The Semantics of the Interaction between Agents and Web Services on the Semantic Web

Sinem Getir sinem.getir@ege.edu.tr
Moharram Challenger moharram.challenger@mail.ege.edu.tr
Sebla Demirkol sebla.demirkol@ege.edu.tr
Geylani Kardas geylani.kardas@ege.edu.tr

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

***Abstract*—Development of agent systems is naturally a complex task due to the fundamental characteristics of agents. In addition, agent internals and inter-agent behavior models inside Multi-agent Systems (MAS) may become even more difficult to implement when interactions of agents with web services on the Semantic Web are taken into account. Our approach consists of the utilization of a Domain-specific Modeling Language (DSML) during MAS development in order to cope with the abovementioned challenge. This paper describes how the formal semantics of this DSML can be defined by especially focusing on its viewpoint on agent-semantic service interactions and discusses the use of this semantics definition on MAS validation. Determined semantic rules are both defined and implemented by using Alloy specification language which has a strong description capability based on both relational and first-order logic.**

Keywords-Semantics of Languages; Multi-agent Systems; Semantic Web; Web Services; Alloy; Domain Specific Modeling Languages

I. INTRODUCTION

Software agents can be used to collect Web content from diverse sources, process the information and exchange the results within the Semantic Web [1] environment. Besides, these 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 [2]. Semantic web services can be simply defined as the web services with semantic interface to be discovered and executed [3]. 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 service's semantic protocol definitions.

Development of agent systems is naturally a complex task due to the characteristics of agents such as autonomy, reactivity, proactivity and social ability. In addition, agent internal and inter-agent behavior models inside Multi-agent System (MAS) organizations may become even more complex and hard to implement when interactions of agents with semantic web services are taken into account. We believe that both domain specific modeling and use of a Domain-specific Modeling Language (DSML) [4] can provide the required abstraction and support a more fruitful methodology for the development of MASs especially interacting with the web services on the Semantic Web environment. Within this context, in our previous work, we first defined a metamodel in several viewpoints [5] for a 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 [6].

Although syntax definition based on a metamodel is an essential part of a modeling language, an additional and required part is the determination and implementation of DSML constraints that constitute the semantics which cannot be defined with just a metamodel. Moreover, formal representation of the semantics helps to have an unambiguous definition and precise meaning of a program and to have a possibility for automatic generation of language-based tools [7]. Considering these advantages, defining the formal semantics of a DSML is one of the crucial tasks of a DSML's development. Hence, in this paper, we first define the semantics of SEA_ML's viewpoints which focus especially on the interactions between software agents and web services on the Semantic Web and then we discuss the use of the related semantics definitions on MAS model checking and validation. Defined semantics were implemented by using Alloy language [8] which is based on first order and relational logics.

In the rest of the paper related work is given in section 2. Section 3 includes an overview of SEA_ML's metamodel. Metamodel and semantics of the agent-services interaction are discussed in Sections 4 and 5, respectively. Model validation using Alloy is exemplified in section 6. Finally, section 7 concludes the paper.

II. RELATED WORK

In order to support collaboration of agents and web services, Sycara et al. [3] proposes a capability representation mechanism for semantic web services and discusses how they can be discovered and executed by agents. Likewise, a set of architectural and protocol abstractions that serve as a foundation for agent - web service interactions are introduced in [9]. Based on this architecture, how agents and semantic web services can be integrated are discussed [10] and [11]. Varga et al. [12] proposes another 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. Model-driven engineering of the interaction in question is not considered in abovementioned studies.

The study of [2] presents a methodology based on the well-known Model Driven Architecture (MDA) for modeling and implementing agent and service interactions on the Semantic Web. But neither a DSML approach nor

semantics of service execution is covered in the study. Hahn et al. [13] defines a DSML for agents and provides extensions for this DSML to integrate semantic web service execution into MAS domain. However service interaction model is not built into agent metamodel which differs from the approach presented in this paper. Our study also contributes to studies in [2] and [13] by defining the formal semantics of the agent-semantic web service interaction.

III. Overview of SEA_ML Metamodel

SEA_ML abstract syntax which basically describes concepts and their relationships is provided by SEA_ML metamodel. This metamodel describes how semantic web enabled agents and concepts are linked each other. The Platform Independent Metamodel (PIMM) which represents the abstract syntax of SEA_ML is divided into eight viewpoints to provide clear understanding and efficient use. Detailed discussion of these viewpoints can be found in [5]. To draw a general picture of SEA_ML metamodel while defining its semantics, following brief descriptions are provided:

1. *Agent's Internal Viewpoint:* This viewpoint is related to the internal structures of SemanticWebAgents *(SWA)* and defines required entities and their relations for the construction of agents. It covers both reactive and BDI 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*, *RegistrationRole* are modeled in this viewpoint.
5. *Environmental Viewpoint:* Agents may need to access some resources (e.g., knowledge bases of environment such as *Fact*s and *Service*s). Use of the resources and interaction of agents with their surroundings are covered in this viewpoint.
6. *Plan Viewpoint:* This viewpoint especially deals with Plan's internal structure. *Plan*s are composed of some *Task*s and some atomic elements such as *Action* which provides to connect them to messages.
7. *Ontology Viewpoint: SWA*s know various ontologies as they work with Semantic Web Service (*SWS*) and some concepts which constitute agent's knowledge base such as *Belief* and environmental knowledge base such as *Fact*.
8. *Agent - SWS Interaction Viewpoint:* It is probably the most important viewpoint of the metamodel. Interaction of semantic web agents with *SWS*s is described. In this viewpoint, entities and relations for service discovery, agreement and execution are defined. Also, internal structure of *SWS*s is modeled within this viewpoint.

IV. Agent-SWS Interaction Viewpoint

Agent-SWS Interaction viewpoint 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, internal structure of SWS is modeled inside this viewpoint.

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 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 extension of *SWA*s 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*.

OWL-S [14] as a SWS modeling approach defines a service with three documents: 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 executor dynamics. Finally, Physical Grounding defines the service's executor protocol. Since the operational part of today's semantic services is mostly a web service, Web Service concept is also included in the metamodel associated with the grounding mechanism. These meta-entities are shown in Fig. 1 with *Interface*, *Process* and *Grounding* entities respectively. These components can use *Input*, *Output*, *Precondition* and *Effect* (a.k.a. IOPE) which are extensions of OWL Class from OMG's Ontology Definition Metamodel (ODM). As an example, *Input* can be some information that user request and precondition represent an internet connection. *Output* can be the internet connection result according to the user's request.

V. Semantic Definitions of Agent-SWS Interactions using Alloy

In this paper, we define formal semantics of agent-SWS Interactions with Alloy specification language [8] which also has a useful tool, Alloy analyzer, to check defined model and find validated instance models according to the constraints.

Alloy's logic comprises objects, relations and functions and it also lies on first order predicate and relational logic. Alloy specifications can be briefly described as following: *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

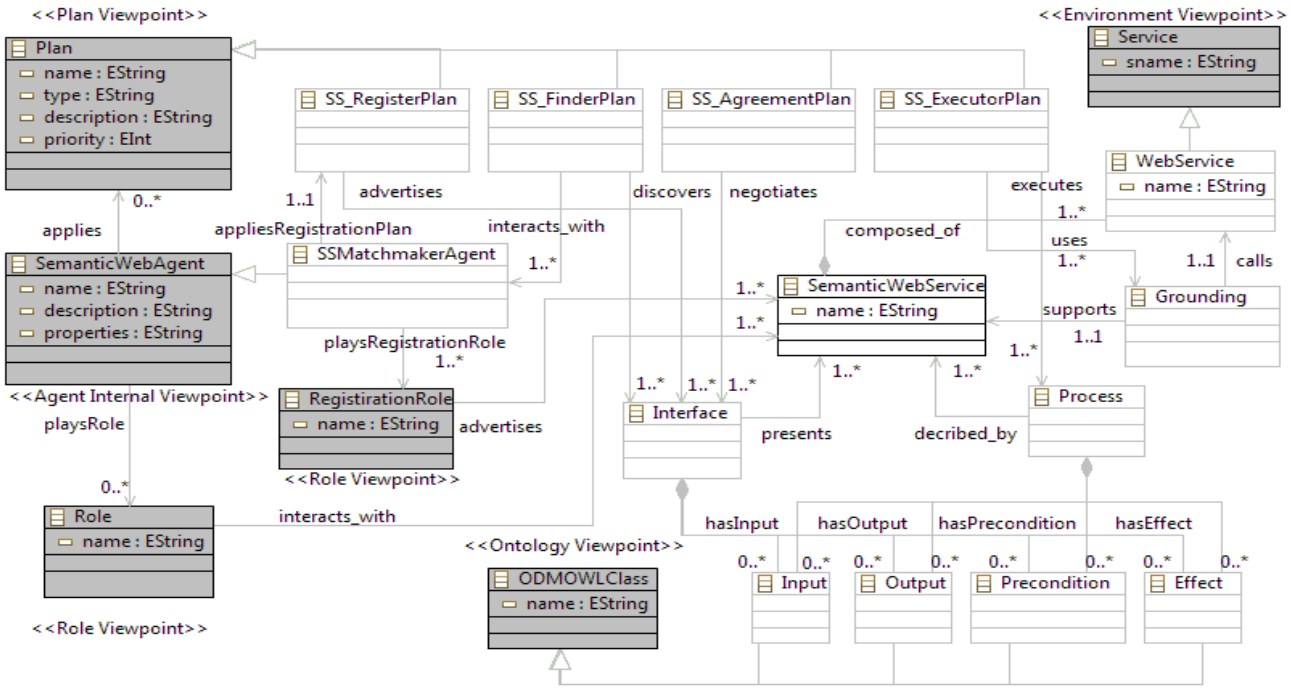

Figure 1. Agent-SWS Interaction Viewpoint

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 counterexamples for assertions.

While defining the rules of the agent-SWS interactions, we represented all meta-model elements with Signatures and added appropriate relations and attributes as Fields in Signatures. Due to space limitation, all abstract syntax definitions are not illustrated in this paper. However an example of Signature definitions is given in Fig. 2 for *SS_MatchmakerAgent*.

First of all, static semantic definitions of abstract syntax in Agent-SWS viewpoint can be done easily with Alloy multiplicity conditions as shown in Fig. 2. Some dynamic Time structures are added to *SS_MatchmakerAgent* and *SS_RegisterPlan* to order their execution in the system. Relationships of these two elements require an order while *SS_MatchmakerAgent* is recording the semantic services (Line 2-3). Hence, time column in the relations obtains us to control them by ordering.

```
01  sig SS_MatchmakerAgent extends SWA{
02    appliesSS_RegisterPlan : SS_RegisterPlan some-> Time,
03    playsRegistrationRole: RegistrationRole some -> Time
04  }
```

Figure 2. A partial representation of abstract syntax elements which carry out the Alloy signature definitions for SS_MatchmakerAgent

Some other static semantics controls for the system are held in Fig. 3. For instance, *SWS*s should be unique in the environment and if they have same name, they should be same *SWS*s. This constraint is bestowed in Fig. 3 with *uniqueSWS* Fact. A *SWS* is composed of some *Web Services* and an *Environment* metaelement, which is included in Environment viewpoint, has some *Services*. On the other hand, *Services* constitutes of various *WebServices* with a composition relationship in the Metamodel. The fact *ServiceComposition* keeps that there should be at least one *Environment* which is related to *Service*s with *has* relationship. This constraint checks whether a *Service*/*WebService* or even a *SWS* exist in the environment. Hence, we can control if there is a *SWS*, there should be a connection to an agent to provide at least one interaction between the agent and SWS.

Since a *WebService* inherits from *Service*, we can add the precondition of implication that tells the environment has some web services with the fact *Agent_SWS_Interaction* in Fig. 3. Post-condition narrates that somehow a *SWA* and SWS should have an interaction. These connections can be possible in two ways. First way is represented with *sws1* which means that a *SWA* plays a *Role* and this *Role* interacts_with *SWS* (Line 14 in Fig. 3). Second way is represented with *sws2* which yields that *SS_FinderPlan* is applied by *SWA*, this plan discovers an *Interface* and this *Interface* presents a *SWS* (Line 15 in Fig. 3). The other ways from *SWA* through other *Plan*s to *SWS* are not added as a connection, because the other plans cannot be applied without an existence of a *SS_FinderPlan.* The value of union of basic connections from *SWA* to *SWS* should be at least one and in this case we can guarantee the *SWS* interactions in the

system. Shortly, if there is a *SWS* in the environment, a *SWA* should interact with it anyway by first discovering its interface and then executing the service.

One of the important dynamic controls for the Agent-SWS Interactions is to provide a proper ordering among Plan types or their relationships, especially ordering the time of access to *SWS* and its elements including *Interface*, *Process* and *Grounding*. In this paper, we present the behavioral and execution semantics of the all connections from an agent to *SWS* and its elements.

```
01  fact uniqueSWS{
02    all n: Name |
03      no disj sws1,sws2: SWS |
04        sws1.name= n && sws2.name = n
05  }
06  fact ServiceComposition{
07      all s:Service | s.~has != none
08  }
09  fact Agent_SWS_Interaction{
10      some ws: WebService, e: Environment|
11      e.has = ws  =>
12      {some swa:SWA, sws1, sws2:SWS, r:Role,
13      t:Time,f:SS_FinderPlan, i:Interface, x:Int  |swa.plays.t= r &&
14      r.interacts_with=sws1 && swa.appliesPlan=f &&
15      f.discovers =i &&    i.presents=sws2 && #sws1 =x &&
16      x.plus[#sws2]    >=1}
17  }
```

Figure 3. An excerpt from the semantics definitions of Agent-SWS interaction

Also, Time column in our semantic definitions not only contributes a dynamic structure to the elements but also gives a facility to order relations for the same element or among the elements. These two features of SEA_ML's semantics are not properly covered in the related work.

To order the Plan types, 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 *PlanStateTransitions fact* in Fig. 4 which explains that next element of *SS_RegisterPlan* can be again a *SS_RegisterPlan* or a *SS_FinderPlan* (lines 2-3). Next element of *SS_FinderPlan* is *SS_AgreementPlan* but in case of the application of *SS_FinderPlan* produces a negative result; this plan can need to be applied again to find another solution. Therefore, a *SS_FinderPlan* can be the next plan of itself as well (lines 4-6) and finally next element of SS_AgreementPlan is SS_ExecutorPlan (lines 7-8).

```
01  fact PlanStateTransitions{
02      { all disj r,r':SS_RegisterPlan| some   f:SS_FinderPlan|
03      aplan/next[r] = r'||  aplan/next[r] = f} &&
04      {all disj f,f':SS_FinderPlan|
05    some    a: SS_AgreementPlan   |
06      aplan/next[f]=f' || aplan/next[f] = a   } &&
07      {all a: SS_AgreementPlan,
08      some e:SS_ExecutorPlan | aplan/next[a]=e}
09  }
```

Figure 4. Intra-Plan Ordering Rules

InterfaceAccessOrdering fact in Fig. 5 orders the access to Interface by Plan Types. More precisely, an *Interface* is firstly advertised by a *SS_RegisterPlan* and assigned its time to t1, secondly it can be discovered by *SS_FinderPlan* and assigned its time to variable t2. Finally, it can be in a negotiation with *SS_AgreementPlan*, and its time value similarly assigned to t3. By using the *Ordering [Time]* in Alloy, we assigned the next event of t1 as t2 and next event of t2 as t3. Hence, *SS_FinderPlan* will try to discover a new service which has been already registered by *SS_RegisterPlan*. Analogously, a *SS_AgreementPlan* should try to negotiate with a *SWS* which has been discovered before (Lines 4-5).

```
01  fact InterfaceAccessOrdering{
02    all r:SS_RegisterPlan,f:SS_FinderPlan,a:SS_AgreementPlan|
03    some t1,t2,t3:Time, i:Interface|    r.advertises[i] = t1  &&
04    f.discovers[i]=t2 &&a.negotiates[i]=t3 &&
05    atime/next[t1] = t2&& atime/next[t2]=t3
06  }
```

Figure 5. Interface access control semantics for other elements

We model the inner relation constraints of each *SS_RegisterPlan* and *SS_FinderPlan* with *Register_FinderPlanExecution fact* in Fig. 6. For *SS_FinderPlan* there is no need for an ordering to interact with a *SS_MatchmakerAgent* or to discover an *Interface* but there is a dependency condition between them which is given between line 2 and line 3. Discovery of SWS by *SS_FinderPlan* depends on an interaction between *SS_MatchmakerAgent* and *SS_FinderPlan.* In the line 8, we use ordering module for Time of elements in *Register_FinderPlanExecution fact* as well. Lines 5-6 extract the times of "SS_MatchmakerAgent applies the SS_RegisterPlan", "SS_RegisterPlan plays a RegistrationRole" and "SS_RegisterPlan advertises Interface" events and assign their times to the t1, t2, t3 time variables respectively. In line 8, we order them such that events pertaining to t1 or t2 should be realized before t3. Precedence between t1 and t2 is not important.

```
01  fact Register_FinderPlanExecution{
02      all f:SS_FinderPlan| #f.discovers >=1 =>
03      #f.interacts_with >= 1
04      all ma:SS_MatchmakerAgent, r:SS_RegisterPlan,
05      role:RegistrationRole, i:Interface | some  t1,t2,t3:Time|
06      r.advertises[i] = t3 && ma.appliesSS_RegisterPlan[r]=t1 &&
07      ma.playsRegistrationRole [role] =t2 &&
08      atime/prev[t3] = t1 || atime/prev[t3]=t2
09  }
10  fact SWSExecutorPlanExecution{
11      all e:SS_ExecutorPlan|some t1,t2:Time,  p:Process,
12      g:Grounding|    #negotiates >= 1 => e.executes[p]=t1&&
13      e.uses[g]=t2 &&atime/next[t1]=t2
14  }
15  fact GroundingExecutorPlanExecution{
16      all g:Grounding, e:SS_ExecutorPlan|some
17      wb:WebService, t1,t2: Time| #executes>=1 =>
18      {e.uses[g]=t1&& g.calls[wb]=t2  &&atime/next[t1]=t2}
19  }
```

Figure 6. Interaction of SWS components and Execution of Plan types

Another inner relationship control of a plan is for *SS_ExecutorPlan*. The *fact SWSExecutorPlanExecution* in Fig. 6 provides to keep the times for the events including,

"SS_ExecutorPlan executes Process" and "SS_ExecutorPlan uses Grounding" and order their times. According to this fact; if there exists a negotiation (Line 12); first, *Grounding* is used, then, *Process* is executed. These events are provided by assigning the times t1 and t2 respectively (12-13) and ordering them in the sequence t1<t2 (Line 13).

Finally, for the accomplishment of Plan executions, *GroundingExecutorPlanExecution fact* in Fig. 6 is provided. If the *SS_ExecutorPlan* has already been applied (Line 17), *Grounding* firstly should be used by *SS_ExecutorPlan* and secondly, *WebService* should be called by Grounding with the times t1 and t2 which indicates the order t1<t2 (Line 18).

VI. Model Validation with Alloy Analyzer

Development with Alloy specification language is simultaneously done with a fully automated analyzer tool which visualizes and checks the models, and also generates instances. Each analysis in tool works through the aim of solving a constraint that either produces a counterexample or produces an instance.

As the first task of analyzer, assertion checking generates counterexamples that counterproofs the properties by searching a binding of negative formula of the given assertion formula. We provided model checking with particular assertions and according to these assertions; the analyzer did not give counterexamples. Hence, within these checking results, we validated our model according to the assertions.

For example, *PlanStateTransitions* constraint in Fig. 4 makes the plans to process in an order. *StateDiagramProperty* in Fig. 8 is checked until scope size 10 against the rule that breaks this order. Within this property, there is no ordering that *SS_FinderPlan* comes after *SS_AgreementPlan* or *SS_ExecutorPlan* after *SS_AgreementPlan*.

Another property is *EnvironmentProperty in* Fig. 8 tells that there is no *WebService* which does not belong to any *Environment*. This assertion also is valid that the analyzer could not find any counterexample which is executed with *check* command until scope size 10 by courtesy of the fact *ServiceComposition* in Fig. 3.

The final assertion we present in this paper is following *InterfaceProperty* in Fig. 8 which claims that the relations towards *Interface* should be kept in the order: First "SS_RegisterPlan advertises an Interface; second, SS_FinderPlan discovers it and then, SS_AgreementPlan negotiates with it." A similar assertion is added to model to check the access of *SWS*, *Process* and *Grounding*.

As the second task of analyzer, running the predicates generates instance models in a visual or textual manner. It generates the instance model without need of any input from user. We create some predicates and take some results that are executed with *run* command and found by analyzer consistent with the model. In Fig. 7, one of the results is represented visually and textually.

Within Show predicate in Fig. 8, we assume a system

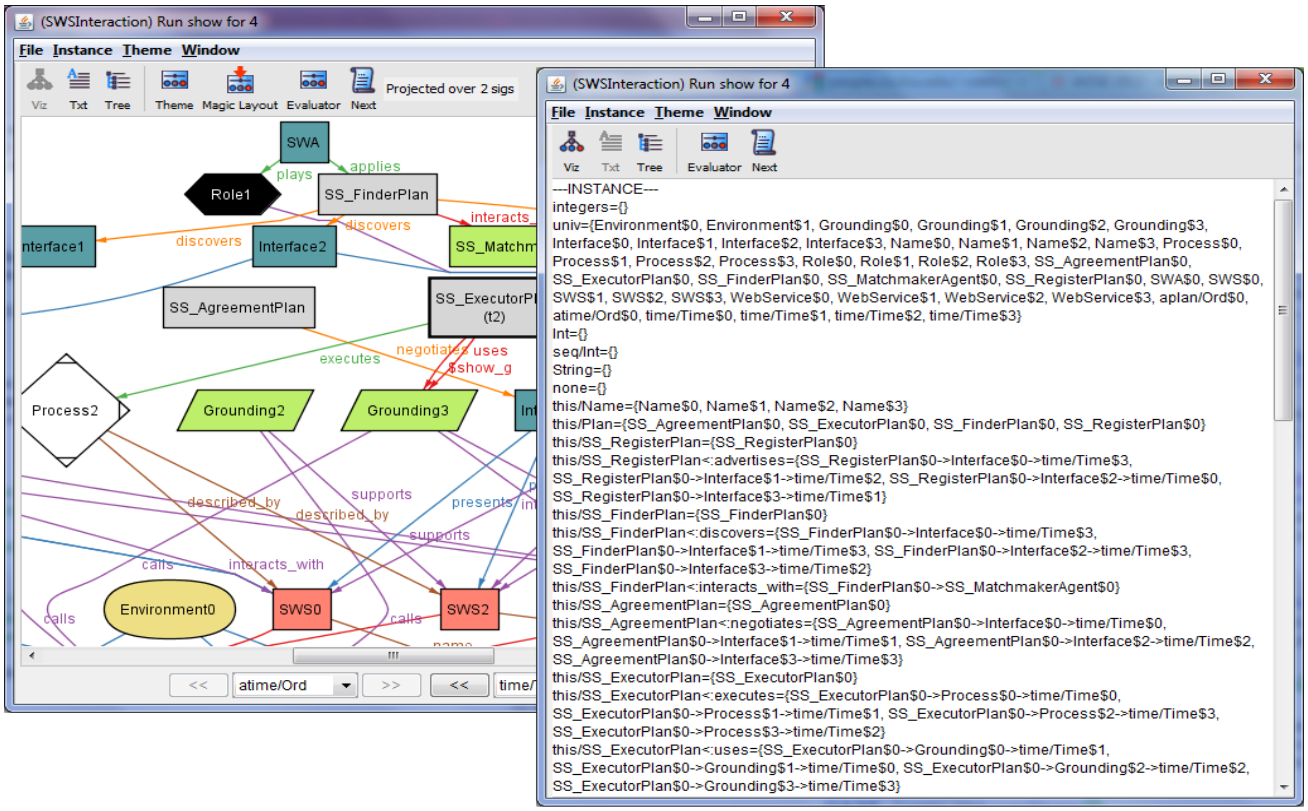

Figure 7. An instance model of Agent-SWS Interactions Alloy model using Alloy Analyzer

which has some *SS_RegisterPlan*s, one *SS_FinderPlan*, one *SS_AgreementPlan*, *SS_ExecutorPlan* and a *SS_MatchmakerAgent*. As an example, we keep the number of *SS_RegisterPlan*s as two. In case the scope size is 3, analyzer cannot find a consistent model. Therefore, the scope of the elements is kept as minimum such that 4 for simplicity. Nevertheless, according to the semantics of model, it creates the required concepts in respect to loyalty of constraints.

```
01  assert StateDiagramProperty{
02    no f:SS_FinderPlan, e:SS_ExecutorPlan,
03    a:SS_AgreementPlan| aplan/next[e]=a && aplan/next[a]=f
04  }
05  assert EnvironmentProperty{no wb:WebService| #wb.~has=0 }
06  assert InterfaceProperty{
07    some r: SS_RegisterPlan, f:SS_FinderPlan,
08    a:SS_AgreementPlan|no i:Interface|
09    atime/prev[r.advertises[i]]=f.discovers[i]||
10    atime/prev[f.discovers[i]]=a.negotiates[i]
11  }
12  pred show{
13    one SS_FinderPlan&& some SS_RegisterPlan&&
14    one SS_AgreementPlan &&  one SS_ExecutorPlan&&
15    one SS_MatchmakerAgent&&  some i:Interface,
16         t:Time|#i.~(advertises.t>=2
17  }
```

Figure 8. Use of Assertions and Predicates for the analysis of the model

The plan ordering and dependency of each other among the plan types which is defined as a fact in the section 5 is held in the instance model. In textual model, the ordering is given with ordering module and represented with the terms such as first and next. The *WebService*s can only exist inside an *Environment*. Also, *SWS* access is controlled and Grounding execution is done in an ordering as can be seen in the instance model in Fig. 2.

As a result, in the orientation of Alloy analyzer, the tool is useful to make the models not only correct but also more concise and appropriate. Given assertions orientates the user to correct the model by generating counterexamples against to the given assumption. Also, the scope of run command can be tested to find validated instance models.

VII. CONCLUSION

In this paper, formal semantics of the Agent-SWS Interaction based on its metamodel are discussed in both static and dynamic aspects. The definitions for this semantics are given with Alloy specifications and model validation is provided with Alloy Analyzer. During the analysis of the model, the scope size can be increased until finding an instance or counter example. However it can take long time as the scope size increases. Still we experienced that it is quite valuable if we can show validation of the model for a possible scope size. We believe that the defined formal semantics in this study can guide to the developers during the design of the interactions between agents and web services on the semantic web environment before their exact implementation.

As the future work, we first plan to integrate the semantic checking controls introduced in this paper into our existing tool [6] which is used for the model driven development of semantic web enabled MAS. Secondly, additional dynamic controls, such as interaction sequence constraints between agents, and knowledgebase controls will be provided to prevent conflicts within the Environment facts.

ACKNOWLEDGMENT

This study is funded by The Scientific and Technological Research Council of Turkey (TUBITAK) Electric, Electronic under grant 109E125.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

[12] L. Z. Varga, A. Hajnal, Z. Werner, "An Agent Based Approach for Migrating Web Services to Semantic Web Services". Lecture Notes in Artificial Intelligence, 3192, pp. 371-380, 2004.

[13] C. Hahn, S. Nesbigall, S. Warwas, I. Zinnikus, K. Fischer and M. Klusch, "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, 2008.

[14] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Sycara, "OWL-S: Semantic Markup for Web Services", W3C Member Submission, 2004.