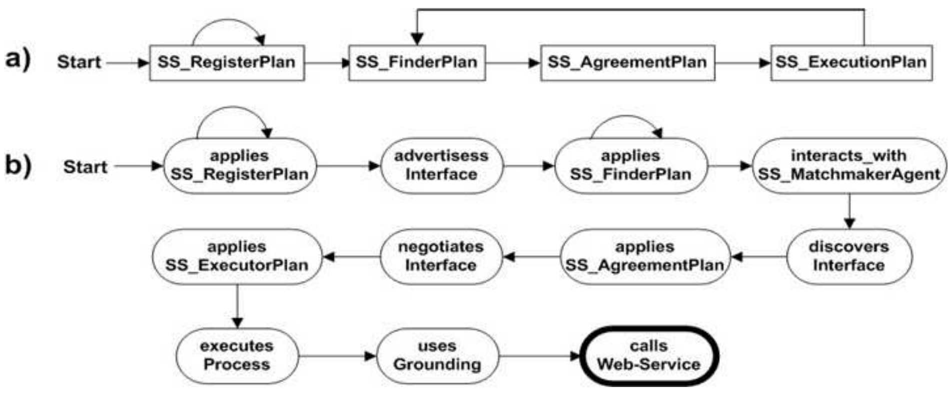


Figure 30: State diagram of Plan types in SEA_ML. a) execution order of Plan types b) Agent-SWS Interaction Procedure

To order the Plan states, we used *util/ordering* module of Alloy. This is appropriate to define the order of plan types for the intra-plan control. These transitions are provided with *PlanStates* fact (Figure 31) which explains that previous element of a *SS_FinderPlan* can be a *SS_RegisterPlan* (Line 3), previous element of a *SS_AgreementPlan* can be a *SS_FinderPlan* (Line 4) and finally previous element of a *SS_ExecutorPlan* can be a *SS_AgreementPlan* (Line 7). This order provides a dependency among plan types for the SWS Interaction process.

```
01 fact PlanStates{
02    all disj f:SS_FinderPlan| some   r:SS_RegisterPlan|
03       prevs[f]=r
04    all  a:SS_AgreementPlan|  some f:SS_FinderPlan |
05       prevs[a] =f
06    all e:SS_ExecutorPlan|some a:SS_AgreementPlan |
07       prevs[e]=a
08 }
```

Figure 31: Semantics of plan state transitions

We model the inner relation ordering from the beginning of the interaction between agent and *SWS* until the execution of *SWS*. *SWSInteractionProcedure* fact in Figure 32 handles this procedure. Line 6 extracts the times of relations "SS_MatchmakerAgent applies SS_RegisterPlan" and "SS_RegisterPlan advertises Interface" to the *t1* and *t2* time variables respectively. In Line 7, we order them in such a way that events pertaining to *t1* should be realized before *t2* (prev[t2]=t1).We use *util/ordering* module to order the times as well, since a definition like t1<t2 is not allowed in Alloy. While *prev[]* and *next[]* are used for the predecessor and successor element, *prevs[]* and *nexts[]* are used for an element of processor and successor sets. Line 8 extracts the time "SWA applies the SS_FinderPlan" and assigns it to the time *t3*. Before applying the *SS_FinderPlan*, at any time, there should be a registration in the previous events. Therefore, we add the prevs[t3]=t2 constraint. On the other hand, roles played by a *SWA* or *SS_MatchmakerAgent* can be realized at any time in the system.

Line 9 in Figure 32 extracts the times of events "SS_FinderPlan interacts_with SS_MatchmakerAgent" and "SS_FinderPlan discovers Interface" and orders in such a way that t3<t4<t5 (Line 10). Similar assignments of *t6* and *t7* are handled for the events "SWA applies SS_AgreementPlan" and "SS_AgreementPlan negotiates Interface". If the result of *SS_FinderPlan* (finding_result) exists, we order the events in the order of t5<t6<t7 (Line 12 in Figure 32), otherwise, event times of "SWA applies SS_AgreementPlan" and "SS_AgreementPlan negotiates Interface" are assigned as an empty set (Line 13 in Figure 32). Analogously, in Lines 13, 14 and 16, the times *t8*, *t9*, *t10*,*t11* are assigned to events "SWA applies SS_ExecutorPlan", "SS_ExecutorPlan executes Process", "SS_ExecutorPlan uses Grounding" and

"Grounding calls WebService" respectively. If the result of SS_AgreementPlan (agreement_result) is negative (Line 14 in Figure 32), *t8* and *t9* will be assigned as empty sets (not applied) (Line 15). Otherwise, we order the events in an ordering such as t8<t9<t10<t11 (Lines 15 and 17 in Figure 32).

```
01  fact SWSInteractionProcedure {
02    all a: SWA, ma:SS_MatchmakerAgent,  rp:SS_RegisterPlan,
03    fp:SS_FinderPlan,  ap:SS_AgreementPlan,ep:SS_ExecutorPlan,
04    i:Interface, p:Process,   g:Grounding,ws:WebService | some
05    t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11: Time |
06    ma.appliesSS_RegisterPlan[rp]=t1 && rp.advertises[i]   = t2 &&
07    prev[t2]=t1  &&
08    (a.applies[fp] =   t3&&  prevs[t3]=t2 &&
09    fp.interacts_with[ma]=t4 &&  fp.discovers[i]=t5 &&
10    prev[t5] = t4 &&   prev[t4]=t3) && (a.applies[ap] = t6   &&
11    ap.negotiates[i]=t7&& (fp.finding_result = True =>
12    (next[t5]=t6 &&  next[t6]= t7) else
13    (t6=none && t7=  none)))&&(a.applies[ep] = t8 &&
14    ep.executes[p]=t9  && (ap.agreement_result!=True =>
15    (t8=none && t9=  none) else (next[t7]=t8 && next[t8]=t9 &&
16    (ep.uses[g] =  t10&& g.calls[ws] =t11 &&
17    (next[t9] =  t10 && next[t10] = t11)))))
18  }
```

Figure 32: Semantics of Agent–SWS interaction based on Time

The "Time" column is added for ordering the relations during agent–SWS interaction. Every event is realized in a specified time. System sequence is provided by ordering these events, in other words, times of events. This constraint is important because it represents the events based on time. Time ordering gives a representation to sort every event in an exact order. However, this constraint needs a huge memory and time complexity during the analysis and creating the subset space as is discussed in Section 4.1 of this paper. Therefore, another type of constraint with subset definitions, which provides the meaning of ordering plan types by reducing to two dimensional relations, is also supplied in our study (Figure 33). According to *Agent-SWSPlanOdering* constraint, if a *SS_FinderPlan* is applied by a *SWA*, at least one *SS_RegisterPlan* should be applied by a *SS_MatchmakerAgent* (Lines 3- 4 in Figure 33). Other plan types are controlled in the same way. If *SS_AgreementPlan* is applied, *SS_FinderPlan* should already be in the related set. In other words, "SS_FinderPlan should be applied before SS_AgrementPlan" constraint is provided (Lines 5-6). Same constraint is applied for *SS_ExecutorPlan* in Lines 7-8.

```
01 fact Agent_SWSPlanOrdering {
02   all swa:SWA, sm:SS_MatchmakerAgent|
03   (SS_FinderPlan in swa.applies =>
04       #(sm.appliesSS_RegisterPlan) >=1)&&
05   (SS_AgreementPlan in swa.applies  =>
06       SS_FinderPlan  in  swa.applies) &&
07   (SS_ExecutorPlan  in swa.applies =>
08       SS_AgreementPlan in swa.applies)
09 }
```

Figure 33: Semantics of Agent–SWS process based on Subset definitions

4. Formal Model Analysis

Model analysis contributes in three ways to the abstraction of software. Firstly, it supports to simulate some possible scenarios by generating concrete examples. Secondly, it keeps the model and instance consistent. Finally, it can extract the faults which could be seen later (Jackson, 2012). On the other hand, model checking and model analysis are becoming critical in the use of DSMLs. Since DSMLs deal with complex systems' domains, they have huge instances and models. Therefore, it needs a system simulation and checking in the abstract level before applying the system. For example, complicated structure of some agent behaviors or interactions of agents with semantic web services should be taken into account during the development of SEA_ML instance models.

Development with Alloy specification language is also supported with a fully automated analyzer tool which visualizes and checks the models, and produces instances. Every analysis in this tool works through the aim of solving a constraint that either produces a counter-example or produces an instance. Alloy analyzer translates constraints (facts) to Boolean constraints and then these constraints are transferred to an off-the-shelf SAT solver (Moskewicz et al., 2001; Goldberg and Novikov, 2002).

Alloy model analyzer is based on the idea of finding counter-examples and witnesses which come from model checking (Clarke et al., 2000). This idea is applied with scope size which defines the maximum number of instances for every element in the instance model that Alloy analyzer generates. Counter-examples find the system faults by generating the negative formula of claim. Hence, they can detect the possible errors according to the assertions.

On the other hand, Alloy simulates the possible scenarios by generating some combinations from instance space. It is also possible to specify an instance model and check it. In this case, analyzer looks for this model inside the instance model combination sets. It does not mean that Alloy finds the model which the user intended if no restriction is applied. However, if a user specifies the predicates and restricts the scope for every instance, it is possible to create intended model within this scope. If a model is not found, it means that there is no instance model that satisfies the needs of the intended model; in other words Alloy cannot find an instance for that specific scope. Assertion checking or model finding can be performed in some scope. As the scope size increases, it may take too much time to find a result. Hence, scope size is a limitation of Alloy.

Considering the DSML perspective, the analyzer has a model structure control. When the analyzer is executed, it controls all sets. Some static semantics which come from the metamodel such as multiplicity relations can be provided easily in set definitions. Dynamic semantics can be defined based on logic and simulated with the analyzer by using the Time column to observe system behavior in runtime. Analyzer does not only check the runtime execution of a rule, but also it detects the inconsistencies among all constraints (facts) and set definitions. Following subsections discuss scope analysis and use of the defined semantics within a case study.

4.1 Scope Analysis

Scope size defines the maximum number of element instances in a model. Every analysis scans all instances in the space of defined scope until finding an instance. If there is not any instance, the result returns null. If the command is an assertion it means that there is no example which disproves the formula in that scope. Unfortunately it does not guarantee that there is no instance in a larger scope. If the command is a predicate, it does not have this kind of scenario in this scope, but it may have in larger scopes. Default scope size is three in Alloy. Scope size can be specified differently for all elements in the model. If Alloy analyzer finds an instance or a counter-example, it means that it will find in the larger scope as well. This case is called Scope Monotonicity. Hence, it provides simplicity for instance models or scenarios.

Property Checking:
We provided model validation with particular assertions in particular scopes. Scope size defines the maximum number of every super set (non-subset) in the instance model. According to the relations in the model, we can define a scope size which can be increased step by step until finding an example. As the scope size increases, it may take hours to have a result. However, it is quite valuable if we can show validation of the model for a possible scope size.

We created some assertions according to SEA_ML properties and obtained results in different scopes. Properties are held for agent-SWS interaction viewpoint since it is crucial for evaluating SEA_ML capabilities. Some of the defined assertions are given in Figure 34. All assertions are checked in a computer with Intel i7 1.73 Ghz CPU and 4 GB RAM. Achieved results are presented in Table 1.

SWSInteractionProcedure fact given previously in Figure 32 creates a huge space for analysis. Assertions in Figure 34 were tried to be tested with this constraint. However, even for the scope size 4, it lasted 3 hours and resulted with out-of-memory error in the computer with above mentioned configuration. The same example was tried for one month with a better computer which has Intel i7 3.20 Ghz CPU and 16 GB RAM. No result was obtained after one month nonstop execution. Therefore, to reduce the space from triples to binary, *Agent_SWSPlanOrdering* fact (Figure 33), which gives the same meaning in a different way, is considered for simulations and property checking.

During Agent-SWS interaction, *SWA*'s plan types are expected to be applied in an order. *PlanTypeProperty* assertion (Figure 34) claims that if the number of *SS_AgreementPlan* is greater than or equal to 1 (which means an *SS_AgreementPlan* exists in the instance model being processed), *SS_FinderPlan* is also greater than or equal to 1. Same stands for *SS_AgreementPlan* and *SS_ExecutorPlan*. It is expected that there is no counter-example which breaks this order and we experienced no counter example until scope size 25 (Table 1). This scope size is selected based on our processing machine power and implies that all combinations of maximum 25 elements for each signature are considered to find possible instances. In the system, services should be registered by *SS_MatchmakerAgent* before a SWA applies a plan and executes them. Hence, *RegistrationProperty* claims that if the set of Plans which SWA applies is not empty then *SS_RegisterPlan* set, which *SS_MatchmakerAgent* applies, should not be empty too.

```
assert PlanTypeProperty {
  all fp: SS_FinderPlan, ap:SS_AgreementPlan, ep:SS_ExecutorPlan|
  #ap>=1 => #fp >=1 && #ep >=1 => #ap >=1
}
assert RegistrationProperty{
  all swa:SWA,  sm:SS_MatchmakerAgent|
  swa.applies !=none => sm.appliesSS_RegisterPlan ! = none
}
assert NoConflictProperty{
  no ma:SS_MatchmakerAgent|
  some rp:SS_RegisterPlan | ma.applies= rp
}
assert EnvironmentProperty{
  no wb:WebService|#wb.~has=0
}
```
Figure 34: Assertions pertaining to the agent-SWS interaction viewpoint

A SWA can apply different kinds of plans during its interaction with SWS. Since *SS_MatchmakerAgent* is a specialization of *SWA*, naturally it inherits "*applies*" relation from SWA. As mentioned before, applying *SS_RegisterPlan* is a plan type that can only be applied by *SS_MatchmakerAgent* instances. However, the relation between *SS_RegisterPlan* instances and *SS_MatchmakerAgent* instances is not represented with the ordinary "*applies*" relationship. It is represented with "*appliesSS_RegisterPlan*". Therefore the constraint called *InheritanceBreak* is provided (See Figure 27) to prevent accidentally establishing "*applies*" relation between a *SS_MatchmakerAgent* and a *SS_RegisterPlan*. *NoConflictProperty* claims that a *SS_MatchmakerAgent* does not have *applies* relation with *SS_RegisterPlan* because it has another relation to access the same *SS_RegisterPlan*.

EnvironmentProperty claims that a *WebService* can exist inside an environment. More precisely, the container set which contains a *WebService* is a non-empty set. No counter-example is expected because of the composition control of these two elements. However analyzer results a counter-example in a large scope (see Table 1). Therefore this constraint was investigated again and changed as follows. No counter-example is found in a larger scope after that modification.

```
assert EnvironmentProperty2{
   no wb:WebService|wb.~has !=none
   }
```

Table 1: Results of scope analysis on some of the properties of Agent-SWS Interaction viewpoint

Assertion	Scope Size	Counter-examples	Elapsed time (ms)	Number of Clauses
PlanTypeProperty	3 4 10 25	No counterexample is found, assertion may be valid.	842 125 374 1357	5474 9857 89465 1567426
PlanTypeProperty	50	Fatal error: Memory exceed	-	-
EnvironmentProperty	5 10 15	No counterexample is found, assertion may be valid	115 260 380	17127 92925 304693
EnvironmentProperty	20	Counterexample is found. Assertion is invalid.	8967	304693
RegistrationProperty	10 25 30	No counterexample is found, assertion may be valid	246 1736 2271	92862 1590572 2964157
NoConflictProperty	10 20 30	No counterexample is found, assertion may be valid	360 1305 2982	92893 758873 2964248

Model Finding:
As the second task of the analyzer, predicates can generate instance models in a visual or textual manner by searching a binding that is true for model formula. Further, in the case that the user specifies the predicates, defines properties of the instance model, and restricts the scope for every instance, an intended model can be created within this scope in the instance set. If the analyzer finds an example in a scope, Alloy claims that it will also find an instance in larger scopes on the basis of scope monotonicity. If it cannot find any example in a reasonable scope (due to computer memory and/or time limitations), it means that Alloy cannot find an instance model according to the specifications in the predicate in that scope. Nevertheless, there may be an instance model in a bigger scope.

Within our study, different predicates are experienced in different scopes and resulted in Table 2. Some predicates are presented in Figure 35. As it is possible to create models without any input, first of all *pred simple {}* is run to control the constraint whether they are consistent with each other. Screenshot in Figure 36 is created in the scope size 1, 2, 1 for *SWA*, *Role* and *Plan* elements respectively. It simulates that a SWA plays different Roles at different times.

```
pred simple {}
pred Initialize {
  one appliesSS_RegisterPlan
}
pred SWAstart { some SWA && one SS_MatchmakerAgent &&
one SS_FinderPlan && SS_MatchmakerAgent.applies = none
}
```

Figure 35: Predicates for agent-SWS interaction viewpoint

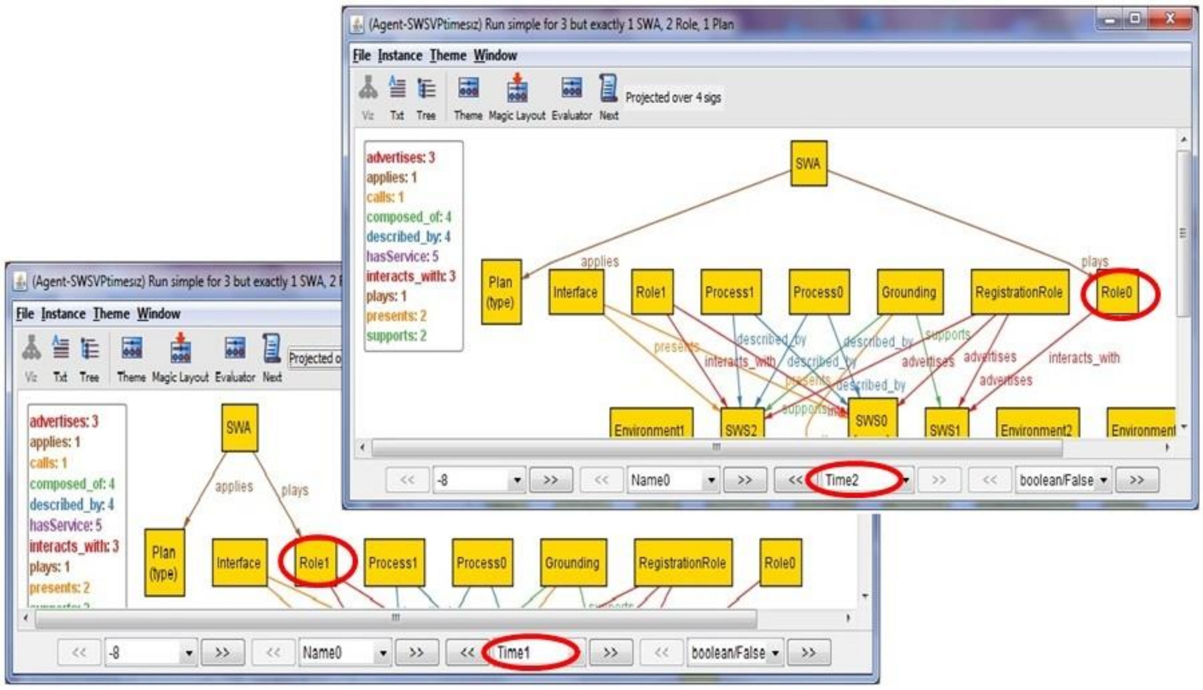


Figure 36: *Pred Simple* simulation screenshot. A SWA can play different roles at different times

Note that the projection feature of Alloy is used during model generation. When a predicate is run and an instance model is found, signature instances exist as elements in the model as can be seen in Figure 36. But some elements, for example Name and Time, do not belong to the metamodel. Therefore, it is not required to keep them as an instance element and instead, their projections are used as seen in Figure 36 for Time instance. Projection is also used for attribute elements in the metamodel. For example, name, description and property seem as some attributes since Name set is projected. Time projection also provides evidence of the behavior of the system at different times by generating different instance models for the same predicate.

Initialize predicate in Figure 35 represents the initialization of the system. *SS_MatchmakerAgent* applies *SS_RegisterPlan* and plays *RegistrationRole*. System starts with the Registration. The smallest scope size is found with 3 and 2 for Plan (Table 2).

SWAstart predicate executes the system. In this scenario, a *SWA* enters the system and applies a *SS_FinderPlan* to fulfill the user's request. Before a *SWA*, a *SS_MatchmakerAgent* should have already been in the system for registration of semantic web services. The smallest scope size is fixed as 3 and 2 for SWA and Plan respectively. Example atoms are represented in Table 2.

Table 2: Results of scope analysis within Agent-SWS Interaction viewpoint and model finding

Predicate	**Scope Size**	**Instance Model**	**Spent Time (milliseconds)**	**Number of Clauses**
Initialize	2, **exactly** 1 Plan	Not found. Predicate may be inconsistent.	401	1467
Initialize	2, **exactly** 2 Plan	Not found. Predicate may be inconsistent.	47	2375
Initialize	3, **exactly** 1 Plan	Not found. Predicate may be inconsistent.	275	3362
Initialize	3, **exactly** 2 Plan	Pred is consistent: univ={-1, -2, -3, -4, -5, -6, -7, -8, 0, 1, 2, 3, 4, 5, 6, 7, Environment$0, Interface$0, Name$0, Name$1, Name$2, Plan$0, RegistrationRole$0, Role$0, SS_MatchmakerAgent$0, SS_RegisterPlan$0, SWS$0, Time$0, Time$1, Time$2, WebService$0, aplan/Ord$0, atime/Ord$0, boolean/False$0, boolean/True$0}	109	4459
SWAstart	2	Not found. Predicate may be inconsistent.	2142	468
SWAstart	2, **but exactly** 2 Plan, 2 SWA	Not found. Predicate may be inconsistent.	2142	172
SWAstart	2 but exactly 3 Plan, 3 SWA	Not found. Predicate may be inconsistent.	3413	125
SWAstart	3, **but exactly** 3 Plan, 3 SWA	Found. Smaller scope size is tested.	5351	561
SWAstart	3, **but exactly** 2 Plan, 2 SWA	{-1, -2, -3, -4, -5, -6, -7, -8, 0, 1, 2, 3, 4, 5, 6, 7, Environment$0, Grounding$0, Interface$0, Name$0, Name$1, Name$2, RegistrationRole$0, Role$0, SS_FinderPlan$0, SS_MatchmakerAgent$0, SS_RegisterPlan$0, SWA$0, SWS$0, Time$0, Time$1, Time$2, WebService$0, aplan/Ord$0, atime/Ord$0, boolean/False$0, boolean/True$0}	3819	141

4.2 Case study: An Agent-based E-barter System

In this section, we discuss the design of an agent-based electronic barter (e-barter) system in order to give some flavor of the use of SEA_ML's formal semantics. An agent-based e-barter system consists of agents that exchange goods or services for their owners without using any currency. In our example, a Barter Manager agent (shown in Figure 37), who is implemented as a SWA, manages all trades in the system. This agent is responsible for collecting barter proposals, matching proper barter proposals and tracking the bargaining process between customer agents. To infer about semantic closeness between offered and purchased items based on some defined ontologies, barter manager may use SWS. Conforming to its Barter Role definition, Barter Manager needs to discover the proper SWS, interact with the candidate service and realize the exact execution of the SWS after an agreement. More information on the development of such a system can be found in (Demirkol et al., 2011).

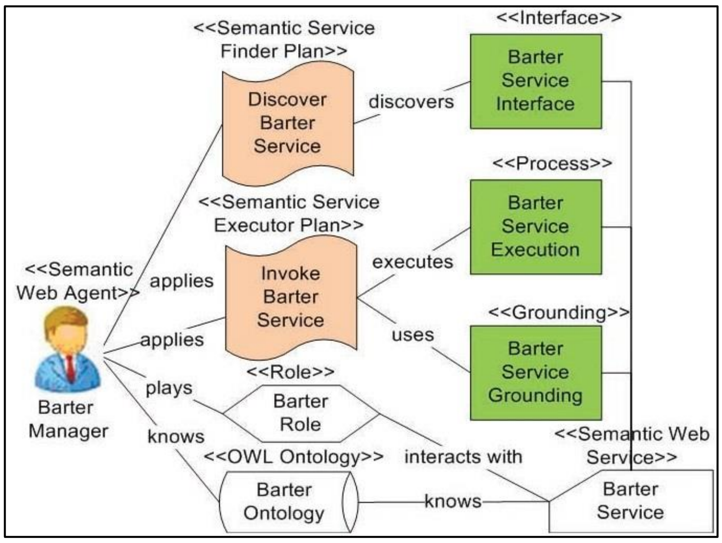


Figure 37: e-Barter Scenario (illustration is taken from (Kardas et al., 2010))

In the system, suppose that a Barter Manager agent needs to interact with semantic web services to match bidden and demanded goods and determine the value of the exchange. For instance, two customer agents (one from the automotive industry and other from the healthcare sector) may need to exchange their offered goods and services such that: A car manufacturer offers to sell car spare parts to a health insurance company (e.g., for company's service cars) and wants to procure health insurance for its employees. Consider that the intention of the health insurance company is vice versa. During the bargain between the agents of the car manufacturer and the health insurance company, our Barter Manager agent may use SWS called Barter Service. In order to invoke that service, Barter Manager first needs to discover the proper semantic web service. Then, Barter Manager interacts with the candidate service(s) and after an agreement; the exact execution of the semantic web service is realized (Kardas et al., 2010).

SWA, SS_FinderPlan, SS_AgreementPlan, SS_ExecutorPlan, Role, Interface, Process, Grounding and *SWS* elements are used in modeling of the e-barter system according to SEA_ML's agent-SWS interaction viewpoint. For instance, *BarterManager* is a kind of SWA. This agent applies *Discover, Haggle* and *Invoke* plans which are instances of *SS_FinderPlan*, *SS_AgreementPlan* and *SS_ExecutorPlan* respectively. *BarterManager* agent plays *BarterRole*. For Barter operations, it uses *BarterService* which is a kind of *SWS. BarterSevice* owns appropriate interface and execution mechanism. In order to create the model of the e-barter system, *eBarter* predicate (Figure 38) is written. It is worth noting that predicates of such instance models can only be written manually due to Alloy restrictions. Alloy does not provide a graphical editor to visually create or modify the instance models which may lead also to the automatic generation of the required predicates. Currently, Alloy only provides a visual and an uneditable representation of a model after creating this instance model with manually given predicates.

In order to execute *eBarter* predicate, scope size for Plans is defined as 4, since the number of Plan types is 4. Furthermore, *RegistrationRole*, *SWS*, *Interface*, *Process*, *Grounding*, *WebService* and *Environment* instances are created exactly as (1,1,1,1,1,1,1). Each instance is an argument in the predicate such as "BarterManager is a SWA". *BarterManager*, *BarterRole*, *BarterService*, *BSInterface*, *BSGrounding*, *BSProcess*, *TradingService*, *Discover*, *Haggle* and *Invoke* are arguments of the predicate. In the body of the predicate, the relation of instances can be defined. If there are wrong bindings, analyzer will give the "inconsistent model" result. In line 5 of Figure 38, the given constraint is to provide *BarterManager* not to be a *SS_MatchmakerAgent* in this system. Therefore, it applies all plans except the one for the service registration (Line 6).

Model of the system is generated according to the above defined semantics (see Figure 39). The analyzer checks the relations and arguments and then generates the model if it is consistent. If the model is missing, the analyzer is capable of supplementing the instance based on SEA_ML's semantics definitions. For example *SS_MatchmakerAgent* instance is not defined in the predicate given in Figure 38. However according to SEA_ML constraints there should be at least one *SS_MatchmakerAgent* for SWS registrations. As it can be observed in Figure 39, *SS_MatchmakerAgent* has been generated automatically and the model is now completed. Nevertheless, if the user specifically does not want to define *SS_MatchmakerAgent* by assigning null in the predicate, then the analyzer cannot find any consistent instance model in any scope size since at least one *SS_MatchmakerAgent* is mandatory for the system initialization. Hence, beyond the model analysis in some scope, we can also check the instance model according to defined semantics. This provides the generation of consistent instance models for SEA_ML.

```
01 pred eBarter (BarterManager:SWA, BarterRole: Role, BarterService: SWS,
02 BSInterface: Interface,BSGrounding:   Grounding, BSProcess: Process,
03 TradingService: WebService,  Discover: SS_FinderPlan,
04 Haggle:SS_AgreementPlan, Invoke:SS_ExecutorPlan ){some t:Time|
05   BarterManager not in SS_MatchmakerAgent &&
06   SS_MatchmakerAgent.applies= none &&
07   BarterManager.plays.t = BarterRole &&
08   Discover in  BarterManager.applies  &&
09   Haggle in BarterManager.applies  &&
10   Invoke in BarterManager.applies &&
11   Discover.discovers =   BSInterface&&
12   Haggle.negotiates = BSInterface   && Invoke.executes = BSProcess &&
13   Invoke.uses=Grounding &&
14   BSProcess.described_by=BarterService  &&
15   BSGrounding.supports =BarterService&&
16   BSGrounding.calls=TradingService &&
17   BSInterface.presents=BarterService &&
18   BarterService.composed_of=TradingService
19 }
```
Figure 38: E-Barter predicate which models the e-barter system according to the agent-SWS viewpoint

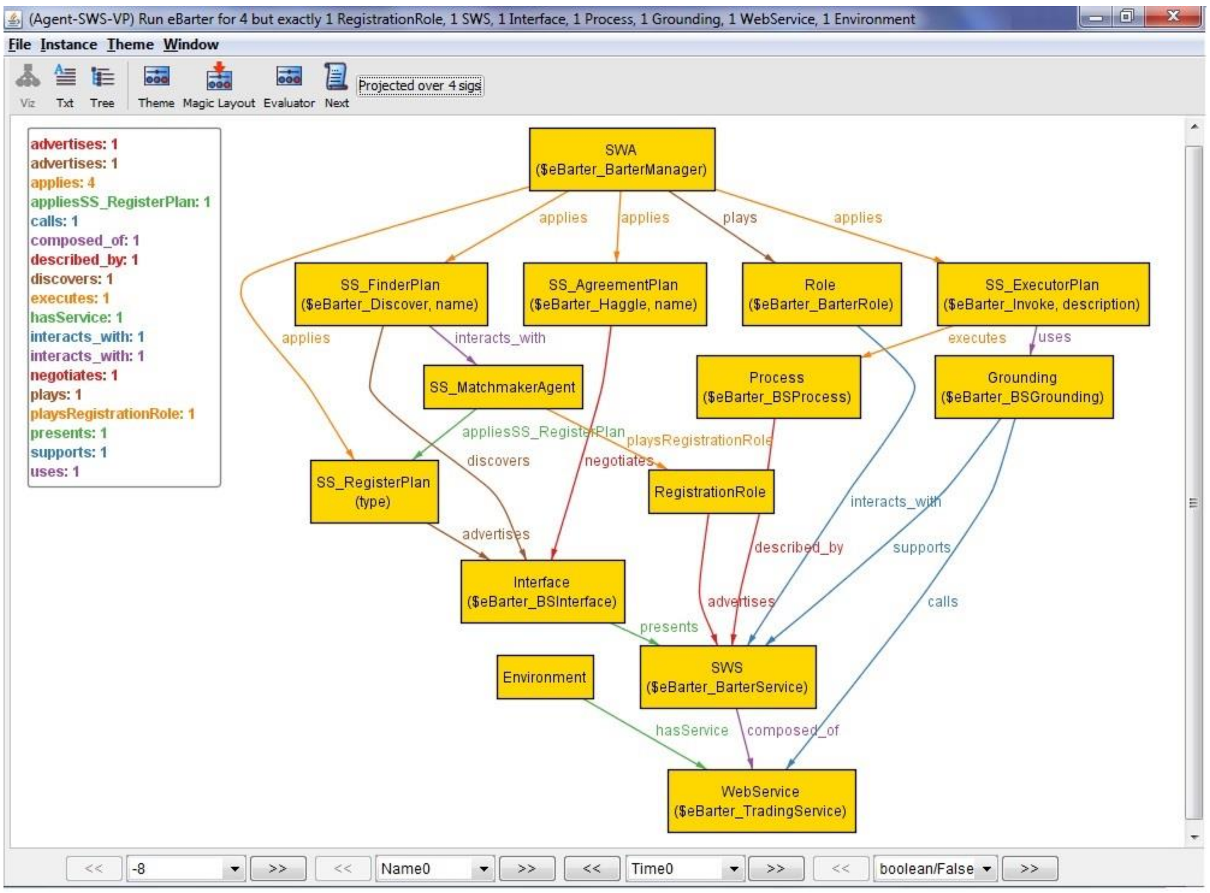
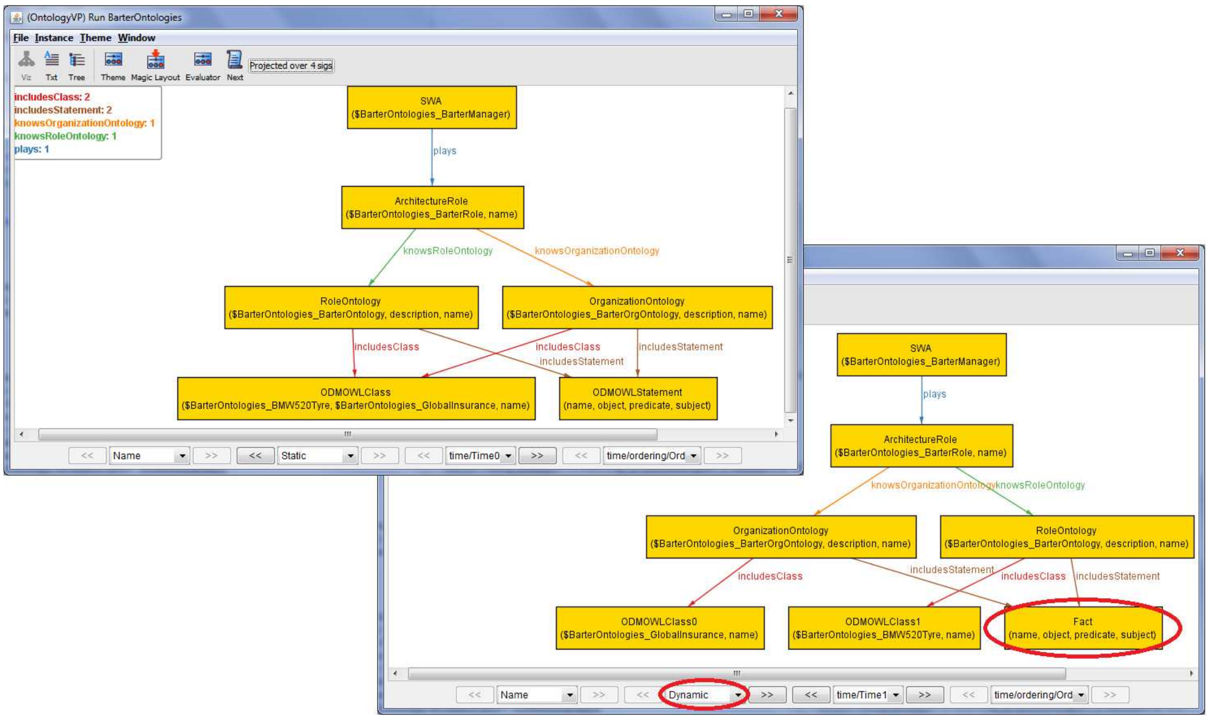

(Agent-SWS-VP) Run eBarter for 4 but exactly 1 RegistrationRole, 1 SWS, 1 Interface, 1 Process, 1 Grounding, 1 WebService, 1 Environment
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over 4 sigs
advertises: 1
advertises: 1
applies: 4
appliesSS_RegisterPlan: 1
calls: 1
composed_of: 1
described_by: 1
discovers: 1
executes: 1
hasService: 1
interacts_with: 1
interacts_with: 1
negotiates: 1
plays: 1
playsRegistrationRole: 1
presents: 1
supports: 1
uses: 1
SWA ($eBarter_BarterManager)
applies
applies
plays
applies
SS_FinderPlan ($eBarter_Discover, name)
SS_AgreementPlan ($eBarter_Haggle, name)
Role ($eBarter_BarterRole)
SS_ExecutorPlan ($eBarter_Invoke, description)
applies
interacts_with
executes
uses
SS_MatchmakerAgent
Process ($eBarter_BSProcess)
Grounding ($eBarter_BSGrounding)
appliesSS_RegisterPlan
playsRegistrationRole
discovers
negotiates
SS_RegisterPlan (type)
RegistrationRole
interacts_with
advertises
described_by
supports
Interface ($eBarter_BSInterface)
advertises
calls
presents
Environment
SWS ($eBarter_BarterService)
hasService
composed_of
WebService ($eBarter_TradingService)
-8
Name0
Time0
boolean/False

Figure 39: Generated model of the *e-Barter* system

Let us consider the Ontology viewpoint of the designed system. According to the scenario, Barter Manager agent first searches for a semantic web service which can match a “Car_Spare” OWL concept with a “Health_Insurance” OWL concept and then executes the service to find counterpart of a bargained car spare part: an OWL individual for BMW 520 Tyre. BMW520Tyre and GlobalInsurance are ODMOWLClass instances for the exchange and they are included in BarterOntology and BarterOrgOntology respectively. These ontologies are known by the BarterRole which is played by the BarterManager. BarterOntologies predicate (Figure 40) is run with the scope size 3 for all elements and we obtain the generated model shown in Figure 41 within these specifications. Agent’s belief update type is static at *Time0* (upper left snapshot in Figure 41) while it is dynamic when a new fact is generated at *Time1* (lower right snapshot in Figure 41) which means the belief base should be updated. Marking the update status of a belief base as static or dynamic originates from the related attribute specification in SEA_ML metamodel. Hence, if the belief base remains same from its initialization up to that specific runtime, it is marked as static. In case of belief (base) modification or new fact insertion, it is marked as dynamic.

```
pred BarterOntologies(BarterManager: SWA,
BarterOntology: RoleOntology,
BMW520Tyre: ODMOWLClass, BarterRole:Role, BarterOrgOntology:
OrganizationOntology,
GlobalInsurance:ODMOWLClass){
   some t:Time | BarterOntology.includesClass=BMW520Tyre
   && BarterRole.knowsRoleOntology = BarterOntology
   && BarterManager.plays.t = BarterRole && Barter
   Role.knowsOrganizationOntology=BarterOrgOntology
   && BarterOrgOntology.includesClass= GlobalInsurance
}
```

Figure 40: *e-Barter* model with Ontologies



(OntologyVP) Run BarterOntologies
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over 4 sigs
includesClass: 2
includesStatement: 2
knowsOrganizationOntology: 1
knowsRoleOntology: 1
plays: 1
SWA ($BarterOntologies_BarterManager)
plays
ArchitectureRole ($BarterOntologies_BarterRole, name)
knowsRoleOntology
knowsOrganizationOntology
RoleOntology ($BarterOntologies_BarterOntology, description, name)
OrganizationOntology ($BarterOntologies_BarterOrgOntology, description, name)
includesClass
includesStatement
ODMOWLClass ($BarterOntologies_BMW520Tyre, $BarterOntologies_GlobalInsurance, name)
ODMOWLStatement (name, object, predicate, subject)
Name
Static
time/Time0
time/ordering/Ord
ODMOWLClass0 ($BarterOntologies_GlobalInsurance, name)
ODMOWLClass1 ($BarterOntologies_BMW520Tyre, name)
Fact (name, object, predicate, subject)
Dynamic
time/Time1

Figure 41: Generated model for *BarterOntologies*

In order to demonstrate a formal check of the dynamic aspects of the developed model, let us suppose there exists a change in the scenario in the course of time; such that a new semantic web agent enters to the current system and wants to interact with the semantic web services for bartering. For this purpose, we refer to the two snapshots of the model taken in two different times. According to the first snapshot of the system (previously given in Figure 39), a barter agent succeeded all plans and found a semantic web service in the system at *Time0* (Note that it is not a real time interval property. *Time0* here just represents the time "*before*" *Time1* in temporal logic language). In this snapshot (Figure 39), a barter manager agent was looking for a web service to bargain health insurances with car spare parts (see also Figure 41) and he/she was playing the Barter role. Barter manager applied all plans to find, agree with and execute a service and hence achieved his/her goal. In *Time1* (second snapshot of the system), a new semantic web agent (called SWA0) has joined the system (see Figure 42) in order to interact again with a web service that enables bartering health insurances with car spare parts. By playing the Barter Role, *SWA0* applied a finder plan (SS_FinderPlan) to achieve this goal. However, as can be seen from Figure 42, *SWA0* currently can apply neither an agreement (SS_AgreementPlan) nor service execution (SS_ExecutorPlan) plan. The reason is that SEA_ML dynamic semantics does not allow an agent to agree with or execute a semantic web service before finding the service. In other words, this is the snapshot of the system that reflects the instant change in the scenario. It simulates the moment when the BarterManager agent just completed the application of all types of plans and a new SWA entered into the system and started to search for a new web service. At the same moment, that new agent can not apply SS_AgreementPlan or SS_ExecutorPlan. In fact, if we gave another snapshot of the system (say the third snapshot after the second snapshot) while the new agent was executing the semantic web service by applying the execution plan, we would clearly see that both the finder and agreement plans had already been applied. Those dynamic semantics checks are automatically performed for the developers during MAS modeling via utilizing the ordering module of Alloy inside SEA_ML's formal definition on the agent plan orderings based on time (as previously discussed in Section 3.8).

Finally, it is worth indicating that the Barter Manager agent had already played the Barter Role before *Time1* and according to the dynamic semantics definitions of SEA_ML, he/she now changes his/her role into another role in the environment exactly at *Time1* (Figure 42). Although it is not specified in the predicate definition, the new role, called Role 2, is automatically assigned by Alloy analyzer in order to comply with this SEA_ML's dynamic semantics constraint. Further, the new agent has started to play BarterRole which was already specified and also used by BarterManager in the past (before *Time1*).

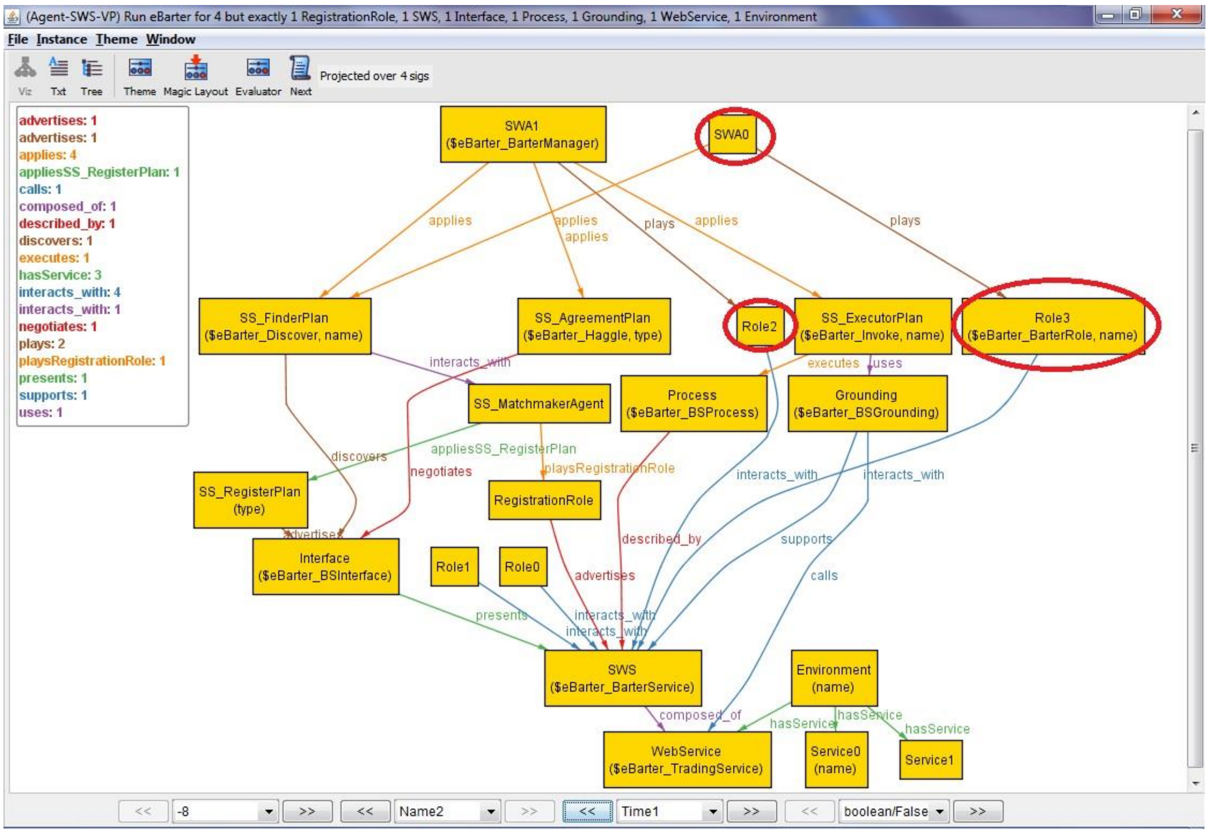


Figure 42: Generated model of the *e-Barter* system that shows a different scenario at a different time

5. Related Work

Studies on DSL and DSMLs for agents are recently emerging. For instance, a DSL called Agent-DSL is introduced in (Kulesza et al., 2005). Agent-DSL is used to specify the agency properties that an agent could have to accomplish its tasks. The proposed DSL is presented only with its metamodel for the visual modeling of the agent systems according to some agent features, such as knowledge, interaction, adaptation, autonomy and collaboration. Likewise, Rougemaille et al. (Rougemaille et al., 2008) introduce two dedicated agent 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 (Rougemaille et al., 2008). However, the study includes just the abstract syntax of the related DSMLs and neither gives the concrete syntax nor semantics of the DSMLs. In fact, the study only defines generic agent metamodels for model driven development of MASs.

Hahn (Hahn, 2008) introduces a DSML for MAS called DSML4MAS. The abstract syntax of the DSML is derived from a PIMM for agents (Hahn et al., 2009), possessing different aspects of such systems including MAS, agent, role and behavior. Hahn also discusses the use of Object-Z (Duke et al., 1995; Smith, 2000) in definition of the static semantics of the individual concepts which ensures that all concepts are statically well-formed by including the formalization of their attributes and invariants. Furthermore, DSML4MAS supports the deployment of modeled MASs both in JACK (AOS, 2001) and JADE (Bellifemine et al., 2001) agent platforms by providing an operational semantics over model transformations. In order to provide a concrete syntax, the appropriate graphical notations for the concepts and relations of DSML4MAS are defined in (Warwas and Hahn, 2008). DSML4MAS can be considered as to be one of the first complete DSMLs for agents with all of its specifications including the formal semantics (Hahn and Fischer, 2009) which will be discussed later in this section.

Another DSML is provided for MASs in (Gascuena et al., 2012) including the abstract syntax, the concrete syntax and related development tools. The abstract syntax is presented using Meta-object Facility (MOF) (OMG, 2002), the concrete syntax and its tool are provided with GMF (Eclipse, 2006), and finally the code generation for the JACK agent platform is realized with model transformations. Introduced syntax is derived from the metamodel of the well-known Prometheus (Padgham and Winikoff, 2004) MAS development methodology. Hence, Prometheus model of the system can be constructed as first. Then, intermediate code of the model is achieved by using the tools also presented in (Gascuena et al., 2012). Finally, the intermediate code is imported into the JACK Development Environment in order to provide code completion and exact system implementation. Agents on the Semantic Web and the interaction of Semantic Web enabled agents with other environment members such as semantic web services are not considered in (Gascuena et al., 2012).

Originating from a well-formalized syntax and semantics, Ciobanu and Juravle define and implement a high-level DSL for mobile agents in (Ciobanu and Juravle, 2012). A text editor with auto-completion and error signaling features is generated and a way of code generation for agent systems starting from their textual description is presented. The introduced DSL solely takes into account the mobile agents domain which differs from the domain of SEA_ML.

The service composition architecture introduced in (Fujii and Suda, 2006) dynamically combines distributed components based on the semantics of the components in order to create a web application. Implementation of the proposed architecture is based on the well-known web service definition and execution standards. Authors also propose an appropriate way of migrating existing web services into the architecture without implementing those services from scratch. In order to support collaboration of agents and web services, Sycara et al. (Sycara et al., 2003) propose a capability representation mechanism for semantic web services and discuss how they can be discovered and executed by agents. Likewise, a set of architectural and protocol abstractions that serves as a foundation for agent - web service interactions is introduced in (Burstein et al., 2005). Based on this architecture, how agents and semantic web services can be integrated are discussed in (Gümüs et al., 2007) and (Gürcan et al., 2007). Instead of semantic web service profiles, use of OWL-S process models during the service discovery is proposed in (Paulraj et al., 2011). Hence, it is aimed to find and match more relevant services with the proposed algorithm. But, service composition and execution by the agents are open issues in the study. Varga et al. (Varga et al., 2004) propose an approach in which descriptions of the agents providing the semantic web service are generated for the migration of existing web services into the Semantic Web via agents. Our study contributes to abovementioned agent-based service composition and execution studies by supporting the model-driven engineering of the interaction between software agents and semantic web services.

The work in (Kardas et al., 2009) presents a methodology based on OMG's well-known Model Driven Architecture (MDA) (OMG, 2003) for modeling and implementing agent and service interactions on the Semantic Web. A PIMM for MAS and model transformations from instances of this PIMM to two different MAS deployment platforms are discussed in the paper. But neither a DSML approach nor semantics of service execution is covered in the study. Hahn et al. (Hahn et al., 2008) define a DSML for agents and provide extensions for this DSML to integrate semantic web service execution into MAS domain. In addition to the MAS metamodel described first in (Hahn, 2008), a new metamodel, called PIM4SWS, is proposed for semantic web services. A relationship between these two metamodels is established in such a way that the MAS metamodel is extended with new meta-entities in order to support semantic web services interoperability, and it also inherits some meta-entities from PIM4SWS. That approach based on the use of two separate metamodels differs from SEA_ML's in which the modeling of agent and semantic web services' interactions is provided with the inclusion of a special viewpoint into MAS metamodel. The semantic internal components of agents, like an agent's knowledgebase, could also be modeled using SEA_ML. Moreover, presenting a dedicated metamodel for SWS brings some benefits. For instance, PIM4SWS provides the platform-independent modeling of semantic web services. After modeling, counterparts of those semantic web service models conforming to various platform-specific metamodels of SWS description languages (e.g. OWL-S) can be generated by employing structural and semantic transformations as discussed in (Klusch et al., 2008). Structural transformation is applied based on the syntactic mapping between corresponding SWS modeling concepts while semantic transformation enables formal verification of the mappings. Z formal specification language (Spivey, 1988) is used for the definition of PIM4SWS's semantic transformation. Klusch et al. (Klusch et al., 2008) also describe a model-driven semantic web service matchmaker in which semantic service selection and composition for implementing business process workflows are provided with the help of the abstraction brought by PIM4SWS.

On the other hand, there are some studies directly related to the formal semantics definition of agent systems. For instance, the study in (Hahn and Fischer, 2009) uses the Object-Z language (Smith, 2000) to define the formal semantics of DSML4MAS (Hahn, 2008). In this way, the system designer is supported in validating and verifying the generated design. An Object-Z class for each concept in the metamodel is given in order to define operational and denotational semantics. While denotational (static) semantics is provided by introducing some semantic variables and invariants, operational (dynamic) semantics is defined by introducing semantic operations and invariants. Boudiaf et al. (Boudiaf et al., 2008) present a framework to support formal specification and verification of DIMA multi-agent models using Maude language (Clavel et al., 2002) based on rewriting logic. DIMA model aims to decompose complex behavior of an agent within a set of specialized behaviors. Further, DIMA allows implementing agents having diverse granularities e.g. size, internal behavior or knowledge. Formalization of both a DIMA agent's behavior and inter-agent control mechanism is given in (Boudiaf et al., 2008). In (Hilaire et al., 2000), the authors believe that Object-Z and statecharts are not powerful enough individually to specify the complex MASs and hence they combine Object-Z and statecharts to define MASs based on an organizational model. Models are shown semi-formally over statecharts. AgentZ (Brandao et al., 2004) extends Object-Z for specifying MASs with adding new constructs to improve its structure with adding new agent-oriented entities such as agents, organizations, roles and environments. However, only the static semantics is supported while our work considers both static and dynamic semantics in MAS modeling. Furthermore, the Semantic Web environment and the interactions of agents inside this new environment are not covered in these formal semantics definition studies.

Validation of the designed agent systems by applying formal methods can also be critical during MAS development. Related worthwhile approaches are extensively discussed in (Dastani et al., 2010) and (Fallah-Seghrouchni et al., 2011). Considering the use of Alloy in MAS development, Podorozhny et al (Podorozhny et al., 2007) present an approach to design a robust MAS and check the properties of coordination, interaction, and agent's data structures using Alloy analyzer. Additionally, Haesevoets et al. (Haesevoets et al., 2010) formally define the relations between the interactions, the exposed information and provided policies and laws of an agent middleware by using Alloy. In this way, they guarantee a number of properties which are important in the use of this middleware. Any kind of full-fledged DSL or DSML is not provided in these studies.

6. Conclusion

In this paper, formal semantics and validation of MAS models, conforming to an agent DSML called SEA_ML, are presented using Alloy specifications[2]. Semantics of agent internal structures, MAS organizations and interactions between software agents and SWSs are discussed in both static and dynamic aspects with their appropriate definitions and modules. Additionally, SEA_ML instance model validations are completed by using Alloy analyzer tool. SEA_ML properties are discovered and possible scenarios, which can occur in SEA_ML domain, are observed by using formal models. Furthermore, MAS model analysis, based on both instance model generation and the application of rules pertaining to counter-example model checking, is performed. We believe that the study contributes to formal semantics definition of agent DSMLs in general and DSMLs for semantic web enabled agent systems in particular.

Modeling and validation of the interoperability between software agents and the semantic web services are achieved with the inclusion of the semantic web service entity and its related components into the definition of SEA_ML's formal semantics. Hence, based on both the defined constrains and the relations between these entities and the classical MAS entities, agent developers can design the whole MAS by including the semantic web service entities and especially checking all the behavioral and dynamic semantics of the agent-service interaction such as the execution ordering among agent plans required for the semantic web service discovery, agreement and invocation. Furthermore, correct transitions of the possible behavior flow for each plan type needed for the interaction steps are automatically supported. This may lead autonomous web service composition for agents within the semantic web environment. We believe that those features, originating from the integration of semantic web service components into the SEA_ML's formal semantics, also pave a way for the concrete implementation of the widely-known protocols (e.g. extensively discussed in (Burstein et al., 2005) and (Kumar, 2012)) which are used by the

[2] Complete SEA_ML metamodel, all written semantics rules along with instance models as Alloy files, and instructions for running them are available as a bundle at: http://mas.ube.ege.edu.tr/downloads/sea_ml.zip

agents in order to interpret and reason with semantic descriptions in the deployment of semantic web services.

Lessons learned during the development of such a MAS DSML by using Alloy are worth reporting. Alloy language provides an easy representation capability with its understandable syntax and semantics. Also, it does not require a prior modeling language experience. We found Alloy quite useful to prepare the MAS domain concepts and relations, which constitute the metamodel in terms of DSMLs. Since Alloy originates from set theory, relational logic and predicate logic, constraints on agent internals, MAS organization and service interactions can be defined based on mathematics. In fact, constraints provide the core of SEA_ML semantics. Although some relationship types such as Unified Modeling Language's (UML) composition and aggregation are not defined in Alloy, they can be obtained by using operations in constraints.

In SEA_ML, ontologies are utilized for modeling both semantic web services and agent internal belief bases. When taking into account the representation and use of these ontologies inside Alloy, we examined that the subject-predicate-object structure of RDF-based ontologies can easily be constructed in Alloy with the use of Alloy signatures and relation entities. Specifically, ontologies conforming to ODM (OMG, 2009) (e.g. ontologies prepared by using OWL) can be represented in Alloy with all of their classes, statements, properties and relations. Within this context, Alloy meets the requirements of ontological aspects of SEA_ML. In addition to our experience, studies like (Wang et al., 2006; Song et al., 2012) also show that Alloy is capable of verifying ontologies with the help of its analyzer. For instance, OWL ontologies can be parsed and converted into an Alloy model and the consistency of an ontology model can be checked automatically. That feature may enable the reasoning for these ontologies. However ontology reasoning capabilities of Alloy is not within the scope of our current work and hence not covered during our evaluation. Furthermore, Alloy's scalability limitation we encountered during model finding has also been reported in (Wang et al., 2006) for reasoning large ontologies.

Alloy analyzer is a strong analyzer which is surrounded with SAT solvers and based on model checking theorems within concepts. We observed that the generated MAS models are consistent with the expectations. Additionally, instance models can be presented as both textual and graphical. Analyzer also purveys some information about the executed predicates and assertions for the models such as spent time, number of clauses and so on.

We run the predicates of SEA_ML models by running them in different scope sizes to determine whether the models are consistent or not. The main idea is to find the desired instance in the subset space of that model considering the constraints (facts) and scope. Hence, MAS model checking is accomplished within that scope. However, when Alloy cannot find an instance or the defined model is inconsistent, source of that problem (e.g. what is the missing part and/or how the predicate should be altered) is not given by the tool. This can be considered as a disadvantage of the tool. Besides, control of the triples or analyzing the triples can get complicated and achieving a result consumes reasonable time and a huge memory.

Finally, we experienced that counter-example approach is a good way to detect the possible system errors in the abstract level for complex systems like the ones modeled via SEA_ML. Also, model finding is a suitable way to observe possible scenarios of the big systems. During these analyses, scope sizes are increased and decreased according to the results. As can be seen in Table 1 and 2, increase in the scope size does not always increase the time elapsed for achieving the results. In some cases, when the scope size is determined according to the model properties and constraints and the scope size is held different for each element, it is possible to get faster results.

In our future work, we plan to add more dynamic semantics such as message controls during the interaction between agents and sequence controls among agents. Moreover, we plan to integrate semantic checking controls introduced in this paper into the MAS DSML development tool presented in (Getir et al., 2011). Such an integration will provide both automatic generation and modification of predicates pertaining to the MAS model instances. Alloy does not currently support the modification of the generated instance text definitions as previously discussed in Section 4.2. Therefore, an integration between our graphical tool for MAS DSML and Alloy enables the creation of signature definitions and instance models automatically which also provides a convenient way for the developers to write and modify the semantic rules. In order to realize the integration, our aim is to define and execute transformations between Alloy models and Ecore (Eclipse, 2005) models that can be interpreted by the DSML tool in question.

Acknowledgements

This study is partially funded by the Scientific and Technological Research Council of Turkey (TUBITAK) under grant 109E125.

References

(Anastasakis et al., 2007) Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2007) "UML2Alloy: A Challenging Model Transformation", In proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering, Languages and Systems (MoDELS), pp. 436 - 450.

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

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

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

(Boudiaf et al., 2008) Boudiaf, N., Mokhati, F., and Badri, M. (2008) "Supporting Formal Verification of DIMA Multi-Agents Models: Towards A Framework Based on Maude Model Checking", International Journal of Software Engineering and Knowledge Engineering, Vol. 18, Issue 7, pp. 853-875.

(Brandao et al., 2004) Brandao, A.A.F., Alencar, P., and de Lucena, C.J.P. (2004) "AgentZ: Extending Object-Z for Multi-agent Systems Specification", Lecture Notes in Artificial Intelligence, Vol. 3508, pp. 125-139.

(Bryant et al., 2011) Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., and Karsai,G. (2011) "Challenges and directions in formalizing the semantics of modeling languages", Computer Science and Information Systems, Vol. 8, Issue 2, pp. 225-253.

(Burstein et al., 2005) Burstein, M., Bussler, C., Zaremba, M., Finin, T., Huhns, M.N., Paolucci, M., Sheth, A.P., and Williams, S. (2005) "A semantic web services architecture", IEEE Internet Computing, Vol. 9, Issue 5, pp. 72–81.

(Challenger et al., 2011) Challenger, M., Getir, S., Demirkol, S., and Kardas, G. (2011) "A Domain Specific Metamodel for Semantic Web enabled Multi-agent Systems", Lecture Notes in Business Information Processing, Vol. 83, pp. 177-186.

(Ciobanu and Juravle, 2012) Ciobanu, G., and Juravle, C. (2012) “Flexible Software Architecture and Language for Mobile Agents”, Concurrency and Computation: Practice and Experience, Vol. 24, Issue 6, pp. 559-571.

(Clarke et al., 2000) Clarke E.M., Grumberg O., Doron A.P. (2000) "Model Checking", MIT Press, 330p.

(Clavel et al., 2002) Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J.F. (2002) " Maude: Specification and Programming in Rewriting Logic", Theoretical Computer Science, Vol. 285, Issue 2, pp. 187-243.

(Dastani et al., 2010) Dastani, M., Hindriks, K.V., and Meyer, J.-J. (2010) "Specification and Verification of Multi-agent Systems (1st edition)", Springer, 405p.

(Demirkol et al., 2011) Demirkol, S., Getir, S., Challenger, M., and Kardas, G. (2011) "Development of an Agent based E-barter System", In proceedings of 2011 International Symposium on Innovations in Intelligent Systems and Applications (INISTA 2011), Istanbul, Turkey, IEEE Computer Society, pp. 193-198.

(Duke et al., 1995) Duke, R., Rose, G., and Smith, G. (1995) "Object-Z: A specification language advocated for the description of standards", Computer Standards & Interfaces, Vol. 17, Issue 5-6, pp. 511-533.

(Eclipse, 2005) Eclipse Consortium (2005) "Eclipse Modeling Framework", available at: http://www.eclipse.org/modeling/emf/ (last access: November 2013).

(Eclipse, 2006) Eclipse Consortium (2005) “Graphical Modeling Framework (GMF)”, available at: http://www.eclipse.org/modeling/gmp/ (last access: November 2013).

(Fallah-Seghrouchni et al., 2011) Fallah-Seghrouchni, A.E., Gomez-Sanz, J.J., Singh, M.P. (2011) "Formal Methods in Agent-Oriented Software Engineering", Lecture Notes in Computer Science, Vol. 6038, pp 213-228.

(Ferber, 1999) Ferber, J. (1999) "Multi-agent Systems: An Introduction to Distributed Artificial Intelligence", Addison-Wesley Professional, 528p.

(Finin et al., 1994) Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994) "KQML as an agent communication language", In proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM 1994), ACM Press, pp. 456-463.

(FIPA, 2002a) IEEE Foundation for Intelligent Physical Agents (FIPA) (2002) "FIPA Standards", available at: http://www.fipa.org, (last access: November 2013).

(FIPA, 2002b) IEEE Foundation for Intelligent Physical Agents (FIPA) (2002) "FIPA Agent Communication Language Specification", available at: http://www.fipa.org/repository/aclspecs.html, (last access: November 2013).

(Fowler, 2011) Fowler, M. (2011) "Domain-specific Languages", Addison-Wesley Professional, 640p.

(Fujii and Suda, 2006) Fujii, K. and Suda, T. (2006) "Semantics-based Dynamic Web Service Composition”, International Journal of Cooperative Information Systems, Vol. 15, Issue 3, pp. 293–324.

(Gascuena et al., 2012) Gascuena, J.M., Navarro, E., and Fernandez-Caballero, A. (2012) "Model-Driven Engineering Techniques for the Development of Multi-agent Systems", Engineering Applications of Artificial Intelligence, Vol. 25, Issue 1, pp. 159–173.

(Getir et al., 2011) Getir, S., Demirkol, S., Challenger, M., and Kardas, G. (2011) "The GMF-based Syntax Tool of a DSML for the Semantic Web enabled Multi-Agent Systems", In proceedings of the Workshop on Programming Systems, Languages, and Applications based on Actors, Agents, and Decentralized Control (AGERE! 2011), held at the 2nd Systems, Programming, Languages and Applications: Software for Humanity Conference (SPLASH 2011), Portland, USA, ACM Press, pp. 235-238.

(Getir et al., 2012) Getir, S., Challenger, M., Demirkol, S., and Kardas, G. (2012) "The Semantics of the Interaction between Agents and Web Services on the Semantic Web", In proceedings of the 7th IEEE International Workshop on Engineering Semantic Agent Systems (ESAS 2012), held in conjunction with the 36th IEEE Signature Conference on Computers, Software, and Applications (COMPSAC 2012), IEEE Computer Society, pp. 619-624.

(Goldberg and Novikov, 2002) Goldberg, E. and Novikov, Y. (2002) "BerkMin: a Fast and Robust SAT Solver", In proceedings of the Conference on Design, Automation and Test in Europe, IEEE Computer Society, pp. 142–149.

(Gray et al., 2007) Gray, J., Tolvanen, J-P., Kelly, S., Gokhale, A., Neema, S., and Sprinkle, J. (2007) "Domain-Specific Modeling", In Fishwick, P. (Ed): CRC Handbook on Dynamic System Modeling, CRC Press, pp. 1-7.

(Gümüs et al., 2007) Gümüs, Ö., Gürcan, Ö., Kardas, G., Ekinci, E.E., and Dikenelli, O. (2007) "Engineering an MAS Platform for Semantic Service Integration based on the SWSA", Lecture Notes in Computer Science, Vol. 4805, pp. 85-94.

(Gürcan et al., 2007) Gürcan, Ö., Kardas, G., Gümüs, Ö., Ekinci, E.E., and Dikenelli, O. (2007) "An MAS Infrastructure for Implementing SWSA based Semantic Services", Lecture Notes in Computer Science, Vol. 4504, pp. 118-131.

(Haag et al., 2003) Haag, S., Cummings, M., McCubbrey, D.J. (2003) "Management Information Systems for the Information Age. 4th edition), McGraw Hill.

(Haesevoets et al., 2010) Haesevoets R., Weyns, D., Torres, M.H.C., Helleboogh, A., Holvoet, T., and Joosen, W. (2010) "A middleware model in alloy for supply chain-wide agent interactions", Lecture Notes in Computer Science, Vol. 6788, pp. 189-204.

(Hahn, 2008) Hahn, C. (2008) "A Domain Specific Modeling Language for Multi-agent Systems", In proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2008), Estoril, Portugal, ACM Press, pp. 233-240.

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

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

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

(Hilaire et al., 2000) Hilaire, V., Koukam, A., Gruer, P., and Muller, J.P. (2000) "Formal Specification and Prototyping of Multi-agent Systems", Lecture Notes in Artificial Intelligence, Vol. 1972, pp. 114-127.

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

(Jackson et al., 2000) Jackson, D., Schechter, I., and Shlyakhter, I. (2000) "Alcoa: the alloy constraint analyzer", In proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Island, pp. 730-733.

(Jackson, 2002) Jackson, D. (2002) "Alloy: A Lightweight Object Modeling Notation", ACM Transactions on Software Engineering and Methodology, Vol. 11, Issue 2, pp.256-290.

(Jackson, 2012) Jackson, D. (2012) "Software Abstractions: Logic, Language, and Analysis (revised edition)", The MIT Press, Cambridge, MA.

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

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

(Klusch et al., 2008) Klusch, M., Nesbigall, S., and Zinnikus, I. (2008) “Model-Driven Semantic Service Matchmaking for Collaborative Business Processes”, In proceedings of the 2nd International Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web, Karlsruhe, Germany, CEUR Workshop Proceedings, Vol. 416, pp. 51-65.

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

(Kumar, 2012) Kumar, S. (2012) “Agent-Based Semantic Web Service Composition”, Springer Briefs in Electrical and Computer Engineering, Springer, 57p.

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

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

(Meseguer, 2000) Meseguer, J. (2000) "Rewriting Logic and Maude: A wide-spectrum semantic framework for object-based distributed systems", IFIP Advances in Information and Communication Technology, Vol. 49, pp. 89-117.

(Moskewicz et al., 2001) Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., and Salik, S. (2001) "Chaff: engineering an efficient SAT solver", In proceedings of the 38th Conference on Design Automation, ACM Press, pp. 530–535.

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

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

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

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

(Padgham and Winikoff, 2004) Padgham, L. and Winikoff, M. (2004) "Developing Intelligent Agent Systems - A practical guide", John Wiley & Sons, 240p.

(Paulraj et al., 2011) Paulraj, D., Swamynathan, S., and Madhaiyan, M. (2011) "Process Model Ontology-based Matchmaking of Semantic Web Services”, International Journal of Cooperative Information Systems, Vol. 20, Issue 4, pp. 357–370.

(Pereira et al., 2008) Pereira, M.J.V., Mernik, M., Cruz, D.d., Henriques, P.R. (2008) "Program Comprehension for Domain-Specific Languages", Computer Science and Information Systems, Vol. 5, Issue 2, pp. 1-17.

(Podorozhny et al., 2007) Podorozhny R., Khurshid S., Perry D., and Zhang X. (2007) "Verification of multi-agent negotiations using the alloy analyzer", In proceedings of the 6th International Conference on Integrated Formal Methods (IFM 2007), Oxford, UK, pp. 501-517.

(Pokahr et al., 2005) Pokahr, A., Braubach, L., Lamersdorf, W. (2005) "Jadex: A BDI Reasoning Engine", Book chapter in Bordini, R.H. et al. (Eds): Multi-Agent Programming, Springer, pp 149-174.

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

(Rougemaille et al., 2008) Rougemaille, S., Migeon, F., Maurel, C., and Gleizes, M-P. (2008) "Model Driven Engineering for Designing Adaptive Multi-Agent Systems", Lecture Notes in Computer Science, Vol. 4995, pp. 318-332.

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

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

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

(Smith, 2000) Smith, G. (2000) "The Object-Z Specification Language", Software Verification Research Centre, University of Queensland.

(Song et al., 2012) Song, Y., Chen, R., and Liu, Y. (2012) "A Non-Standard Approach for the OWL Ontologies Checking and Reasoning", Journal of Computers, Vol. 7, Issue 10, pp. 2454-2461.

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

(Spivey, 1988) Spivey, J.M. (1988) "Understanding Z: A specification language and its formal semantics", Cambridge University Press.

(Spivey, 1992) Spivey, J.M. (1992) "The Z Notation: a Reference Manual (2nd edition)", Prentice Hall.

(Sycara et al., 2003) Sycara, K., Paolucci, M., Ankolekar, A., and Srinivasan, N. (2003) "Automated discovery, interaction and composition of Semantic Web Services", Journal of Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 1, Issue 1, pp. 27-46.

(Taghdiri and Jackson, 2003) Taghdiri M. and Jackson D. (2003) "A Lightweight Formal Analysis of a Multicast Key Management Scheme", Lecture Notes in Computer Science, Vol. 2767, pp 240-256.

(van Deursen et al., 2000) van Deursen, A., Klint, P., and Visser, J. (2000) "Domain-specific languages: an annotated bibliography", ACM SIGPLAN Notices, Vol. 35, Issue 6, pp. 26-36.

(Varga et al., 2004) Varga, L.Z., Hajnal, A., and Werner, Z. (2004) "An Agent Based Approach for Migrating Web Services to Semantic Web Services", Lecture Notes in Computer Science, Vol. 3192, pp. 371-380.

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

(Wang et al., 2006) Wang, H. H., Dong, J. S., Sun, J., and Sun, J. (2006) "Reasoning support for Semantic Web ontology family languages using Alloy", Multiagent and Grid Systems, Vol. 2, Issue 4, pp. 455-471.

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

(Wooldridge and Jennings, 1995) Wooldridge, M. and Jennings, N.R. (1995) "Intelligent Agents: Theory and Practice", The Knowledge Engineering Review, Vol. 10, Issue 2, pp. 115-152.