Declarative specifications for the development of multi-agent systems

Declarative specifications for the Development of Multi-agent Systems

Moharram Challenger[1,a], Marjan Mernik[2,b], Geylani Kardas[1,c], Tomaž Kosar[2,d],

[a]moharram.challenger@mail.ege.edu.tr, [b]marjan.mernik@uni-mb.si, [c]geylani.kardas@ege.edu.tr, [d]tomaz.kosar@uni-mb.si,

[1]International Computer Institute, Ege University, Izmir, Turkey.
[2]Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia.

Abstract

The designing and implementation of a Multi-agent System (MAS), where autonomous agents collaborate with other agents for solving problems, constitute complex tasks that may become even harder when agents work in new interactive environments such as the Semantic Web. In order to deal with the complexities of designing and implementing a MAS, a Domain-Specific Language (DSL) can be employed inside the MAS's development cycle. In such a manner, a MAS can be completely specified by programs written in a DSL. Such programs are declarative, expressive, and at the right abstraction level. In this way the complexity of MAS development is then partially shifted to DSL development and the task herein can be much more feasible by using a proper DSL development methodology and related tools. This paper presents and discusses our methodology for DSL development based on declarative formal specifications that are easy to compose, and its usage during MAS development. A practical case-study is also provided covering an example of a MAS's development for expert finding systems. By using denotational semantics for precisely defining the language, we show that it is possible to generate the language automatically. In addition, using attribute grammars makes it possible to have modular methodology within which evolutionary language development becomes easier.

Keywords: Domain-specific Language, Multi-agent System, Semantic Web, Formal Semantics, Declarative Specifications

1. Introduction

A software agent is an encapsulated software system situated within a certain environment, and is capable of flexible autonomous action within this environment in order to meet its design objectives [1]. These autonomous, reactive, and proactive agents can also behave in a cooperative manner and collaborate with other agents for solving common problems. In this way, these intelligent agents constitute systems called Multiagent Systems (MASs).

The design and implementations of MASs are complex tasks when considering their dynamicity and autonomous characteristics. Their behavior is more complex when taking into account the agent's interaction with new agent environments such as the Semantic Web [2, 3].

The Semantic Web extends the current web in such a way that the web pages' contents can be interpreted using ontologies [2] that help machines to understand web-content. Software agents can fulfill the interpretations in question by handling the semantic contents on behalf of their human users by collecting web contents from diverse sources, processing the information, and exchanging the results.

In addition, autonomous agents can evaluate semantic data and collaborate with semantically-defined entities of the Semantic Web, such as semantic web services (SWS), by using content languages [4]. SWSs can be simply defined as the web services with semantic interfaces to be discovered and

executed [5]. In order to support the semantic interoperabilities and automatic compositions of web services, the capabilities of web services are defined in service ontologies that provide the required semantic interfaces. Such interfaces of the SWSs can be discovered by software agents and these agents may then interact with these services to complete their tasks. The engagements and invocations of a semantic web service are also performed according to the service's semantic protocol definitions. For instance, the dynamic composition of heterogeneous services for the optimal selection of service providers can be achieved by employing agents and ontologies, as proposed in [6].

However, agent interactions with semantic web services add further complexities when designing and implementing MASs. Different methodologies can be applied in order to deal with this complexity. One of the possible alternatives is domain-specific languages (DSLs) [7-10] that have notations and constructs tailored towards a particular application domain (e.g. MAS and Semantic Web). In this way, DSLs raise the abstraction level, expressiveness, and ease of use. In other words, when using this methodology the users focus on the program model of the solution instead of a platform code for the solution, and they can automatically generate code from the model using DSL tools. As a result, the DSLs' users mostly need the knowledge from the problem domain [11] but little programming experience.

We are convinced that the use of a DSL can provide the required abstraction and support a more fruitful methodology for the development of MASs especially when working within a Semantic Web environment. Within this context, prior to the work discussed here, we first provided a metamodel defined in several viewpoints [12] for MASs working within a Semantic Web environment. Then, based on this metamodel, we developed a DSL called the Sematic web-Enabled Agent Language (SEA_L) [13, 14] including an interpreter mechanism for SEA_L which is defined for enabling code generation regarding the implementation of SEA_L agents (e.g. in JADEX platform [15]).

Although syntax definition based on a metamodel is an essential part of a modelling language, an additional and required part is the determination and implementation of DSL constraints that constitute those semantics that cannot be defined solely by a metamodel or syntax. In line with these constraints, the semantics of a DSL include some rules that restrict the instance models created according to the language.

The definition of the semantic rules was provided towards this end in our previous study [16, 17] focusing on Agent-Semantic Web Service interaction. However, the represented semantic rules were not implemented and applied to SEA_L language (although, it is one of the crucial tasks during a DSL's development). In other words, the defined formal semantic rules have not been applied to any translational/operational semantics and have no effects on the quality of the generated code. Such an application is important as the execution of the language is described directly or by translating to another language in translational/operational semantics.

Therefore, in this study, the specification of the static semantics regarding the interactions between software agents and semantic web services in SEA_L language is formally declared using denotational semantics [18] which paves the way for the implementation of these specifications using attribute grammars [19, 20]. Static semantics is used to control some constraints during system modelling in the DSL. On the other hand, denotational semantics interpret the phrases of a language as mathematical denotations and conceptual meanings that can be thought of abstractly.

Our main motivation is that declarative specifications can present the meanings of both associations and constraints for the language in a formal way. The formal representation of the semantics helps to provide an unambiguous definition and precise meaning of a program. It also helps to have the possibility for more accurate code generation by the language-based tools [21], when they are

implemented in the DSL (which is lacking in the related work). A successful system verification and validation can also be achieved with a proper formal semantics definition.

In order to implement the defined declarative semantics of SEA_L, we have employed the LISA tool [22] which is based on attribute grammars. As can be noticed in further sections of this paper, implementation using LISA enables the separation of concerns between the syntax and the semantics. Moreover, it offers tools for syntax and semantics evaluation separately.

Hence, the main contributions of this paper are listed as follows:

- a new DSL-based methodology for the development of MAS with Semantic Web, where DSLs are formally specified with formalism that enables incremental specifications; and
- the implementation of the static semantics of SEA_L MAS development language and integrating them within the operational semantics of SEA_L in the form of declarative specifications using attribute grammars; this leads to productivity and advanced semantic controls in our DSL, and modularity and extendibility for the new language.

The remainder of the paper is organised as follows: Section 2 discusses the related work. The applied approach is elaborated in Section 3. The proposed methodology is discussed in Section 4 by especially taking into consideration the Agent-SWS interaction viewpoint of the system. In Section 5, use of the language and its semantics is presented using a case-study. Section 6 covers the evaluation and discussion for this study. Finally, the paper is concluded in Section 7.

2. Related Work

The design and implementation of MAS working on the Semantic Web keep their emphasis since the first introduction of this new generation web in [2]. Berners-Lee et al. [2] took software agents as the central point of distributed content collection, knowledge formalisation, processing and interpretation of data, which are all required for the realisation of such a web environment. Hence MAS composed of many autonomous agents now became one of the major components of the Semantic Web. More specifically, the integration of agents and the knowledge ontologies steers the use of web services [23] and enables the automatic discovery and execution of services by the agents within the Semantic Web environment. Many researchers have investigated how the agents can participate in service execution inside the Semantic Web environment and have provided noteworthy methodologies and/or protocols for collaboration between agents and other Semantic Web entities, e.g. semantic web services. For instance, Paolucci et al. [24], Sycara et al. [5] and Li and Horrocks [25] proposed various capability representation mechanisms for semantic web services and discussed how they can be discovered and executed by agents. Agents infer about the suitability of the advertised semantic web services for the required action according to those defined service representations and decide on executing the more appropriate service. The studies in [26] and [27] described agent environments which use OWL-S ontologies to advertise descriptions of agent services to transport them using communication messages. Those descriptions provided for the use of agent services as if they were semantic web services. OWL-S [28] is an ontology built on top of Web Ontology Language (OWL) for describing Semantic Web Services and enables users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints. In addition, OWL [29] is a family of knowledge representation languages for authoring ontologies which are a formal way of describing taxonomies and classification networks.

A set of architectural and protocol abstractions that serves as a foundation for agent - web service interactions on the Semantic Web was introduced in [30]. This initiative architecture addressed the

requirements of dynamic service discovery, service engagement, service process enactment and management, community support and quality of service for the Semantic Web. The architecture is based on the MAS infrastructure because the specified requirements can be accomplished with asynchronous interactions and using goal-oriented software agents. Two implementations of this conceptual architecture were provided in [31] and [32] which mainly considered the service matchmaking, service discovery and service execution functionalities of software agents. Preparation of existing web services into the Semantic Web environment or migrating them via software agents is also possible by following approaches such as given in [33] and [31].

Similar to [5] and [25], Talantikite et al. [34] provided an input-output similarity measure for OWL-S ontologies for determining the best semantic services and compose them for meeting client requests. Instead of semantic web service profiles, the use of OWL-S process models is proposed during the service discovery in [35]. Hence, it is aimed at finding and matching more relevant services with the proposed algorithm. Kumar [36] discussed multi-attribute negotiation between the agents working on the Semantic Web and the benefits of such negotiation on both the selecting and composing of semantic web services. Our study contributes to the above-mentioned MAS and the Semantic Web research by providing a DSL which can be used to formally define and implement the interaction between software agents and semantic web services. MAS developers can use the specifications given in this paper to realise concrete implementations of collaboration protocols and abstract Semantic Web service architectures (e.g. [30] and [36]) discussed above; such that agents may interpret and reason with semantic descriptions during the deployment of semantic web services.

On the other hand, Agent-oriented Software Engineering (AOSE) is one of the major areas of agent research intersecting Software Engineering. Different researchers have applied model-driven techniques based on DSL and DSML (Domain-Specific Modeling Language), as a development methodology for MASs. In the remainder of this section, we discuss these studies and compare them with our study.

There are several studies that only describe the main elements and the relationships between them as a metamodel for MASs. Some of these studies have presented general-purpose metamodels e.g. AALAADIN [37], ACSM [38], PIM4Agents [39], and FAML [40], yet some others have presented metamodels for specific methodologies, e.g. [41] for ADELFE [42], Gaia [43], and PASSI [44]. Another example is the metamodel presented in [45] for SODA [46] agent development methodology. In fact, these studies present some conceptual class diagrams for MASs and are not considered as modeling languages.

Therefore, AOSE researchers have made great strides into developing MAS modelling languages in addition to MAS metamodelling studies. For instance, Depke et al. [47] introduced an agent-oriented modelling technique based on the UML notation. On the other hand, Agent UML (AUML) [48] is perhaps the better-known modelling language within the agent community, which presents new agent-based extensions to package and template structures and sequence, interaction, activity and class diagrams of UML. AUML relies too much on UML which is proposed for object-oriented system specification and also, as stated in [49], AUML's visual notation is incomplete and does not provide a textual notation for exchanging with other developers. The Agent Modeling Language (AML) [50] is another general modelling language for MASs that is based on the UML 2.0 superstructure. Although it provides a visual modelling language, it lacks textual notation and formal semantics. Another metamodel-driven agent modelling language is introduced in [51] which aims at expressing models for various AOSE methodologies and specifically enables both comprehension and interchange of MAS models amongst agent developers.

In order to present more powerful methodologies for MAS developers, some researchers have provided DSLs or DSMLs for MASs. For example in [52] and [53] the authors presented MAS metamodels and related tools and called them DSL and DSML respectively. However, they lack the concrete syntax and semantics that are required for a complete DSL. In [54], the authors defined and implemented a formalised syntax and semantics of DSL for mobile agents including a textual editor, and code generation. However, the introduced DSL considers the mobile agent domain that completely differs from the specific domain of SEA_L. Another DSML was provided in [55] for developing MASs using Prometheus methodology [56]. The abstract and concrete syntax, the tool, and the code generation for the JACK agent platform [57] were provided. A similar study was discussed in [58] that proposed a technique for defining agent-oriented engineering process models that can be used to define processes for creating both hardware and software agents. This study also offered a related model-driven development (MDD) [59] tool using Software & System Process Metamodel (SPEM) [60] and based on INGENIAS methodology [61] for MAS development. Nevertheless, both of the studies are bound to specific methodologies and their metamodels are not general enough for MASs. Also, neither [55] nor [58] covered software agents in the Semantic-Web.

In [62] a new DSML for MAS, called DSML4MAS, was introduced. The abstract syntax of DSML4MAS was provided from a platform-independent metamodel (PIMM) that contained several viewpoints of a MAS. The abstract syntax of a model is its structure described as a data type, independent of any particular representation or encoding. In addition, the concrete syntax provides a mapping between meta-elements and their representations for models and includes the set of notations which facilitates the presentation and construction of the language. The concrete syntax of DSML4MAS was provided by means of appropriate graphical notations for the concepts and relationships [63]. However, it supported neither the agents on the Semantic Web nor the interaction of Semantic Web-enabled agents with other environment members such as semantic web services. Our study contributed to the afore-mentioned efforts by also specialising in the Semantic Web support of MASs. In [64], the authors introduced their approach on integrating agents with semantic web services. In addition to the MAS metamodel described in [62], a new platform-independent metamodel was proposed for semantic web services. A relationship between these two metamodels was established in such a way that the MAS metamodel was extended with new meta-entities in order to support semantic web services' interoperabilities, and it also inherited some meta-entities from the metamodel proposed for semantic web services. Instead of using two separate metamodels, our DSL has a built-in support for the modelling of agent and semantic web services' interactions by including a special viewpoint. Finally, the work in [4] presented a methodology based on OMG's well-known Model Driven Architecture (MDA) [65] for modelling and implementing agent and service interactions on the Semantic Web. However, neither a DSL approach nor the semantics of service execution were covered in the study.

Semantics control and its formal representation are key factors in the success of DSLs and DSMLs. A few studies have considered this aspect of MDD on MAS development. For instance, the study in [66] uses the Object-Z language [67] for defining the formal semantics of DSML4MAS [62]. In this way, the system designer is supported when validating and verifying the generated design. In [68], the authors presented a framework for supporting the formal specifications and verifications of DIMA multi-agent models using Maude language [69] based on rewriting the logic. This presented work only covers DIMA MAS models and does not have a generic MAS perspective such as SEA_L's. In [70], the authors believed that Object-Z and statecharts individually were not powerful enough for specifying the complex MASs and hence they combined Object-Z and statecharts in order to define MASs based on an organisational model. AgentZ [71] extended Object-Z for specifying MASs by adding new constructs for improving its structure, namely agent-oriented entities such as agents,

organisations, roles, and environments. Our work differs from these studies by specialising in the Semantic Web support in MASs.

Validation of the designed agent systems by applying formal methods can also be critical during MAS development. Related worthwhile approaches are extensively discussed in [72] and [73]. Considering the use of Alloy [74, 75] in MAS development, Podorozhny et al. [76] presented an approach for designing a robust MAS and checking the properties of coordination, interaction, and agents' data structures, using the Alloy analyzer. Additionally, Haesevoets et al. [77] formally defined the relationships between the interactions, the exposed information and provided policies and laws for an agent's middleware by using Alloy. In this way, they guaranteed a number of properties that are important in the usage of this middleware. Any kind of fully fledged DSL or DSML is not provided in these studies.

By considering our previous studies, in [78], we have shown how domain-specific engineering can provide the easy and rapid construction of a Semantic Web-enabled MASs. Ideas have been discussed for abstract syntax, concrete syntax, and formal semantics. Furthermore, a metamodel, which in fact constitutes the preliminary version of the abstract syntax of SEA_L, was introduced in [12]. Also, textual [14] and graphical [79] languages were presented that can be used during both the modelling of Semantic Web-enabled MASs and the syntactic checking of designed models. In addition, the formal semantics of the Agent-SWS interaction viewpoint regarding our DSL is introduced in [16] and [17]. Based on these building blocks, in this paper we discuss the representation of SEA_L's formal semantics in the form of denotational semantics and implementing them by transforming them to attribute grammars using LISA. In this way, the translational and denotational semantics come together in a modular way, which is lacking in the literature. The usefulness of attribute grammars as high-level declarative specifications for specifying various XML processing tasks have also been shown in [80]. Our approach for combining declarative attribute grammar specifications is more powerful than the approach presented in [80], where a combination of non-conflicting attribute grammar fragments is only possible. In other words, no conflicts amongst the combined specifications are allowed in [80], while our approach enables language extensions, language unifications, self-extension, and extension compositions of attribute grammar fragments [81].

3. DSL Development Methodology

A DSL life-cycle was described in [8, 82]. It is comprised of the following phases: decision, domain analysis, DSL design, DSL implementation, DSL testing, DSL deployment, and DSL maintenance. During the decision phase several criteria need to be evaluated and contrasted to find out whether the development of a new DSL is a solution to our problem. In this respect decision patterns [8] might be helpful as they indicate those situations in the past where the introduction of a DSL into a process has been successful. If the decision about implementing a DSL was positive during the initial phase then the next stage is a DSL development, which is comprised of the following phases: domain analysis, DSL design, and DSL implementation. These phases are crucial during a DSL life cycle and appropriate methodology is needed to do it correctly. Many DSLs have been developed from scratch by informally performing a particular phase (domain analysis, DSL design, DSL implementation), certain parts of a phase (e.g., semantic part of a DSL design), or even all the phases. There are several problems with the 'from scratch' approach. The more notable problems are: that often unsatisfactory DSL is developed and several costly re-development iterations are needed, difficult maintenance, and that DSL evolution is hard. For example, often problems that should have been identified during early phases only become visible during later phases. Hence, such an informal approach to DSL development is not recommended. A formal approach to DSL development has already been proposed

in Mernik et al. [8] and in this section a particular formal DSL development methodology is described. Namely, domain analysis, DSL design and DSL implementation are not narrow processes and various formalisms can be applied.

The task of domain analysis is to select and define the domain of focus, collect appropriate domain information, and integrate them into a coherent domain model that represents concepts within a domain and relationships within the domain concepts. Here several existing domain analysis methodologies can be used, such as: DARE – Domain Analysis and Reuse Environment [83], FAST – Family-Oriented Abstractions, Specification, and Translation [84], FODA – Feature-Oriented Domain Analysis [85], and ODE – Ontology-based Domain Engineering [86]. In our approach we have used FODA, as the common and variable properties of a domain are easy to identify in feature diagrams (i.e., variation points). In fact, the list of variations indicates precisely which information is required for specifying an instance within a system. This information must be directly specified within programs written in a DSL or be derived at from them. On the other hand, the commonalities are used for defining the execution model (through a set of common operations) and the primitives of the language. The outputs from domain analysis are: terminology, concepts, commonalities, and variations. These are easily identified from FODA feature diagrams [87] and should be used as inputs into the next phase – DSL design.

Designing a language involves defining the constructs within the language (syntax) and giving semantics to the language. Both sub-phases, syntax and semantics, can be managed informally or formally. The advantages of formal syntax and semantic specification of programming languages are well-known: the structure and meaning of a program is precisely and unambiguously defined, and it offers a unique possibility for the automatic generation of compilers or interpreters. Those programming languages that have been designed using one of the various formal methods for syntax and semantic definitions have better syntax and semantics, less exceptions, and are easier to learn. Moreover, researchers have recognised the possibility that many other language-based tools could be generated from formal language specifications. Therefore, many language implementation systems not only automatically generate a compiler/interpreter but also complete language-based environments including editors, type checkers, debuggers, various analysers, and animators [88]. The following formal methods for DSL syntax definition have been used: BNF, FDL [89], metamodels, DTD, and XML Schema. The powers of all these formal methods for syntax definition are the same. Hence, transformations between different syntax descriptions are more or less easy to achieve. In our DSL development methodology we have opted for metamodels because class diagrams are often familiar to programmers using object-oriented languages, and relationships between class diagram entities can also be named. We can define a metamodel as a model of a model, providing the frameworks, rules, and constraints for defining the models for modelling a predefined class of problems. Many language implementation systems (i.e., compiler generators) use variants of Backus–Naur Form (BNF) and additional transformation is needed from metamodels to BNF. BNF is a widely-known notation technique for context-free grammars, which is used to describe the syntax of a computer programming language. It provides an exact description for specifying the syntax of formal languages [90]. There are some compiler generators (e.g., YAJCo [91], Xtext [92]) where syntax definition is given with a metamodel. Transformation between BNF and metamodels was described in [93, 94]. As semantic formalisms are usually based on abstract syntax instead of concrete syntax, both forms need to be developed as concrete syntax is needed later when parsing. While different syntax formalisms are equivalent, the situation is quite different for the semantics, where approaches such as attribute grammars, axiomatic semantics, operational semantics, denotational semantics, and translational semantics are complementary, and used by different stakeholders. For example, attribute grammars are used by compiler writers, while axiomatic and denotational semantics are used by language designers

to prove various language properties without concentrating on particular implementation. On the other hand, operational semantics define the meaning of the language through configuration changes and are closer to the implementation on virtual machines. Another distinction amongst different semantic formalisms is whether they are able to describe the static and/or dynamic semantics of a language. In our DSL development methodology we have used a mixture of different semantic formalisms. For describing static and dynamic semantics, we have used denotational semantics, and for code generation the translational semantics.

Finally, after DSL has been designed, it is time for its implementation. Different approaches for DSL development have been introduced in [8]: interpreter, compiler/application generator, embedding, preprocessing, extensible compiler/interpreter, Commercial Off-The-Shelf (COTS), and the hybrid approach. Clearly we want to select an approach that requires the least effort during implementation and offers the greatest efficacy to the end-user [95]. In our approach to DSL development, the formal specifications during the design phase constitute an important part. Of course, it is harder to design a DSL formally than informally. This pays off during the DSL implementation phase, where a complete compiler/interpreter can be automatically generated. This is achieved in our case by mapping denotational and translational semantics to the language implementation system LISA [22], which is based on attribute grammars. LISA is a compiler generator and tool with the following features:

- It offers an integrated development environment where users can specify-generate-compile-execute programs in a newly specified language.
- Lexical, syntax and semantic analysers can be of different types and can operate standalone;
- It provides visual presentations of different structures, such as finite state automata, BNF, syntax tree, semantic tree, and dependency graphs.
- Animation is possible for lexical, syntax and semantic analyzers.
- The specification language supports multiple attribute grammar inheritance, which enables regular definitions, grammar production rules, attributes, semantic rules and operations on semantic domains to be inherited, specialised or overridden from ancestor specifications.

Code generation using translational semantics is easy to implement in attribute grammars, while transformation from denotational semantics into attribute grammars is more complicated but similar to the translation scheme from natural semantics into attribute grammars [96]. The former transformation was described in [97].

The whole process of our methodology for DSL development is presented in Figure 1, where it can be seen that the outputs from syntax and semantic definitions are used as inputs into our language implementation system LISA. Section 4 shows how our DSL development methodology has been used for the development of the Agent-SWS Interaction Language.

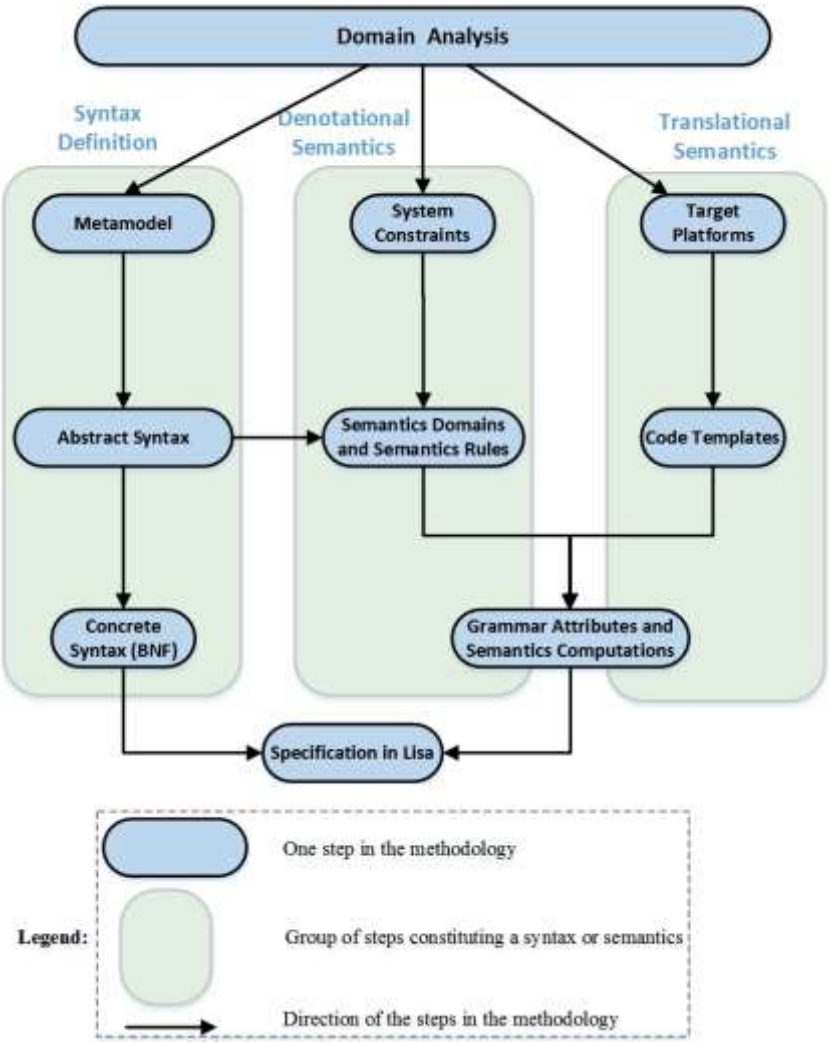

Figure 1: Overall DSL development methodology

4. Implementation of the Agent-SWS Interaction Language

Our approach to DSL development is clearly shown for the Agent-SWS Interaction Language, which is a small sub-language of the Semantic web-Enabled Agent Language (SEA_L) [14]. SEA_L is a DSL for specifying agents interacting with each other within a MAS and with semantically-defined entities of the Semantic Web (e.g., Semantic Web Services). The main goal of this language is to provide a convenient environment for agent developers to construct and implement software agent systems working on various application domains. In order to support MAS experts when programming their own systems, and to be able to fine-tune them, SEA_L covers all aspects of an agent system from the internal view of a single agent to the complex MAS organisation. In addition to these capabilities, SEA_L also supports the model-driven design and implementation of autonomous agents who can evaluate semantic data and collaborate with semantically-defined entities of the Semantic Web, such as Semantic Web Services (SWS). That feature exactly differentiates SEA_L and makes it unique regarding any other MAS DSL currently available. Within this context, it includes new viewpoints which specifically pave the way for the development of software agents working on the Semantic Web environment. Modelling agents, agent knowledge-bases, platform ontologies, semantic web services, and interactions between agents and SWS are all possible in SEA_L.

The whole SEA_L is split into several sub-languages or viewpoints [12], each of which represents a different aspect for developing Semantic Web enabled MASs: Agent's Internal Viewpoint is related to the internal structures of semantic web agents (SWAs) and defines entities and their relationships required for the construction of agents. It covers both reactive and Belief-Desire-Intention (BDI) [98] agent architectures. Interaction Viewpoint expresses the interactions and communications in a MAS by taking messages and message sequences into account. MAS Viewpoint solely deals with the construction of a MAS as a whole. It includes the main blocks that compose the complex system as an organisation. Role Viewpoint delves into the complex controlling structure of the agents and addresses role types. Environmental Viewpoint addresses the use of resources and interactions between agents within their surroundings. Plan Viewpoint deals with the internal structure of an agent's plan, which is composed of Tasks and atomic elements such as Actions. Ontology Viewpoint addresses ontological concepts that constitute an agent's knowledge-base (such as belief and fact). Agent - SWS Interaction Viewpoint defines the interactions of agents with semantic web services (SWS) [5] including the definition of entities and relationships for service discovery, agreement and execution. The relationships between these viewpoints for building the whole metamodel are depicted in Figure 2. The last viewpoint and definitely the more important sub-language, Agent-SWS Interaction Language, is mainly discussed in this paper.

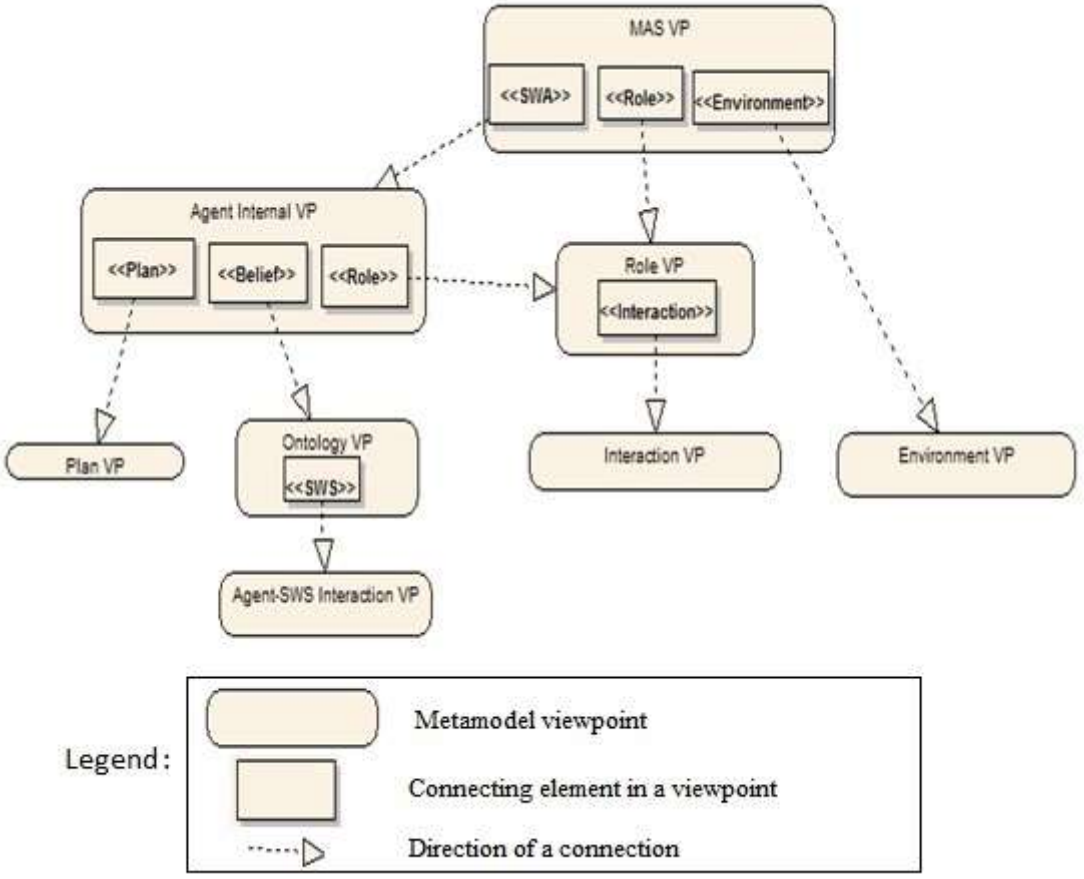

Figure 2: The relationships between different viewpoints of the metamodel

The first step in our DSL development methodology (Section 3) is the domain analysis where a domain and its concepts and relationships amongst them are identified using FODA methodology [85]. A domain is an area of knowledge or activity characterised by a set of concepts and terminology

understood by practitioners within that area. A domain model in FODA is captured using a feature diagram. It is obtained after an extensive study of the domain under discussion by studying existing literature, existing implementations, and discussions with domain experts. In the past we have been extensively involved in the development of MASs and Semantic Web Services (e.g. [4, 78, 79]). Hence identifying concepts and their relationships within this domain has been fairly easy. In Figure 3, a feature diagram for Agent-SWS Interaction is presented, where it can be seen that the main concepts are: SemanticWebAgent (SWA), SemanticWebService (SWS), and SemanticMatchmakerAgent (SMA). A SWA, which interacts with SWSs, might play different Roles and have various Plans (e.g., registration, finding, agreement, and execution plans) for achieving its goals. These agents are planned to collect Web content from diverse sources, process the information and exchange the results on behalf of their human users. SWA can also evaluate semantic data within these MASs and collaborate with semantically defined entities such as SWSs by using content languages. A SWS represents any service (except agent services) the capabilities and interactions of which are semantically described. SWS modelling approaches (e.g., OWL-S [99], OWL-L [100]) mostly describe the services by three semantic documents: Service Interface, Process Model, and Physical Grounding. We represent them as Interface, Process and Grounding elements. On the other hand, to realise agents communicating with a service registry and to discover service capabilities, an entity called the Semantic Service Matchmaker Agent is used that stores the capability advertisements of SWSs within a MAS and matches those capabilities with the service requirements sent by the other platform agents by playing its Registration Role.

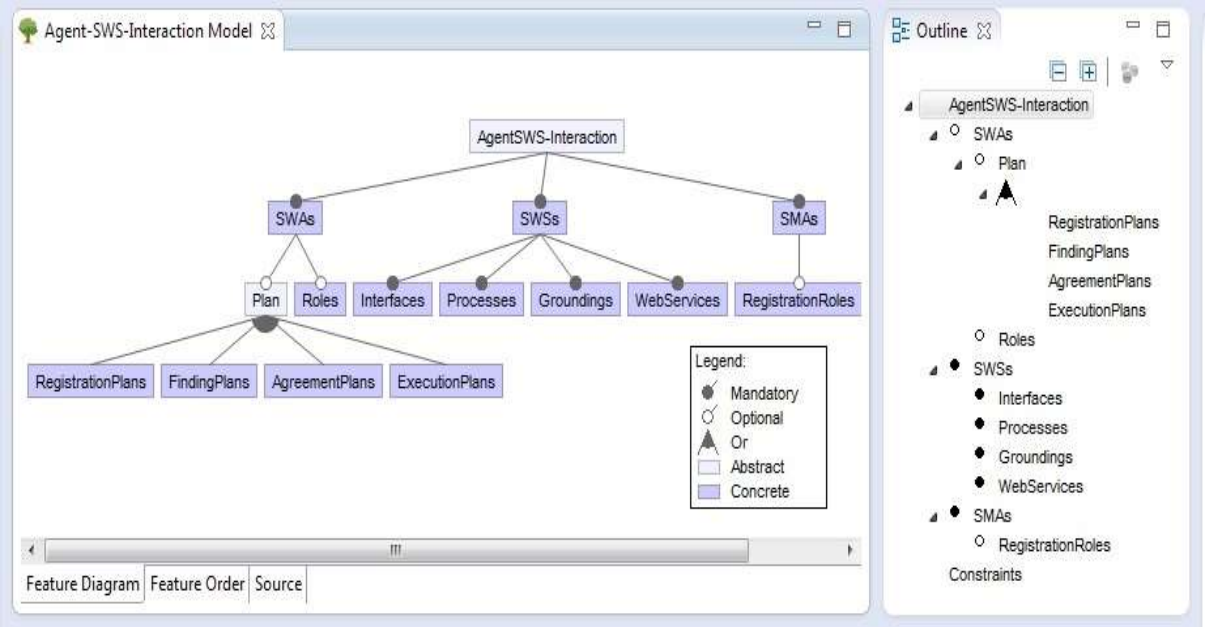

Figure 3: Feature diagram for Agent-SWS Interaction

The second step in our DSL development methodology (Section 3) is DSL designing where the language syntax and semantics are defined using a metamodel for describing a structure and denotational/translational semantics for describing a meaning. Concepts from a feature diagram (e.g., Figure 3) become classes within the conceptual class diagram [101], while the relationships amongst the concepts within the feature diagram become associations enhanced with cardinality, navigability, composition, aggregation, and inheritance. Note that a conceptual class diagram is just another name

for a metamodel where, at higher abstraction levels, classes are used to describe different concepts within a domain (e.g. a declaration within a general-purpose programming language). While, at a more concrete level, classes are used to describe the structure of a concept (e.g. a variable declaration consists of a variable type and a variable name). Figure 4 presents a metamodel for the Agent-SWS Interaction sub-language. It was derived at from the feature diagram (Figure 3) by adding additional information about cardinality, and the names of associations.

In Figure 4, the elements filled-in with light grey come from those other viewpoints that are shown at the top or bottom of the element using "<<" and ">>" stereotypes. In other words, these elements are common elements amongst viewpoints, and tailored to each other according to Figure 2. For example, in the Agent-SWS Interaction viewpoint, the Role meta-element comes from <<Role Viewpoint>> (see Figure 4). Furthermore, the main element of the viewpoint is depicted using darker borders. Taking the Agent-SWS Interaction viewpoint into consideration, SWS is the main element of the viewpoint and has a darker border than the other elements of the viewpoint (see Figure 4).

SWS describes services by Service Interface, Process Model, and Physical Grounding. Service Interface is represented as the Interface element and provides the capability representation of the service in which service Input, Output, Precondition, and Effect (IOPE) descriptions are listed. The Process Model, as represented by the Process element, describes the internal composition and execution dynamics of the semantic service including IOPE. Finally, Physical Grounding, represented by the Grounding element, defines the invocation protocol of the web service. As the operational part of today's semantic services is mostly a web service, the Web Service concept is also included within the metamodel and associated with the grounding mechanism.

SWAs apply Plans for discovering, negotiating and executing the SWSs dynamically. The SS_FinderPlan is a Plan in which the discovery of candidate SWSs (considering their Interfaces) takes place. The SS_AgreementPlan involves negotiation on the QoS metrics of the service (e.g., service execution cost, running time, location) and an agreement settlement using the service Interface. After service discovery and negotiation, the agent applies the SS_ExecutorPlan for executing appropriate SWSs regarding their Processes and Groundings.

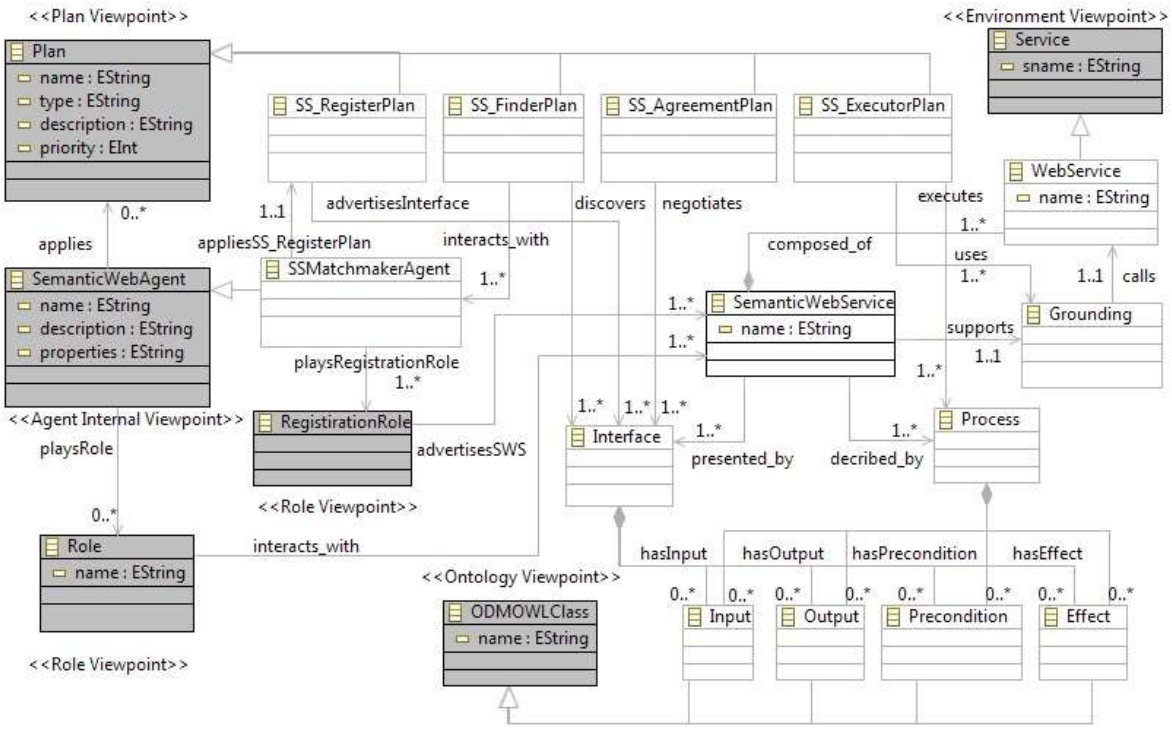

Figure 4: Agent-SWS Interaction metamodel [14]

However, a formal semantic description is usually based on abstract syntax. Here, we are using transformations between metamodel and BNF, as described in [93, 94]. The abstract syntax given in Listing 1 was derived at from the metamodel shown in Figure 4.

Note that in the abstract syntax, a non-terminal PROGRAM is denoted as the main concept. Classes from the metamodel are mapped to non-terminal symbols inside this starting non-terminal. For the purpose of readability, it has been decided to move most of the non-terminal definitions inside a new non-terminal DEFS. The exceptions are the main concepts: the semantic web agents (SWA), semantic web services (SWS), and semantic matchmaker agent (SMA). These metamodel classes may be represented in the abstract syntax as keywords. In this manner, SWA is denoted by the keyword "swa". Furthermore, each non-terminal is defined with a corresponding metamodel's class attributes and connections. For instance, the terminal symbol "id" stands for SemanticWebAgent attribute "name", and non-terminal APPLIES for connection with the class PLAN. Note that the cardinality of connections also needs to be transformed into the abstract syntax. For instance, from the metamodel we can see that SWA may have several Plans. For this purpose, non-terminal APPLIES contains first and third productions that bring the possibility of several plans' definitions inside the semantic web agent.

```
PROGRAM      ::= DEF SWA SWS SMA
DEF          ::= REGPLAN FINDPLAN AGREEPLAN EXECPLAN ROLE REGROLE INTERFACE
                 PROC GROUNDING
SWA          ::= swa id APPLIES PLAYS DESCRIPTION PROPERTIES
             |   SWA1 SWA2
SWS          ::= sws id DESCRIBE PRESENT SUPPORT
             |   SWS1 SWS2
SMA          ::= sma id APPLIESREGPLAN PLAYSREGROLE
             |   SMA1 SMA2
APPLIES      ::= applies id
             |   APPLIES1 APPLIES2
             |   ε
PLAYS        ::= plays  id
             |   PLAYS1 PLAYS2
             |   ε
DESCRIPTION  ::= description id
PROPERTIES   ::= properties id
DESCRIBE     ::= describedBy id
             |   DESCRIBE1 DESCRIBE2
PRESENT      ::= presentedBy  id
             |   PRESENT1 PRESENT2
SUPPORT      ::= supportedBy id
APPLIESREGPLAN ::= appliesRegPlan id
PLAYSREGROLE ::= playsRegRole id
             |   PLAYSREGROLE1 PLAYSREGOLE2
REGPLAN      ::= regPlan id ADVERTISE TYPE DESCRIPTION PRIORITY
             |   REGPLAN1 REGPLAN2
ADVERTISE    ::= advertises id
             |   ADVERTISE1 ADVERTISE2
TYPE         ::= type id
PRIORITY     ::= priority num
FINDPLAN     ::= findPlan id INTERACT DISCOVER TYPE DESCRIPTION PRIORITY
             |   FINDPLAN1 FINDPLAN2
             |   ε
INTERACT     ::= interactsWith id
             |   INTERACT1 INTERACT2
DISCOVER     ::= discovers id
             |   DISCOVER1 DISCOVER2
AGREEPLAN    ::= agreePlan id {NEGOTIATE TYPE DESCRIPTION PRIORITY }
             |   AGREEPLAN1 AGREEPLAN2
             |   ε
NEGOTIATE    ::= negotiates id
             |   NEGOTIATE1 NEGOTIATE2
EXECPLAN     ::= execPlan id USE EXECUTE TYPE DESCRIPTION PRIORITY
             |   EXECPLAN1 EXECPLAN2
             |   ε
USE          ::= uses id
             |   USE1 USE2
EXECUTE      ::= executes id
             |   EXECUTE1 EXECUTE2
ROLE         ::= role id INTERACT
             |   ROLE1 ROLE2
             |   ε
REGROLE      ::= RegRole id ADVERTISE
             |   REGROLE1 REGROLE2
INTERFACE    ::= interface id
             |   INTERFACE1 INTERFACE2
PROC         ::= process  id
             |   PROC1 PROC2
GROUNDING    ::= grounding id
             |   GROUNDING1 GROUNDING2
```

Listing 1: Abstract Syntax for Agent-SWS Interaction sub-language

When the abstract syntax is defined, we can develop a concrete syntax in parallel, as needed for parsing and semantics, and needed for describing a meaning. A part of concrete syntax for Agent-SWS Interaction sub-language is shown in Listing 2 where additional keywords are added and multiplicities are exactly defined.

```
PROGRAM      ::= AgentSWSinteraction #Identifier \{ DEFS SWAS SWSS SMAS \}
DEFS         ::= REGPLANS FINDPLANS AGREEPLANS EXECPLANS ROLES REGROLES
                 INTERFACEIDS PROCIDS GROUNDINGID
SWAS         ::= SWA SWAS
             |   SWA
SWA          ::= swa #Identifier \{ APPLIES PLAYS DESCRIPTION PROPERTIES \}
SWSS         ::= SWS SWSS
             |   SWS
SWS          ::= sws #Identifier \{ DESCRIBES PRESENTS SUPPORTS \}
SMAS         ::= SMA SMAS
             |   SMA
SMA          ::= sma #Identifier \{ APPLIESREGPLAN PLAYSREGROLES \}
...
APPLIES      ::= APPLY APPLIES
             |   epsilon
APPLY        ::= applies #Identifier \;
PLAYS        ::= PLAY PLAYS
             |   epsilon
PLAY         ::= plays #Identifier \;
DESCRIPTION  ::= description \" #Identifier \" \;
PROPERTIES   ::= properties \" #Identifier \" \;
```

Listing 2: Part of the concrete Syntax for the Agent-SWS Interaction sub-language

Complex constraints and relationships amongst concepts in the Agent-SWS Interaction sub-language is captured by denotational semantics. In order to define the meaning of language constructs, we first define their semantic domains followed by auxiliary functions and the semantic equations. The semantic domain describes the domains for each syntactic construct within the language. Part of the semantic domain is presented in Listing 3. In addition some auxiliary functions are required for facilitating the implementation of semantic rules. Listing 3 shows the semantic domains and auxiliary functions for SWA, SWS, and SMA. For example, a "SW-Agents" is a domain mapping function which maps an id of the agent into a set of plans and a set of roles (along with additional description and properties) that are associated with that agent. The Plan, as defined for SWA, is a collection of FinderPlans, AgreementPlans and ExecutorPlans. In a similar way, the domains are defined for the FinderPlan, AgreementPlan and ExecutorPlan. Some of the auxiliary functions are also shown in Listing 3 for the domain elements. For example the getPlans function of SWA takes the id of the agent, the list of the agents, and returns the plans for that agent id.

The meaning of the language is defined using semantic equations of denotational semantics. Part of the semantic equations for SWA, SWS, and their relationships are given in Listing 4.

Lines 2 and 3 of Listing 4 define the meaning of the "swa" rule within the concrete syntax. The meaning of the rule expression "swa id APPLIES PLAYS" for context elements such as *ag*, *pl*, and *ro* (representing agent, plan, and role) is a mapping from the id of the swa agent to the meaning of the APPLIES rule within the context of *pl* and the meaning of PLAYS rule within the context of ro. These denotations are defined in A1 and A2 respectively. Line 4 defines the rule called swas within the concrete syntax. Similarly the meanings of APPLIES and PLAYS rules are defined in Lines 6-9 and 11-14, respectively. The semantic equations in Listing 4 collect any necessary information about SWA, SWS, and SMA in order to perform various controls on the constraints between the concepts. For example, the following four semantic controls are considered, which cover almost all of the relationships in the metamodel (Figure 4):

```
01 // Semantic Web Agents
02 SW-Agents = Id → Plan* × Role* × Description × Properties
03 Plan = RegPlan + FindPlan + AgreePlan + ExecPlan
04 RegPlan = Id → Interface+ × Type × Description × Priority
05 FindPlan = Id → SM-Agents+ × Interface+ × Type x Description × Priority
06 AgreePlan = Id → Interface+ × Type x Description × Priority
07 ExecPlan = Id → Grounding + × Process+ × Type × Description × Priority
08 Description = Id
09 Properties = Id
10 Role = Id → SW-Services+
11
12 getPlans(id, SW-Agents): return Plans for web agent id
13 getRoles(id, SW-Agents): return Roles for web agent id
14 getDescription(id, SW-Agents): return Description for web agent id
15 getProperties(id, SW-Agents): return Properties for web agent id
16
17 // Semantic Web Services
18 SW-Services = Id → Process+ × Interface+ × Grounding+
19 Process = String
20 Interface = String
21 Grounding = String
22
23 getProcess(id, SW-Services): return Processes of service id
24 getInterface(id, SW-Services): return  Interfaces of service id
25 getGrounding(id, SW-Services): return Grounding of service id
26
27 //Semantic Matchmaker agents
28 SM-Agents = Id → RegPlan × RegRole+
29 RegRole = Id → SW-Service+
30
31 getRegPlan(id, SM-Agents) : returns RegPlan of agent id
32 getRegRoles(id, SM-Agents):  return RegRoles of agent id
```

Listing 3: Semantic Domain and Auxiliary Functions of Denotational Semantics

A. During the design of an Agent-SWS Interaction, if a SWA applies the FinderPlan that discovers an Interface of a SWS then, as the result of this discovery can be positive, at least one of the following interactions (2, 3, and 4) should exist between the Agent and SWS through AgreementPlan and ExecutorPlan. In a short form:
   If
      1) SWA-FindPlan-Interface-SWS
   Then
      2) SWA-AgreePlan-Interface-SWS and
      3) SWA-ExecPlan-Process-SWS and
      4) SWA-ExecPlan-Grounding-SWS

B. The philosophy behind Agent-SWS Interaction is the designing of possible interactions between a SWA and a SWS. So, for each SWA, there should be at least one interaction in some way with one of the SWSs. This interaction can be realised using one of the following:
   1) SWA-Role-SWS
   2) SWA-FindPlan-Interface-SWS

C. If a SWA associates with a SMA and an Interface through a FinderPlan, then the SMA should associate with the same Interface through RegPlan, and with the SWS of the Interface, through RegRole. In a short form:

If

1) SWA-FindPlan-SMA and
2) SWA-FindPlan-Interface

Then

3) SMA-RegPlan-Interface and
4) SMA-RegRole-SWS

D. For each SWS there is a SWA, such that one of the rules in constraint B is true for these SWAs and SWSs. In other words, for each SWS, there should be a SWA in such a way that one of the followings is true:

1) SWA-Role-SWS
2) SWA-FindPlan-Interface-SWS

```
01 A: SWA → SW-Agents × Plan* × Role* → SW-Agents
02 A ⟦swa id APPLIES PLAYS DESCRIPTION PROPERTIES ⟧(ag, pl, ro) =
03     ag[id →(findPlans(A1⟦ APPLIES ⟧, pl), findRoles(A2⟦ PLAYS ⟧, ro), A3⟦DESCRIPTION⟧, A4⟦PROPERTIES⟧)]
04 A⟦ SWA1 SWA2 ⟧(ag, pl, ro) = A⟦SWA2⟧(A⟦SWA1⟧  ag, pl, ro)
05
06 A1: APPLIES → Id*
07 A1⟦ε ⟧= ∅
08 A1⟦applies  id ⟧=  { id }
09 A1⟦APPLIES1 APPLIES2⟧= append(A1⟦APPLIES1⟧, A1⟦APPLIES2⟧)
10
11 A2: PLAYS → Id*
12 A2⟦ε ⟧= ∅
13 A2⟦plays  id ⟧=  { id }
14 A2⟦PLAYS1 PLAYS2⟧= append(A2⟦PLAYS1⟧, A2⟦PLAYS2⟧)
15
16 A3: DESCRIPTION → Id
17 A3⟦description  id ⟧=  { id }
18
19 A4: PROPERTIES → Id
20 A4⟦properties  id ⟧=  { id }
21
22 B: SWS → SW-Services × Process* × Interface* × Grounding→  SW-Services
23 B⟦ sws id DESCRIBE PRESENTS SUPPORTS⟧(ser, pro, int, gro) =
24    ser[ id → (findProcesses(B1⟦DESCRIBE⟧ pro), findInterfaces(B2⟦PRESENTS⟧ int),
25                                                  findGrounding(B3⟦SUPPORTS⟧, gro))]
26 B⟦SWS1 SWS2 ⟧(ser, pro, int, gro)  = B⟦SWS2⟧(B⟦SWS1⟧ser, pro, int, gro)
27
28 B1: DESCRIBE → Id+
29 B1⟦describedBy  id ⟧=  { id }
30 B1⟦DESCRIBED1 DESCRIBED2⟧= append(B1⟦DESCRIBE1⟧, B1⟦DESCRIBE2⟧)
31
32 B2: PRESENTS → Id+
33 B2⟦presentedBy  id ⟧=  { id }
34 B2⟦PRESENTS1 PRESENTS2⟧= append(B2⟦PRESENTS1⟧, B2⟦PRESENTS2⟧)
35
36 B3: SUPPORTS → Id
37 B3⟦supportedBy  id ⟧=  { id }
```

Listing 4: Part of the semantic equations for the SWA and SWS of the SEA_L language

In order to understand these semantic controls more easily, constraint A including four rules is illustrated on the Agent-SWS Interaction diagram, see Figure 5. Some of the unrelated elements e.g. IOPE are unconsidered to make the rules simpler to understand.

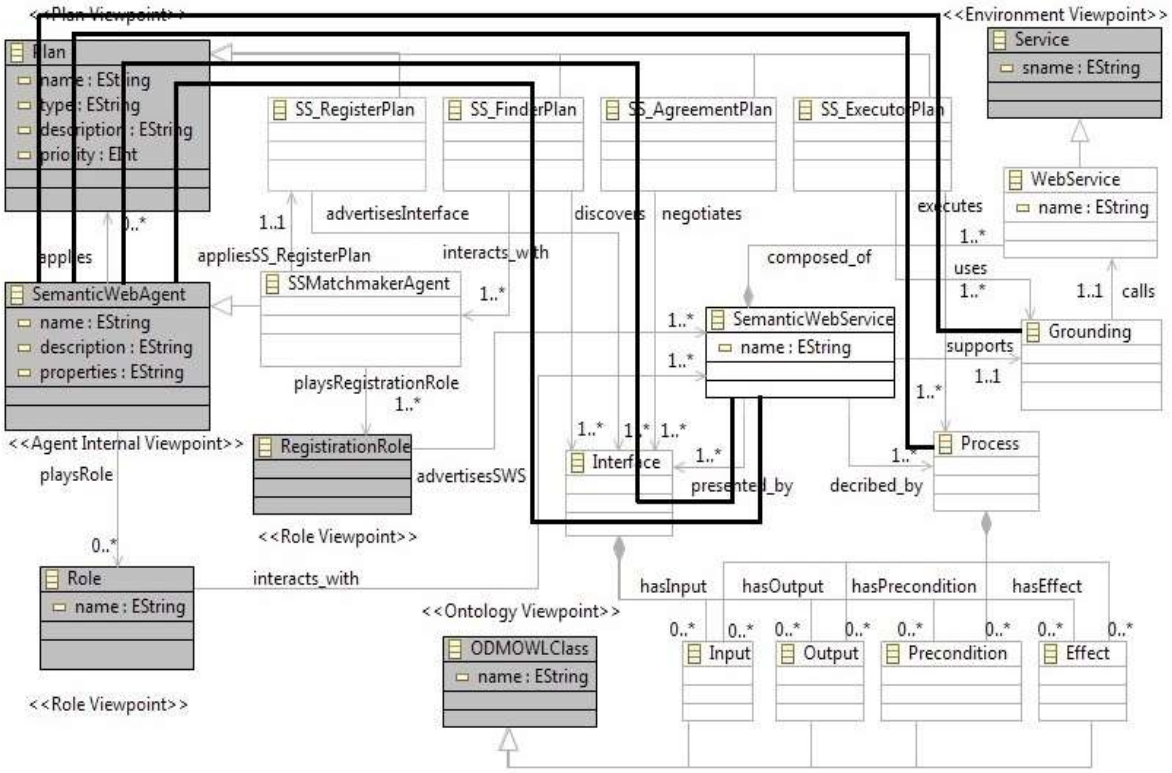

Figure 5: Demonstration of constraint A on Agent-SWS Interaction diagram

Denotational semantics of a SEA_L program is used for specifying its meaning. To this end, the denotational equation of semantic function **P** with constraint A is shown in Listing 5. Semantic function **P** defines the meaning of a program with the meaning of its main components such as SWA, SWS, and SMA. Correspondingly, the meanings of these components are defined by their plans, roles for SWAs, processes, interfaces and groundings for SWSs, and registration plan and registration roles for SMA.

```
01 P: Program → Code
02 P⟦DEF SWA SWS SMA ⟧= let (fp, ap, ep, ro, pro, int, gro, rp, rr) = D⟦DEF⟧,
03          swa = A⟦ SWA ⟧(∅, fp+ap+ep, ro) ,
04          sws = B⟦SWS ⟧ (∅, pro, int, gro) ,
05          sma = C⟦SMA ⟧(∅, rp, rr)  in
06          if (∀id-swa, id-sws,id-fp, id-ap, id_ep, id-int, id-pro, id-gro:  // constraint A
07                    // SWA-FindPlan-Interface-SWS
08                    id-fp ∈ swa(id-swa )↑1 and
09                    id-int ∈ fp(id-fp) ↑2 and
10                    id-int ∈ sws(id-sws) ↑2 and
11                    // SWA-AgreePlan-Interface-SWS
12                    id-ap ∈ swa(id-swa )↑1 and
13                    id-int ∈ ap(id-ap)  and
14                    // SWA-ExecPlan-Process-SWS
15                    id-ep ∈ swa(id-swa )↑1 and
16                    id-pro ∈ ep(id-ep) ↑2 and
17                    id-pro  ∈ sws(id-sws) ↑1 and
18                    // SWA-ExecPlan-Grounding-SWS
19                    id-gro ∈ ep(id-ep) ↑1 and
20                    id-gro  = sws(id-sws) ↑3)
21          then   generateCode(swa, sws,sma)
22          else   "constraint check error"
23
```

Listing 5: Denotational equation of semantic function **P** with constraint A.

The third and last step in our DSL development methodology (Section 3) is a DSL implementation, where a complete compiler/interpreter is automatically generated from formal specifications written in LISA tool. LISA specifications are attribute grammar-based and consist of lexical, syntax, and semantic parts (Listing 6). The LISA tool produces highly efficient source code for the scanner, parser, and interpreter/compiler in Java. The lexical and syntactical parts of a language's specification use well-known formal methods, such as regular expressions and BNF. The semantics are further defined using attribute grammars.

```
01 language L1 [extends L2, ..., LN ] {
02     lexicon {
03         [[Q] overrides | [Q] extends] R regular expr.
04     }
05     attributes type At1, ..., AtM
06     rule [[Y] extends | [Y] overrides] Z {
07         X ::= X11 X12 ... X1p compute {
08             semantic functions
09         |
10         Xr1 Xr2 ... Xrt compute {
11             semantic functions
12         };
13     }
14     method [[N] overrides | [N] extends] M {
15         operations on semantic domains
16     }
17 }
```

Listing 6: Template of LISA for language specification definition

The lexicon part of the Agent-SWS Interaction sub-language is defined as Listing 7 in LISA. In addition, those attributes that are needed for some of the non-terminals are shown in Listing 8. Each attribute type can be any valid type of Java, including any class which we define. So, as we can see in Listing 8, "val" and "code" are attributes of type Java String but the attribute called "pro" is of type class Process, which is defined at the end of the language specification definition in LISA.

```
01 ReservedWord  AgentSWSinteraction | swa | applies | plays | findPlan | interactsWith | discovers | type
02     | description | priority | properties | agreePlan | negotiates | execPlan | uses | executes | role | describedBy
03     | presentedBy | supportedBy | interface | grounding | appliesRegPlan | playsRegRole | regPlan
04     | advertises | regRole
05  Separator   \{ | \}
06  Number      [1-9][0-9]*
07  Identifier  [a-zA-Z0-9]+
08  Delimiter   \;
09  Quotes      \"
10  ignore      [\0x0D\0x0A\ \0x09]+ | \/\/[^\0x0A\0x0D]*
```

Listing 7: Lexicons of Agent-SWS Interaction sub-language in LISA

Attributes from Listing 8 directly correspond to the semantic domain defined by the denotational semantics in Listing 3. Some attributes have been added to support four different constraints and code generation, as defined in the denotational equations (Listing 5 shows just the equation for constraint A).

Therefore, an attribute definition such as "Grounding *.gro" means that an attribute named "gro" of type class Grounding is added to all the non-terminals of the language (due to *). Of course it is

possible to define an attribute to a specific non-terminal, e.g. "Grounding GROUNDINGID.gro", however, to ease the programming the attributes are defined for all of the non-terminals.

Generally in LISA, non-terminals are defined in capital letters and the first rule is automatically considered as the starting rule of the language. The start rule of the Agent-SWS Interaction sub-language in LISA is shown in Listing 9.

```
01 attributes
02   SWAgent *.ag;
03   AllPlans *.pl;
04   RegPlan *.rp;
05   FindPlan *.fp;
06   AgreePlan *.ap;
07   ExecPlan *.ep;
08   Role *.ro;
09   SWServices *.sws;
10   Process *.pro;
11   Interface *.inter;
12   Grounding *.gro;
13   SMA *.sma;
14   RegRole *.rr;
15   String *.val;
16   ArrayList *.outEnv;
17   String *.code;
```

Listing 8: Attributes of Agent-SWS Interaction sub-language in LISA

According to Listing 9, the rule named "program" is the starting rule of the language. So, the starting non-terminal is PROGRAM. As we can see, the attributes for that non-terminal are obtaining their values from child nodes which mean they are synthesised attributes. This is because, the starting non-terminal has no parent node and its attributes are always of the type synthesised. In this rule, all the attributes of the other non-terminals (DEFS, SWSS, SMAS, SWAS) are inherited attributes. Note that the generated code is kept in an attribute of the element called “code”, as defined in the attribute section (Listing 8). The main task of the specifications in Listing 9 is to collect important information from DEFS, SWSS, SMAS, and SWAS. The collected data is then used to generate a code. Note that the constraints from Listing 5 are not yet implemented. In other words, the code is generated regardless of whether the constraints are fulfilled or not.

```
01 rule program {
02   AgentSWSinteraction #Identifier \{ DEFS SWAS SWSS SMAS \} compute {
03       SWAS.pl = DEFS.pl;
04       SWAS.ro = DEFS.ro;
05       SWSS.pro = DEFS.pro;
06       SWSS.inter = DEFS.inter;
07       SWSS.gro = DEFS.gro;
08       SMAS.rp = DEFS.rp;
09       SMAS.rr = DEFS.rr;
10       DEFS.sma = SMAS.sma;
11       DEFS.sws = SWSS.sws;
12       PROGRAM.sws = SWSS.sws;
13       PROGRAM.sma = SMAS.sma;
14       PROGRAM.ag = SWAS.ag;
17       PROGRAM.code = generate(SWSS.sws, SMAS.sma, SWAS.ag);
18   };
19 }
```

Listing 9: Starting rule of Agent-SWS Interaction sub-language in LISA

In a similar manner, all of the required rules are defined to allow the developer to generate all the necessary elements and relationships, as is stated in the metamodel. For example, implementation of the semantic equations of SWS rules (named B, B1, B2, and B3 in Listing 4) is shown in Listing 10.

```
01 rule swss {
02   SWSS ::= SWS SWSS compute {
03     SWS.pro = SWSS[0].pro;
04     SWSS[1].pro = SWSS[0].pro;
05     SWS.inter = SWSS[0].inter;
06     SWSS[1].inter = SWSS[0].inter;
07     SWS.gro = SWSS[0].gro;
08     SWSS[1].gro = SWSS[0].gro;
09     SWSS[0].sws = addSWS(SWS.sws, SWSS[1].sws);
10   }
11   | SWS compute {
12     SWS.pro = SWSS.pro;
13     SWS.inter = SWSS.inter;
14     SWS.gro = SWSS.gro;
15     SWSS.sws = SWS.sws;
16   };
17 }
18 rule sws {
19   SWS ::= sws #Identifier \{ DESCRIBES PRESENTS SUPPORTS \} compute {
20     SWS.sws = newSWS(#Identifier.value(), findProcess(DESCRIBES.outEnv, SWS.pro),
21         findInterface(PRESENTS.outEnv, SWS.inter), findGrounding(SUPPORTS.val, SWS.gro));
22   };
23 }
24 rule describes {
25   DESCRIBES ::= DESCRIBE DESCRIBES compute {
26     DESCRIBES[0].outEnv = append(DESCRIBES[1].outEnv, DESCRIBE.val);
27   }
28   | DESCRIBE compute {
29     DESCRIBES.outEnv = append(new ArrayList<String>(), DESCRIBE.val);
39   };
31 }
32 rule describe {
33   DESCRIBE ::= describedBy #Identifier \; compute {
34     DESCRIBE.val = #Identifier.value();
35   };
36 }
37 rule presents {
38   PRESENTS ::= PRESENT PRESENTS compute {
39     PRESENTS[0].outEnv = append(PRESENTS[1].outEnv, PRESENT.val);
40   }
41   | PRESENT compute {
42     PRESENTS.outEnv = append(new ArrayList<String>(), PRESENT.val);
43   };
44 }
45 rule present {
46   PRESENT ::= presentedBy #Identifier \; compute {
47     PRESENT.val = #Identifier.value();
48   };
49 }
50 rule supports {
51   SUPPORTS ::= supportedBy #Identifier \; compute {
52     SUPPORTS.val = #Identifier.value();
53   };
54 }
```

Listing 10: Implementation of semantic equations for SWS related rules of Agent-SWS Interaction sub-language in LISA

The “sws” rule (Line 18 of Listing 10) represents semantic equation B (Line 21 in Listing 4) where the “sws” has an id, set of Processes, set of Interfaces and a Grounding implemented via DESCRIBES, PRESENTS, and SUPPORTS rules. These rules are implemented in Lines 24, 32 , and 37 of Listing 10 for the semantic equations B1, B2, and B3, respectively (Lines 27, 31, and 35 of Listing 4).

Implementations of some of the auxiliary functions are displayed in Listing 11.

```
01 method Utility {
02       public SWServices newSWS(String iden, Process pro, Interface inter,
03                         Grounding gro) {
04              SWServices sws = new SWServices();
05              ProcessInterfaceGrounding pig = new ProcessInterfaceGrounding();
06              pig.addProcess(iden, pro);
07              pig.addInterfaces(iden, inter);
08              pig.addGrounding(iden, gro);
09              sws.addSWServices(iden, pig);
10              return sws;
11       }
12 }
13 class SWServices {
14              private Hashtable<String, ProcessInterfaceGrounding> sws;
15              public SWServices() {       sws = new Hashtable<String, ProcessInterfaceGrounding>();  }
16              public SWServices(Hashtable<String, ProcessInterfaceGrounding> s) { super(); this.sws = s; }
17              public Hashtable<String, ProcessInterfaceGrounding> getServices() { return this.sws; }
18              public ProcessInterfaceGrounding getServicesByID(String id) { return this.sws.get(id);  }
19              public void setServices(Hashtable<String, ProcessInterfaceGrounding> s) { this.sws = s; }
20              public void addSWServices(String id, ProcessInterfaceGrounding pr) { this.sws.put(id, pr);  }
21 }
```

Listing 11: Implementations of some of the auxiliary functions for SWS related rules of Agent-SWS Interaction sub-language in LISA

According to the code of Listing 11, in order to create any new element within the new language, e.g. new SWS, a hash table is created and all the instances of the element are stored within it, SWServices. In fact this class, SWServices, is a type of attribute of the SWS. As a result, during compilation both the abstract syntax tree and semantic tree are built. The prior helps to find out if the structure/syntax of the program is accurate. The latter uses the computations and auxiliary functions, and creates hash tables as attributes and gathers whole information about the instances of the elements and their relationships, within these hash tables. The attributes can be strings representing the code needed to be generated for the element. This can help to provide code generation.

The modularity of a specification can easily be achieved in LISA through language inheritance [22]. We have decided to implement the constraints separately from the generation of the code. It can be learned, from Listing 12, that language SWSIV_Constraints is an extension of language SWSIV_Generation. The extension of SWSIV_Generation was possible in a non-invasive way and corresponds to language extension pattern as a special case of language composition [81]. So, within the first language (see Listings 9) there is the functionality of code generation. The second language provides constraints checking. The only rule that needs to be written inside language SWSIV_ Constraints is the rule “program”. Note, that this one is the first rule of the language SWSIV_Generation and we are actually extending it. The attributes of language SWSIV_ Constraints are those that carry the results of constraints checking (constA, constB, constC, and constD) and are further given to the function that is responsible for the code generation. Note that attribute “code” is evaluated in both languages and as such the attribute grammar fragments are conflicting. Hence, the approach by Sarasa-Cabezuelo and Sierra [80] would be inapplicable for this example. However, in our approach the rule from language SWSIV_ Constraints extends the rule from the core language,

and the semantic equation from the first language is overridden by the semantic equation from the language SWSIV_ Constraints.

```
language SWSIV_Constraints extends SWSIV_ Generation {
    attributes
       boolean *.constA;
       boolean *.constB;
       boolean *.constC;
       boolean *.constD;

    rule extends program {
    PROGRAM ::= AgentSWSinteraction #Identifier \{ DEFS SWAS SWSS SMAS \} compute {

      PROGRAM.constA = checkConstraintA(SWAS.ag, DEFS.sws);
      PROGRAM.constB = checkConstraintB(SWAS.ag, DEFS.sws);
      PROGRAM.constC = checkConstraintC(SWAS.ag, DEFS.sws, SMAS.sma,
                                                DEFS.inter);
      PROGRAM.constD = checkConstraintD(SWAS.ag, DEFS.sws);

      PROGRAM.code = generate(SWSS.sws, SMAS.sma, SWAS.ag,
                          PROGRAM.const1, PROGRAM.const2, PROGRAM.const3, PROGRAM.const4);
    };
    }

   ...
}
```

Listing 12: Modularity example in LISA for SEA_L

During the parsing of the program, the code is generated while the parse tree is being established and is merged while traversing between the nodes of the tree. Finally, at the end of compiling, all the attributes including the code attribute are collected within the start symbol of the grammar, “PROGRAM”. According to the computation rules for the start symbol, all of the code attributes are collected within the code attribute of the start symbol using the “generate” function (Listing 13).

```
public  String  generate(SWServices sws, SMA sma, SWAgent swa,
                         boolean constA, boolean constB, boolean constC, boolean constD) {
    if (constA && constB && constC && constD){
        sws.generateFiles("Templates\\ServiceTemplate");
sws.generateFiles("Templates/ServiceTemplate", "swsivOUT/", "Service");
        sws.generateFiles("Templates/ProfileTemplate", "swsivOUT/", "Profile");
        swa.generateFilesJava("Templates/", "swsivOUT/", ".java");
        sma.generateFiles("Templates/RegPlanTemplate", "swsivOUT/", ".java");
        swa.generateFilesXML("Templates/", "ADFTemplate", "swsivOUT/", ".xml");
  }
  return "check files in folder 'swsivOUT/'";
}
```

Listing 13: Code generation of Agent-SWS Interaction sub-language in LISA

When considering the invocation of constraints A (in Listing 9 and 12), an attribute is added to the language specifications called “constA” and a computation rule is added to the start symbol as

PROGRAM.constA = checkConstraintInteractionAgentSWS(SWAS.ag, DEFS.sws);

The auxiliary function called checkConstraintInteractionAgentSWS, performs the required semantic controls declared in constraint A using some other auxiliary functions, and other attributes. This function is illustrated in Listing 14.

In Listing 14, the list of agents including their plans and role relationships are taken in Line 1 and iteration is started in Line 5 over these plans and roles. Then the plans are extracted from the hash table in Line 7 and in the case of it not being empty, iteration is started over those plans in Line 10. Then if a plan is the type of FinderPlan in Line 12 (until now we have found a FinderPlan associated with our SWA), the control continues as follows: In Line 13, the elements of the FinderPlan and its relationships with other elements are extracted from its hash table and in Line 15 iteration is started over its elements (SMA and Interface). Line 18 checks if there is any Interface, with which the FinderPlan is associated, the list of the interfaces is extracted in Line 19, and iteration starts over them in Line 20. Until now, for our SWA, we have found an associated FinderPlan that has an association with an Interface. In the next step, we are going to find a SWS that the Interface describe. This is fulfilled in Lines 23-31. In Lines 32 and 33, if a SWS is found, the result of the first part of the semantic rule is true.

```
01 public boolean checkConstraintInteractionAgentSWS(SWAgent swa, SWServices sws) {
02   boolean ret = false;
03   Hashtable<String, PlansRoles> plansroles = swa.getAgents();
04   Iterator<PlansRoles> itr = plansroles.values().iterator();
05   while (itr.hasNext()) {
06     PlansRoles elementPlanRole = itr.next();
07     Hashtable<String, Plan> plans = elementPlanRole.getPlan();
08     if (!plans.isEmpty()) {
09       Iterator<Plan> itr2 = plans.values().iterator();
10       while (itr2.hasNext()) {
11         Plan elementPlan = itr2.next();
12         if (elementPlan instanceof FindPlan) {
13           Hashtable<String, AgentInterface> fp = ((FindPlan) elementPlan).getFindPlans();
14           Iterator<AgentInterface> itr3 = fp.values().iterator();
15           while (itr3.hasNext()) {
16             AgentInterface elementAgentInterface = itr3.next();
17             Hashtable<String, Interface> interfaces = elementAgentInterface.getInterfaces();
18             if (!interfaces.isEmpty()) {
19               Iterator<Interface> itr4 = interfaces.values().iterator();
20               while (itr4.hasNext()) {
21                 Interface interf = itr4.next();
22                 for (String s : interf.getInter()) { // interfaces
23                   Hashtable<String, ProcessInterfaceGrounding> services = sws.getServices();
24                   Iterator<ProcessInterfaceGrounding> itr5 = services.values().iterator();
25                   while (itr5.hasNext()) {
26                     ProcessInterfaceGrounding pig = itr5.next();
27                     Hashtable<String, Interface> sis = pig.getInterfaces();
28                     if (!sis.isEmpty()) {
29                       Iterator<Interface> itr6 = sis.values().iterator();
39                       while (itr6.hasNext()) {
31                         Interface si = itr6.next();
32                         for (String ss : si.getInter()) { // interfaces
33                               if (ss.equals(s)) return true;
34   }}}}}}}}}}}}
35   return ret;
36 }
```

Listing 14: Implementation of constraint A in LISA

This means that there is a FinderPlan for the SWA with which it associates and the FinderPlan discovers an Interface of a SWS. The three other constraints are controlled in a similar manner and the result is manipulated using the logical ‘and’ operation, since “if A then B and C” is logically equal to “A and B and C”. The final result is used to control whether the code should be generated or a special error-code should be produced.

It is also possible to generate specific error codes depending on the place at which the rule goes wrong. The mechanism for this kind of error generation is conceptually specified in Listing 5 with the denotational equations for the semantic function (named P) of a SEA_L program considering constraint A (Listing 4). Accordingly, during implementation, different parts of a constraint, such as constraint A, can be checked and in the case of any violation, an error can be generated. For example, in Listing 14, considering the implementation of constraint A, it is possible to distinguish various cases for violation of the constraint and generate different errors, e.g. if a SWA has no plan at all in Line 08, we can return a specific error code. Similarly, in Line 18, we can check if there is no interface connected to the plan of the SWA, which violates the constraint A and is another type of error for constraint A.

5. Case-Study: Development of an Agent-based Expert Finding System

SEA_L can be utilised for different domains when helping in the designs and developments of the agent-based systems of those domains such as agent-based business evaluation [102], e-commerce [103], document management [104], streaming on networks [105], energy saving [106], the e-barter system [107], and the stock exchange system [108]. In order to illustrate the usage of the introduced DSL and its semantics, the modelling of an Agent-SWS interaction for a multi-agent-based expert finding system is considered in this study.

Let us consider that there exist web services for supplying expert needs for people. Suppose that the software agents in an expert finding system work on the web by using service ontologies, find candidate services, and then try to make an agreement with those services on behalf of their human users by taking into consideration QoS metrics. Starting from a motivating example, we discuss the development of the system in the remainder of this section.

As a motivating example, consider the following scenario in which a user (person) requests an expert on communication services. The user Ann wants to communicate with her cousin Lee. However, she has had no contact with her for a long time and does not have Lee's contact information including what kind of communication services Lee uses, and those services which could be used to contact her (e.g. Social network, e-mail, VoIP, and mobile phone number).

First of all, Ann has to find the right person anyway. When she chooses to search, a graphical user interface is created automatically by her agent so that she can provide some information regarding the request about finding this person. She chooses the concept of a family tree from the filtering criteria in the user interface and then she enters the name, surname and relationship to her. Ann was a bit surprised when she received the results. She saw that Lee Smith could not be found, but the semantic matcher returned a person Lee Burke as a possible match according to the family ontology. If the search of Ann's agent had been based on a traditional text search, then Ann would not have found her cousin. However, the use of a semantic matcher that works on ontological representation of the family tree enabled Ann to find her cousin under another surname. Ann was surprised as she did not know her cousin had just got married. Nevertheless, the photo of her cousin definitely confirmed that the system had returned the right person.

Within a semantic environment, such a system works using ontology graphs. Semantic matchers traversed the family ontology to find Ann's cousin. Traversing on the graph and inference based on this traversal can be accomplished by the applications of certain algorithms that basically deal with information extraction. These approaches and algorithms can find the required approximate node on an ontology tree. In fact, they succeed in finding the nearest nodes to the desired value. Details of this ontology traversal and semantic matching operation are beyond the scope of this paper. However, interested readers may refer to [5] and [25] for the general idea of semantic matching.

After finding the right person, according to Ann's request for communicating with Lee, in the next graphical user interface, she is then asked by her agent to choose the way she wants to communicate.

She prefers audio talking with her cousin. Her agent offers to contact Lee via a VoIP using GoogleTalk (GTalk) [109]. This selection is made upon the intelligence that checks both the user's input such as free talking and the user's information, which is gathered automatically using such as connection bandwidths, applications installed on mobile devices, and so on. In addition, Ann decides to send Lee a bunch of flowers, in view of her recent marriage. So, Ann asks her own agent to find the appropriate service. To do this, Ann enters the required information such as the desired flower's name, colour, amount, and the cost range, in order to limit the selections. The agent considers these parameters along with some other QoS parameters such as the distance of the flower shop from Lee's home address (which is automatically extracted), and the cost of the delivery service (as a composite service) accordingly. The result includes selecting a flower shop and a delivery service company that are altogether within the cost range of Ann's request.

When considering the system design of this case-study, the required result about communication and shopping services is gathered by the interaction between semantic services and agents within a MAS. Ann's request is held by a SWA inside a SS_FinderPlan instance according to the SEA_L's agent-SWS interaction viewpoint. The SS_FinderPlan instance basically finds the appropriate semantic web services that have already been registered with a SS_RegisterPlan, and returns the list of these services to a SSMatchmakerAgent to advise Ann's agent about candidate services. The discovery of the semantic services by the SS_FinderPlan is made semantically by traversing the Service Ontology. A graphical representation of an example of the Service Ontology structure is given in Figure 6 with ontology classes, and their subclass-superclass relationships.

Ann's agent (which is an instance of SEA_L SWA meta-entity) uses both the input given by Ann via the system interface and the information gathered by the agent while traversing the ontology graph. As a result for the communication part of the request, using ontologies, the agent decides that "talking" is a kind of communication that is online. The input requires a talk thus a chat or social network is unconsidered. According to the information, the bandwidth is insufficient for multimedia video talking. So, it goes on searching for a suitable medium within the VoIP services in regard to the sub-ontology, which is illustrated in Figure 7.

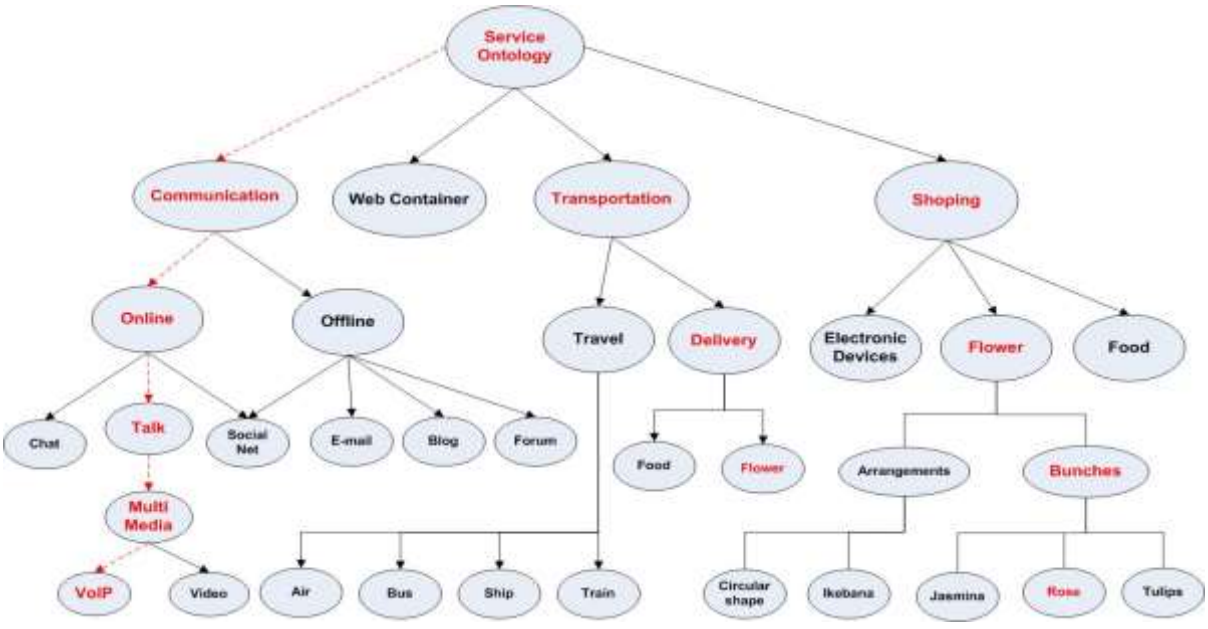

Figure 6: An example of the Service Ontology used in the expert finder system.

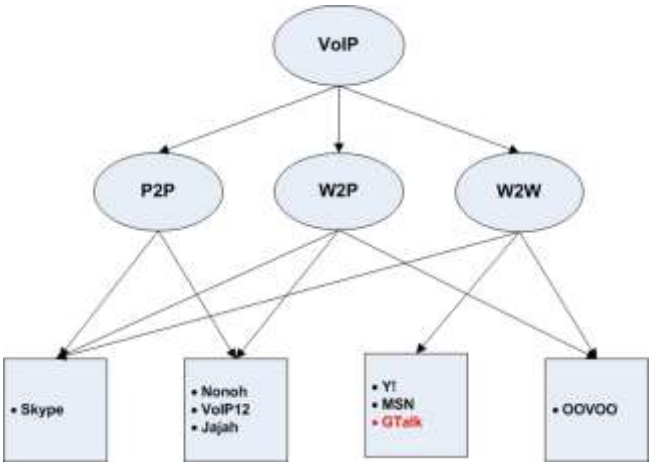

Figure 7: Sub-ontology for VoIP services.

Figure 7 shows a sample VoIP service ontology. Ann's agent communicates with Lee's agent to query whether Lee's phone has the required phone application. Since Ann and Lee's mobile phones have Internet connections, any kind of VoIP including phone to phone (P2P), web to phone (W2P), and web to web (W2W) is possible. However, when considering free calls, Nonoh [110] and Jajah [111] are excluded. In the same way, in regard to the applications installed on both mobile phones, Ann's agent's SS_FinderPlan provides the candidate services list containing GTalk and OOVOO [112] by interacting with the SSMatchmakerAgent. Then, the agent's SS_AgreementPlan chooses GTalk based on its sound quality.

In a similar manner, based on the human user request, the agent decides to buy the flowers from an e-flower shop called "Beautiful Flowers", and sends them via a delivery company called "Deliver Anywhere". The decision is made based on the QoS parameters that are both taken from Ann and extracted from the system automatically. For example, some of the flower shops are never considered in the result of the negotiation due to the type of flowers (see Figure 6), their colors (exact matching), and some others that are not selected due to the result of negotiation regarding the price. Furthermore, some of the delivery companies are omitted because of their delivery times, the delivering service being unavailable for small things such as flowers, or as a result of negotiation on the cost of the delivery.

When considering the scenario, we modelled the communication and shopping processes using SEA_L. We considered SWA agent instances and modelled these agents' interactions with semantic web services and web service internal components.

In order to provide a bigger picture of the case-study, the general idea of the system from the Agent-SWS interaction point of view is depicted in Figure 8. This diagram shows the main elements of the expert finding system and its relationships. It is based on the metamodel discussed in Section 4. When considering this big picture of the system, we could easily write the program model in SEA_L.

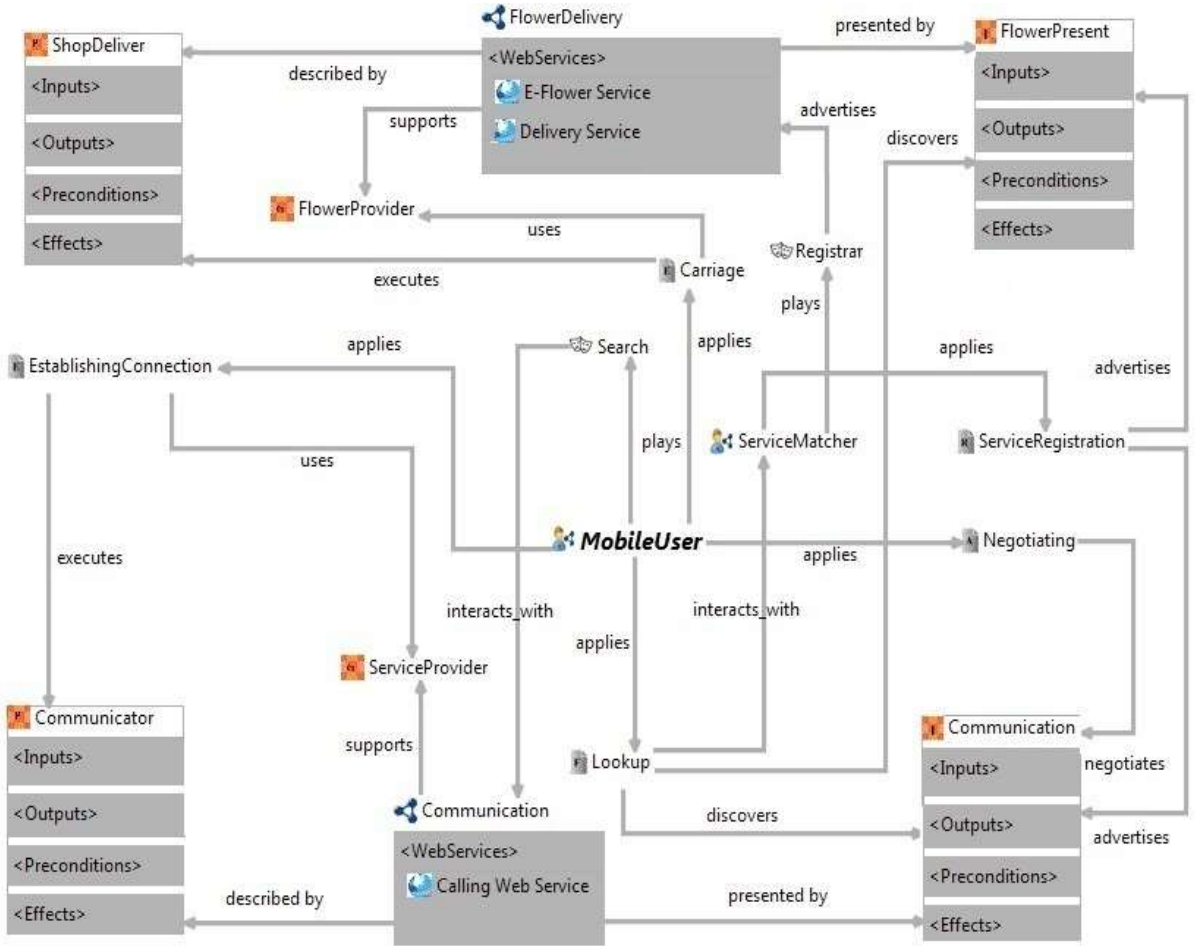

Figure 8: Graphical modelling for the Agent-SWS Interaction viewpoint of a multiagent Expert Finding system in the graphical syntax tool of SEA_L.

Listing 15 shows the instance program model from the Agent-SWS interaction viewpoint including semantic services and the required plan instances. Ann's agent, "MobileUser", is modelled with proper plan instances to find, make the agreement with and execute those services that are instances of the SS_FinderPlan, SS_AgreementPlan, and the SS_ExecutorPlan, respectively. The services are also modelled with the interaction between the semantic web service's internal components (such as Process, Grounding, and Interface), and the SWA's plans.

So, when considering Ann's communication request, her agent plays the "Search" role and applies its "Lookup" plan to find an appropriate "Communication" interface of "Communication" SWS. This plan realises the discovery via interacting with the "ServiceMatcher" that has registered the services by applying the "ServiceRegistration" plan. Next, the agent applies its "Negotiating" plan for negotiating with the already discovered services. This negotiation is done through the "Communication" interface of the SWS. Finally, if the result of the negotiation is positive, the agent applies the "EstablishingConnection" plan to call the "Calling Web Service" of the SWS by executing its "Communicator" process and using its "ServiceProvider" grounding with which the service is realised.

In a similar manner, due to Ann's request for flower shopping and delivery, her agent plays the "Search" role. It also applies the "Lookup" plan to find the "FlowerPresent" interface of the "FlowerDelivery" semantic web service, which is composed of the "E-Flower" and "Delivery" services. The "ServiceMatcher" and its "ServiceRegistration" plan help the "Lookup" plan in its goal. Then, the agent negotiates with the selected services' interfaces and if the result is successful, the agent applies its specific execution plan, "Carriage", to call the "FlowerDelivery" SWS, by executing its "Shop&Deliver" process, and using its "FlowerProvider" grounding.

```
01 AgentSWSinteraction ExpertFinding {
02
03    regPlan ServiceRegistration{
04       advertises FlowerPresent;
05       advertises Communication;
06       type "active";
07       description "worksWithMachmaker";
08       priority 6;
09    }
10    findPlan Lookup{
11       interactsWith ServiceMatcher;
12       discovers FlowerPresent;
13       discovers Communication;
14       type "active";
15       description "worksWithMatchmaker";
16       priority 5;
17    }
18    agreePlan Negotiating{
19       negotiates Communication;
20       type "waiting";
21       description "dependsOnTheResultOfFinderPlan";
22       priority 4;
23    }
24    execPlan Carriage{
25       uses FlowerProvider;
26       executes ShopDeliver;
27       type "waiting";
28       description "dependsOnTheResultOfAgreementPlan";
29       priority 4;
30    }
31    execPlan EstablishingConnection {
32       uses ServiceProvider;
33       executes Communicator;
34       type "waiting";
35       description "dependsOnTheResultOfAgreementPlan";
36       priority 4;
37    }
38    role Search {
39       interactsWith Communication;
40    }
41    regRole Registrar{
42       advertises FlowerDelivery;
43    }
44    interface Communication;
45    interface FlowerPresent;
46    process Communicator;
47    process ShopDeliver;
48    grounding ServiceProvider;
49    grounding FlowerProvider;
50
51    swa MobileUser {
52       applies Lookup;
53       applies Negotiating;
54       applies Carriage;
55       applies EstablishingConnection;
56       plays Search;
57       description "SystemAgent";
58       properties "AgentParameters";
59    }
60    sws FlowerDelivery {
61       describedBy ShopDeliver;
62       presentedBy FlowerPresent;
63       supportedBy FlowerProvider;
64    }
65    sws Communication {
66       describedBy Communicator;
67       presentedBy Communication;
68       supportedBy ServiceProvider;
69    }
70    sma ServiceMatcher{
71       appliesRegPlan ServiceRegistration;
72       playsRegRole Registrar;
73    }
74 }
```

Listing 15: The program model for the agent-based expert finding system

Finally, in order to develop the system the program model is controlled in SEA_L based on the constraints previously discussed in Section 4 and the architectural code is generated automatically for JADEX and OWL-S. A snapshot of the result for the semantics control is illustrated in Figure 9.

In Figure 9, firstly the result of semantics control is provided and next a report for the generated code is shown. In the semantic control part, we can see the results for the rules within the constraints including constraints 1 to 4 and their sub rules. For the code generation, the language presents the number of ADF files, Plan files and OWL files that are generated from the program model.

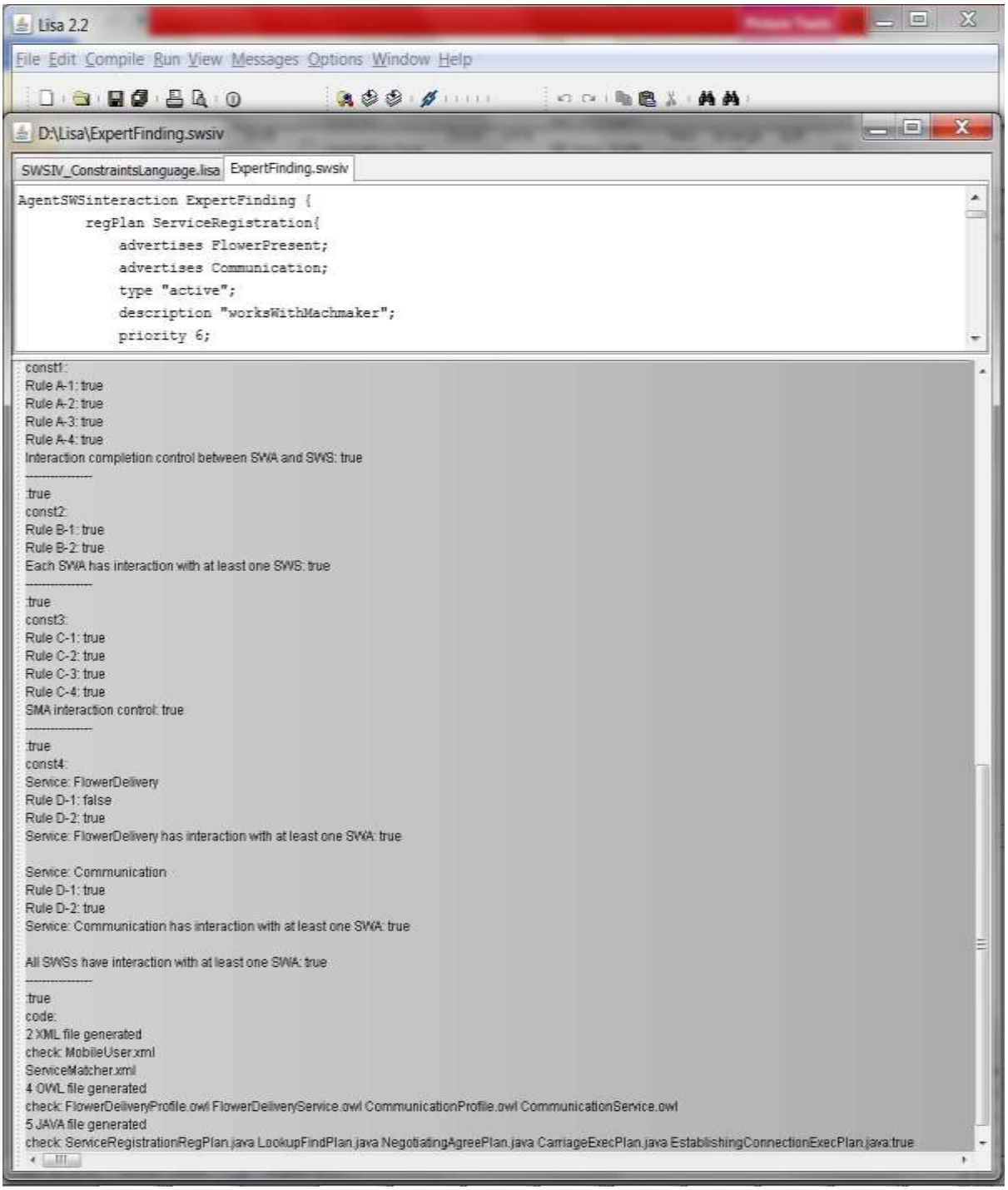

Figure 9: Snapshot of the constraints control for the program in SEA_L

As discussed in Section 4, modularity of specification was achieved with language extension [81] (see Listing 12, again). The language extension was easy, as concepts from language SWSIV_Constraints are orthogonal to the concepts from language SWSIV_Generation. As a result, we can also use the same DSL program (Listing 15) for original language SWSIV_Generation. Of course, the result of the program in Listing 15 on language SWSIV_Generation is different than in Figure 9 – the result contains only a summary of the generated code.

JADEX is one of the well-known and frequently used agent platforms in agent research and development studies. Its open-source Application Programming Interface (API) enables agent programmers to develop Belief-Desire-Intention (BDI) agents [98]. JADEX has an agent-oriented reasoning engine for coding rational agents with Extensible Markup Language (XML) and the Java programming language. The development of JADEX agents is based on a hybrid approach in which a declaration of static agent properties and the programming of executable agent plans take place. The declaration of static agent properties is given in files called Agent Definition Files (ADF). An ADF is written using XML and specifies the BDI model of the related agent. Moreover, agent plans are those executable components that are given in the Java program files. The JADEX reasoning engine starts the deliberation process by considering the goals requested by the agent. To this end, it adopts those goals stored in the database that contain all the adopted goals by the agent, called the agent's goal-base [15].

There is a base notation including three major terms regarding the JADEX architecture: beliefs, goals and plans. The beliefs are Java objects that represent the environmental facts that an agent has and are stored in a belief-base. The belief-base contains the facts that an agent possesses, in other words, it represents the knowledge of the world in which the agent is situated. Beliefs may change over the course of time within a dynamic environment, thus the belief-base needs to be updated in the long run. The goals in JADEX resemble the desires discussed in [98] to some extent. However, the goals are a vital part of JADEX rather than the events in traditional BDI systems. JADEX plans are Java classes that can be executed in order to achieve the goal of an agent. Plans have two parts: plan head and plan body. Furthermore, each agent has an ADF file, an XML-formatted file, in order to configure the agent's structure.

The result is an architectural code generated by the tool. For example, an ADF file is generated for the system agent, MobileUser. The generated ADF file is given in Figure 10. In this figure, all of the meta-elements and their attributes correspond to the related tags of JADEX ADF. An excerpt of the code for defining a "MobileUser" agent and some of its plans is given in Figure 10. All the attributes of the JADEX metamodel are excluded in the SEA_L metamodel. Therefore, lines 21 and 30 contain the default values of the corresponding attributes. In order to prevent repetition, only two of the Plan instances are given in Figure 10. The "Lookup" plan, Line 15 is the Finder Plan of the "MobileUser" agent for finding the "Communication" semantic web service. Similarly, the "Negotiating" plan in Line 24 is the Agreement Plan for negotiating with the "Communication" semantic web service.

By considering the generated artifacts, two types of files are generated for the Agent-SWS Interaction viewpoint. One group is for the agent part of the system (JADEX agents) that includes the ADFs of the JADEX agents and different plans for these agents (e.g. RegisterPlan, FinderPlan, etc.). Another group is for the Semantic Web Service part of the system (OWL-S components) that includes related files for service interface, process, and grounding.

As a result of compiling the program for this case-study, 2 ADF files (one for the SWA and one for the Matchmaker in the case model namely "MobileUser" and "ServiceMatcher" agents), 5 different plan files (EstablishingConnection, Lookup, Negotiating, ServiceRegistration, and Carriage plans for both agents), and 4 OWL documents are generated based on the designed model for the case-study, see code reporting part in Figure 9. Figure 10 shows an excerpt of the generated code for MobileUser agent in ADF format. As was discussed earlier, in this excerpt of the generated code, part of the header and the body of the generated ADF file are illustrated. As we can see, in the Plans section of the body, the definition of each plan of the agent is provided.

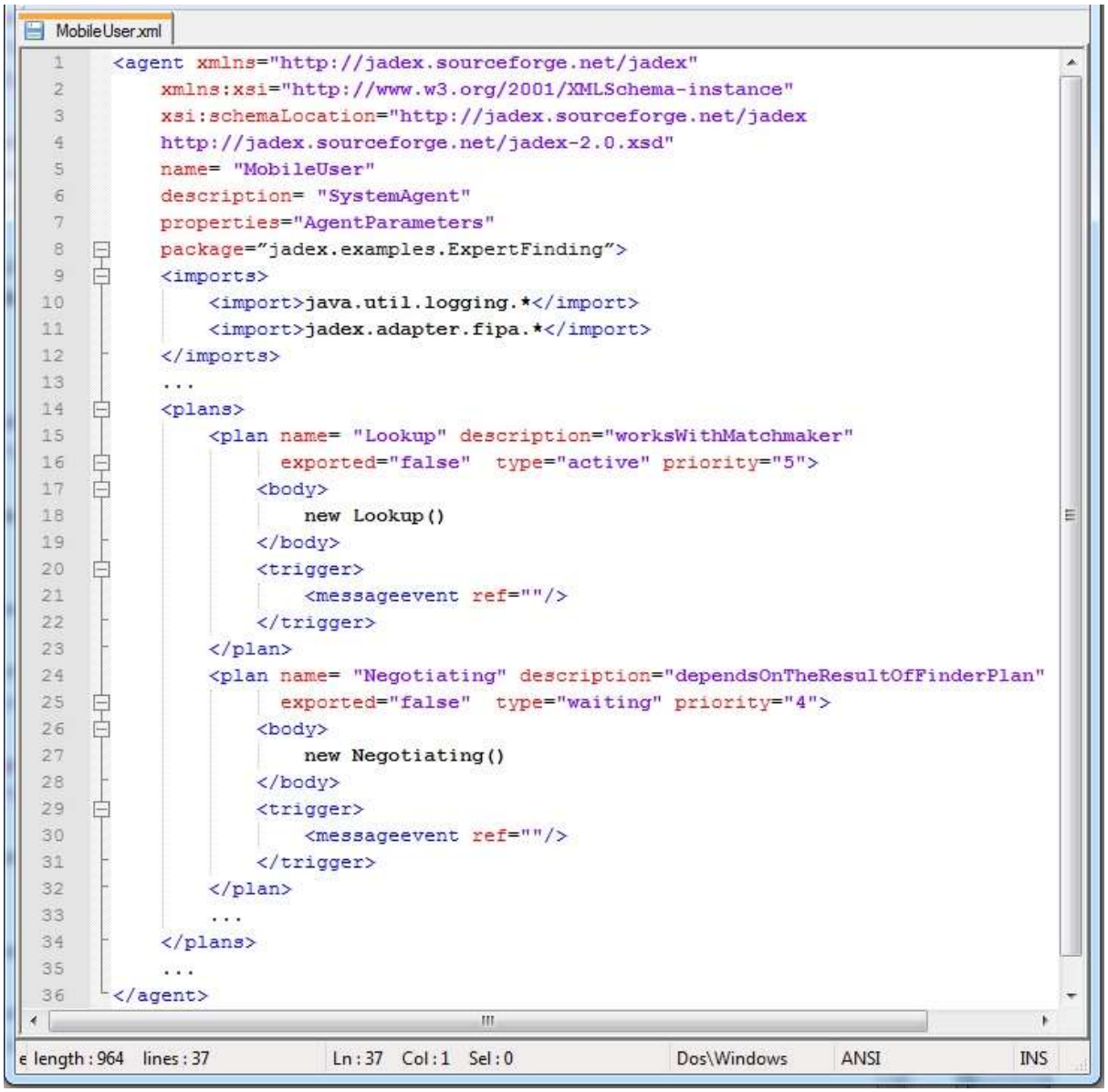

Figure 10: An excerpt from the generated ADF file for the Agent-SWS Interaction of MobileUser agent in the "Expert Finding" case-study

6. Evaluation and Discussion

Generally, the semantics of the programming languages deal with the meanings of the programs written in those languages. Formal semantics try to describe the meaning using mathematical approaches.

In our previous studies [13, 14] translational semantics were provided using transformations to other formal languages such as JADEX and OWL-S. That means we first defined entity mappings between SEA_L instance models and JADEX agent platform and again between SEA_L service entities and OWL-S profile, process and grounding components. Based on these mappings, rules were implemented for transformation from SEA_L to JADEX and OWL-S. At runtime, those rules were executed on SEA_L instance models and hence target instance models conforming to JADEX and OWL-S metamodels were automatically achieved. The operational or translational semantics are essential for a DSL or DSML in order to have executable results from these languages. Therefore, we also provided code generation from models using model-to-code transformations. However, the

translational semantics are not powerful enough to provide a formal way of describing the meaning of the software system; and we needed more formal approaches for describing the specifications of these languages.

For this purpose, we used Alloy in [16] and [17] for providing formal semantics and describing the logic of the instance models. Alloy is based on a set theory, predicate logic and first-order logic [74, 75] which have been used for analyses and the semantic controls in various studies [76, 77, 113]. It is suitable for representing the semantics of a DSL, as the meta elements can be represented by signatures (sets), and the relationships can be modelled as relationships between these sets. The constraints can be defined using multiplicity and the semantics rules can be stated as predicates in which we could have details of the rules described using predicate and first order logics. Hence, for each viewpoint of the language, we provided Alloy signatures, semantic constraints and rules ([16], [17]). Inside the environment of the Alloy tool, such a definition of formal semantics enabled us to examine the scope of analysis regarding instance models, property checking based on some assertions, and model finding within a specified scope. In addition, monitoring the change encountered in agent models at runtime was also possible. For instance, the effect of creating and adding a new semantic web agent to a running MAS environment can be dynamically examined and checked according to the semantics of the language.

However, Alloy defines the elements and relationships using its own meta elements, signatures, and the relationships between them; and defines the rules based on these elements. In other words, axiomatic semantics generally and Alloy language specifically, make no distinction between a phrase's meaning and the logical formulas that describe it; its meaning is exactly what can be proven about it in some logic. When considering the representation of DSL semantics with Alloy, the formal semantics and the operational semantics cannot be covered within the same model. Although Alloy has the power of demonstrating the dynamic semantics using Time and Order concepts, operational and axiomatic semantics are not integrated. As a result, in our case, the code generation and semantic controls with Alloy could not be attributed to a single language.

Hence, we can state that the required language semantics were not built into the language itself in those previous works [13, 14, 16, 17] and dynamic control of the semantics is directly bound within other environments. In our study, attribute grammars are used to implement part of the semantics of SEA_L. Attribute grammars can be mapped to denotational semantics where the target language is simply the original language enriched with attribute annotations. These attributes can be used to carry the desired semantics of the language.

Some of the constraint programming languages, e.g. OCL [114, 115, 116], can implement and integrate the constraint controls within the language and its translational semantics. In this way the translational semantics' and the constraints' controls are embedded within the language [117]. However, most researchers do not consider the representation of the semantics, some constraints in OCL, as a formal representation method. In addition, implementing these constraints in OCL is more difficult than implementing them within a general purpose language such as pure Java, in the case of attribute grammars in LISA.

However, this integration problem can be partially solved by other alternative formal approaches, .e.g. by using denotational semantics, more specifically by using the attribute grammars in our study. In the case of MDA, where we can use a grammar to represent the abstract and concrete syntax, attribute grammars can be used for resolving some of the formal semantics' shortages discussed above. In fact, the grammar part of this approach helps to build a language that can be used to provide operational semantics and execution items, and on the other hand, attributes added to the grammar can represent those semantics used to control the constraints. In this way, both the translational and the formal semantics are integrated within a single language.

In addition, applying a formal declarative description for the syntax and the semantics of the language has other advantages. For example, it is possible to automatically generate the language from the high level descriptions. It is also possible to do automatic model checking from these descriptions. On the

other hand, we can discover the problems with the language in the earlier stages of its development, during the design period. So, substantial effort and time are saved when testing and iterating the development process.

Moreover, the use of declarative description as the formal method for language description leads to higher modularity for the system. Not only does declarative description separate the syntax and semantics definitions but it also provides the required mechanism for dividing the grammar and semantics definitions in some modules. In other words, specification can be divided into different fragments, each dealing with different aspects. In this way the language is easy to extend, so the developer of the language can build the core of the language and provide new properties for the language by adding new modules; or even by inheriting from available languages and extending them within the new language (additional examples are shown in [81, 118, 119]). For example, in our case we developed the base language for SEA_L and then added the constraint controls in a separate module.

Finally, when using LISA as a tool to implement an attribute grammar based language, we could provide some additional advantages in addition to the above-mentioned advantages for declarative descriptions for formal semantics. LISA, as a compiler-compiler, is the provider of BNF Viewer, Syntax Tree Visualiser, Evaluator Tree Visualiser (Semantic Tree), Automata Visualizer, and DG (Dependency Graph) Viewer. When taking Semantic Tree into consideration (Figure 11), it can provide the tree for representing the way the meaning of a program is calculated based on the semantic rules during the compilation. This is similar to the Syntax tree that shows how the syntax of the program is evaluated based on the underlying syntax rules of the grammar. Figure 11 is the snapshot of a moment during the compilation and semantic computation. As the semantic for the whole case is very extensive, only a part of it is illustrated in Figure 11. These facilities helped us during the development of the language and the semantics. They can also help the user to have more syntax and semantic control of his or her program during the early stages.

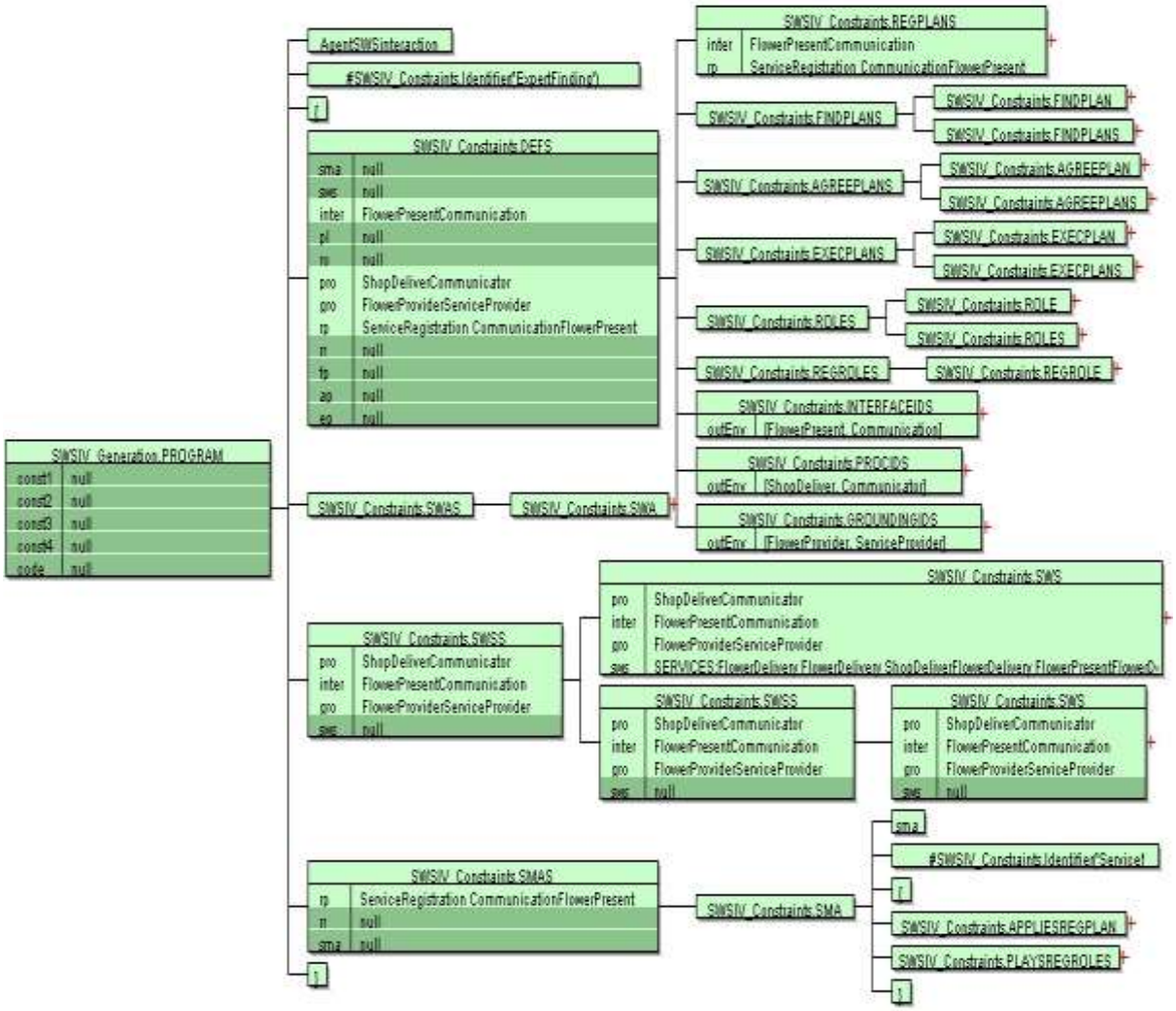

Figure 11: Partial Semantic Evaluation Tree for the case-study

7. Conclusion

This paper presented the implementation of a DSL for MAS called SEA_L including its translational semantics (code generation) and semantics controls for the language (constraints to check the programs) in an integrated way. The syntax and semantics of the language are defined formally using denotational semantics and the implementation is realized based on the formal semantics using attribute grammars. In addition, the application of the language is represented via a case-study. The implementation is realised in LISA tool.

As with the methodology, the domain analysis is performed with FODA. We created a metamodel from the feature diagram by adding cardinality and more relationship types. Using this metamodel, the abstract syntax of the language was defined as a context-free grammar. By adding the keywords of the language to the context-free grammar and using syntax sugaring, we provided the concrete syntax which is implemented in LISA. At the same time, we defined the semantics' domains and their equations in the form of denotational semantics based on those constraints that we had extracted from the metamodel using our experience within that domain. Then, as a modular approach, firstly, we developed the core language including the syntax with the ability of code generation. Next, the core

language was extended by transforming the semantic equations of the denotational semantics into semantics computations in LISA.

Applying denotational semantics provided us with accurate and precise definitions of the language. Also, by using the discussed methodology, we can generate the aimed language automatically from formal semantics. In addition, using attribute grammars adds modularity to our methodology. Not only the syntax (BNF) and the semantics (semantics computations) are separated but also the development of the problem domain can be realised step by step. In other words, the core language, e.g. syntax and code generation, can be implemented as the first step and a new extension of this language can be provided by adding semantic controls to the core language. This methodology makes evolutionary language development easier and may be a guide to the implementation of tools which support language extensions (e.g. Neverlang [120]).

Despite the advantages of this methodology, we experienced some challenges and difficulties. First of all, there is a learning-curve for beginners to become familiar with attribute grammars and denotational semantics. On the other hand, only the static properties can be checked using attribute grammars (i.e., only those which can be computed from a DSL program and not from its execution). For example, we would like to perform various checks depending on the availabilities of various web services, which can be computed only during DSL program run-times. Hence, in our next work, we will consider enriching the semantics of the language to include the dynamic semantics integrated within the language and with its operational semantics. The static semantics can help the user during the design time and these types of constraints are controlled during the compilation time. They can help the developer to determine the static problems of the program model. However, dynamic semantics can be checked during runtime in order to guide the user to observe the rules of the system as it runs. This kind of checking is more favourable within autonomous and dynamic systems such as MASs. Considering these rules during the design time can lead to the saving of substantial cost and effort during software development.

Acknowledgment

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

References

1. Wooldridge, M. and Jennings, N.R., *Intelligent Agents - Theory and Practice.* Knowledge Engineering Review, 1995. **10**(2): p. 115-152.
2. Berners-Lee, T., Hendler, J. and Lassila, O., *The semantic web.* Scientific American, 2001. **284**(5): p. 34-43.
3. Shadbolt, N., Hall, W. and Berners-Lee, T., *The Semantic Web revisited.* IEEE Intelligent Systems, 2006. **21**(3): p. 96-101.
4. Kardas, G., Goknil, A., Dikenelli, O. and Topaloglu, N.Y., *Model Driven Development of Semantic Web Enabled Multi-Agent Systems.* International Journal of Cooperative Information Systems, 2009. **18**(2): p. 261-308.
5. Sycara, K., Paolucci, M., Ankolekar, A. and Srinivasan, N., *Automated discovery, interaction and composition of Semantic Web services.* Web Semantics: Science, Services and Agents on the World Wide Web, 2003. **1**(1): p. 27-46.
6. Bădică, C., Ilie, S., Kamermans, M., Pavlin, G., Penders, A. and Scafeş, M., *Multi-agent systems, ontologies and negotiation for dynamic service composition in multi-organizational environmental*

management. Software Agents, Agent Systems and Their Applications, NATO Science for Peace and Security Series - D: Information and Communication Security, 2012. **32**(12): p. 286-306.
7. van Deursen, A., Klint, P. and Visser, J., *Domain-specific languages: An annotated bibliography.* ACM SIGPLAN Notices, 2000. **35**(6): p. 26-36.
8. Mernik, M., Heering, J. and Sloane, A., *When and How to Develop Domain-specific Languages.* ACM Computing Surveys, 2005. **37**(4): p. 316-344.
9. Varanda-Pereira, M.J., Mernik, M., da-Cruz, D. and Henriques, P.R., *Program Comprehension for Domain-Specific Languages.* Computer Science and Information Systems, 2008. **5**(2): p. 1-17.
10. Fowler, M., *Domain-specific Languages*. 2010: Addison-Wesley Professional.
11. Sprinkle, J., Mernik, M., Tolvanen, J.-P. and Spinellis, D., *Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer?* IEEE Software, 2009. **26**(4): p. 15-18.
12. Challenger, M., Getir, S., Demirkol, S. and Kardas, G., *A Domain Specific Metamodel for Semantic Web enabled Multi-agent Systems.* Advanced Information Systems Engineering Workshops, Lecture Notes in Business Information Processing, 2011. **83**: p. 177-186.
13. Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G. and Mernik, M., *SEA_L: A Domain-specific Language for Semantic Web enabled Multi-agent Systems*, in *The Second Workshop on Model Driven Approaches in System Development (MDASD), Federated Conference on Computer Science and Information Systems (Fedcsis)*. 2012: Wrocław-Poland. p. 1373-1380.
14. Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G. and Mernik, M., *A DSL for the Development of Software Agents working within a Semantic Web Environment.* Computer Science and Information Systems, 2013. **10**(4): p. 1525-1556.
15. Pokahr, A., Braubach, L. and Lamersdorf, W., *Jadex: A BDI Reasoning Engine.*, in *Multi-agent Programming Languages, Platforms and Applications*. 2005, Springer. p. 149-174.
16. Getir, S., Challenger, M., Demirkol, S. and Kardas, G., *The Semantics of the Interaction between Agents and Web Services on the Semantic Web*, in *7th IEEE International Workshop on Engineering Semantic Agent Systems (ESAS), IEEE 36th Annual Computer Software and Applications Conference (COMPSAC)*. 2012, IEEE: Izmir-Turkey. p. 619-624.
17. Getir, S., Challenger, M. and Kardas, G., *The Formal Semantics of a Domain-specific Modeling Language for Semantic Web enabled Multi-agent Systems.* International Journal of Cooperative Information Systems, 2014. **23**(3): p. 1-53.
18. Paulson, L., *A Semantics-directed Compiler Generator*, in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1982, ACM. p. 224-233.
19. Knuth, D.E., *Semantics of Context-Free Languages.* Mathematical Systems Theory, 1968. **2**(2): p. 127-145.
20. Paakki, J., *Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation.* ACM Computing Surveys (CSUR), 1995. **27**(2): p. 196-255.
21. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B. and Karsai, G., *Challenges and Directions in Formalizing the Semantics of Modeling Languages.* Computer Science and Information Systems, 2011. **8**(2): p. 225-253.
22. Mernik, M., Lenič, M., Avdičaušević, E. and Žumer, V., *LISA: An interactive environment for programming language development.* R. N. Horspool (ed.): Compiler Construction. Lecture Notes in Computer Science, 2002. **2304**: p. 1-4.
23. Hendler, J., *Agents and the Semantic Web.* IEEE Intelligent Systems, 2001. **16**(2): p. 30-37.
24. Paolucci, M., Kawamura, T., Payne, T.R. and Sycara, K., *Semantic Matching of Web Services Capabilities*, in *The Semantic Web — ISWC 2002, Lecture Notes in Computer Science*. 2002, Springer. p. 333-347.
25. Li, L. and Horrocks, I., *A software framework for matchmaking based on semantic web technology.* International Journal of Electronic Commerce, 2004. **8**(4): p. 39-60.
26. Gibbins, N., Harris, S. and Shadbolt, N., *Agent-based semantic web services*, in *Proceedings of the 12th International World Wide Web Conference*. 2003: Budapest, Hungary. p. 710–717.
27. Zou, Y., *Agent-Based Services for the Semantic Web*, in *Faculty of the Graduate School*. 2004, University of Maryland.
28. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B. and Payne, T., *OWL-S: Semantic markup for web services*, in *W3C member submission*. 2004.
29. W3C. *Web Ontology Language (OWL).* 2004 [cited 2015 August]; Available from: http://www.w3.org/TR/owl-features/.
30. Burstein, M., Bussler, C., Zaremba, M., Finin, T., Huhns, M.N., Paolucci, M., Sheth, A.P. and Williams, S., *A semantic web services architecture.* IEEE Internet Computing, 2005. **9**(5): p. 72-81.
31. Gümüs, Ö., Gürcan, Ö., Kardas, G., Ekinci, E.E. and Dikenelli, O., *Engineering an MAS platform for semantic service integration based on the SWSA*, in *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, Lecture Notes in Computer Science*. 2007, Springer. p. 85-94.

32. Gürcan, Ö., Kardas, G., Gümüs, Ö., Ekinci, E.E. and Dikenelli, O., *An MAS infrastructure for implementing SWSA based semantic services*, in *Service-Oriented Computing: Agents, Semantics, and Engineering, Lecture Notes in Computer Science*. 2007. p. 118-131.
33. Varga, L.Z., Hajnal, A. and Werner, Z., *An agent based approach for migrating web services to semantic web services*, in *Artificial Intelligence: Methodology, Systems, and Applications, Lecture Notes in Artificial Intelligence*, C. Bussler and Fensel, D., Editors. 2004. p. 371-380.
34. Talantikite, H.N., Aissani, D. and Boudjlida, N., *Semantic annotations for web services discovery and composition.* Computer Standards & Interfaces, 2009. **31**(6): p. 1108-1117.
35. Paulraj, D., Swamynathan, S. and Madhaiyan, M., *Process Model Ontology-Based Matchmaking of Semantic Web Services.* International Journal of Cooperative Information Systems, 2011. **20**(4): p. 357-370.
36. Kumar, S., *Agent-Based Semantic Web Service Composition*. Springer Briefs in Electrical and Computer Engineering. 2012: Springer. 67.
37. Ferber, J. and Gutknecht, O., *A Meta-model for the Analysis and Design of Organizations in Multi-agent Systems*, in *The 3rd International Conference on Multi Agent Systems*. 1998: Paris-France. p. 128-135.
38. Odell, J., Nodine, M. and Levy, R., *A Metamodel for Agents, Roles, and Groups.* Agent-Oriented Software Engineering V, Lecture Notes in Computer Science, 2005. **3382**: p. 78-92.
39. Hahn, C., Madrigal-Mora, C. and Fischer, K., *A Platform-Independent Metamodel for Multiagent Systems.* Autonomous Agents and Multi-Agent Systems, 2009. **18**(2): p. 239-266.
40. Beydoun, G., Low, G.C., Henderson-Sellers, B., Mouratidis, H., Gomez-Sanz, J.J., Pavon, J. and Gonzalez-Perez, C., *FAML: A Generic Metamodel for MAS Development.* IEEE Transactions on Software Engineering, 2009. **35**(6): p. 841-863.
41. Bernon, C., Cossentino, M., Gleizes, M.-P., Turci, P. and Zambonelli, F., *A Study of Some Multi-Agent Meta-Models.* Agent-Oriented Software Engineering V, Lecture Notes in Computer Science, 2005. **3382**: p. 62-77.
42. Bernon, C., Gleizes, M.-P., Peyruqueou, S. and Picard, G., *ADELFE: A Methodology for Adaptive Multi-Agent Systems Engineering*, in *Engineering Societies in the Agents World III, Lecture Notes in Artificial Intelligence*. 2003, Springer. p. 70-81.
43. Zambonelli, F., Jennings, N.R. and Wooldridge, M., *Developing Multiagent Systems: The Gaia Methodology.* ACM Transactions on Software Engineering and Methodology, 2003. **12**(3): p. 317-370.
44. Cossentino, M. and Potts, C. *A CASE Tool Supported Methodology for the Design of Multi-agent Systems*. in *International Conference on Software Engineering Research and Practice (SERP'02)*. 2002. Las Vegas.
45. Molesini, A., Denti, E. and Omicini, A., *MAS Meta-Models on Test: UML vs. OPM in the SODA Case Study*, in *Multi-Agent Systems and Applications IV, Lecture Notes in Artificial Intelligence*, M. Pechoucek, Petta, P. and Varga, L.Z., Editors. 2005. p. 163-172.
46. Omicini, A., *SODA: Societies and Infrastructures in the Analysis and Design of Agent-based Systems.* First international workshop on Agent-Oriented Software Engineering, Lecture Notes in Computer Science, 2001. **1957**: p. 185-193.
47. Depke, R., Heckel, R. and Kuster, J.M., *Agent-Oriented Modeling with Graph Transformation.* First International Workshop on Agent-Oriented Software Engineering, Lecture Notes in Computer Science, 2001. **1957**: p. 105-119.
48. Bauer, B., Muller, J.P. and Odell, J., *Agent UML: A Formalism for Specifying Multi-agent Software Systems.* International Journal of Software Engineering and Knowledge Engineering, 2001. **11**(3): p. 207-230.
49. Huget, M.-P. *Modeling Languages for Multiagent Systems*. in *Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005)*. 2005. Utrecht, the Netherlands.
50. Cervenka, R., Trencansky, I., Calisti, M. and Greenwood, D., *AML: Agent Modeling Language - Toward Industry-Grade Agent-based Modeling*, in *Agent-Oriented Software Engineering V, Lecture Notes in Computer Science*, J. Odell, Giorgini, P. and Muller, J.P., Editors. 2005. p. 31-46.
51. García-Magariño, I., *Towards the integration of the agent-oriented modeling diversity with a powertype-based language.* Computer Standards & Interfaces, 2014. **36**: p. 941-952.
52. Kulesza, U., Garcia, A., Lucena, C. and Alencar, P., *A Generative Approach for Multi-agent System Development*, in *Software Engineering for Multi-Agent Systems III, Lecture Notes in Computer Science*. 2005, Springer. p. 52-69.
53. Rougemaille, S., Migeon, F., Maurel, C. and Gleizes, M.-P., *Model Driven Engineering for Designing Adaptive Multi-Agents Systems.* Engineering Societies in the Agents World VIII, Lecture Notes in Artificial Intelligence, 2008. **4995**: p. 318-332.
54. Ciobanu, G. and Juravle, C., *Flexible Software Architecture and Language for Mobile Agents.* Concurrency and Computation-Practice & Experience, 2012. **24**(6): p. 559-571.

55. Gascuena, J.M., Navarro, E. and Fernandez-Caballero, A., *Model-Driven Engineering Techniques for the Development of Multi-agent Systems.* Engineering Applications of Artificial Intelligence, 2012. **25**(1): p. 159-173.
56. Padgham, L. and Winikoff, M., *Developing Intelligent Agent Systems: A Practical Guide*. Vol. 13. 2004: John Wiley and Sons.
57. AOS. *Agent Oriented Software Pty., Ltd., Inc., JACK Intelligent Agents*. 2001 [cited 2015 August]; Available from: http://www.aosgrp.com/products/jack/.
58. Fuentes-Fernandez, R., Garcia-Magarino, I., Gomez-Rodriguez, A.M. and Gonzalez-Moreno, J.C., *A Technique for Defining Agent-Oriented Engineering Processes with Tool Support.* Engineering Applications of Artificial Intelligence, 2010. **23**(3): p. 432-444.
59. da Silva, A.R., *Model-driven engineering: a survey supported by the unified conceptual model.* Computer Languages, Systems & Structures, 2015. **43**: p. 139–155.
60. OMG. *Software & System Process Engineering Metamodel Specification Version 2.0, formal/2008-04-01*. 2008 [cited 2015 August]; Available from: http://www.omg.org/spec/SPEM/2.0/.
61. Pavón, J., Gómez-Sanz, J.J. and Fuentes-Fernandez, R., *The INGENIAS Methodology and Tools*, in *Agent-Oriented Methodologies, Article IX*, B. Henderson-Sellers and Giorgini, P., Editors. 2005, Idea Group Publishing. p. 236-276.
62. Hahn, C., *A Domain Specific Modeling Language for Multiagent Systems*, in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 1*. 2008, International Foundation for Autonomous Agents and Multiagent Systems: Estoril, Portugal. p. 233-240.
63. Warwas, S. and Hahn, C., *The Contrete Syntax of the Platform Independent Modeling Language for Multiagent Systems*, in *The 7th International Conference on Autonomous Agents and Multi Agent Systems (AAMAS), Agent-based Technologies and applications for enterprise interOPerability (ATOP 2008)*. 2008: Estoril, Portugal.
64. Hahn, C., Nesbigall, S., Warwas, S., Zinnikus, I., Fischer, K. and Klusch, M., *Integration of Multiagent Systems and Semantic Web Services on a Platform Independent Level*, in *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2008)*. 2008, IEEE Computer Society: Sydney, Australia. p. 200-206.
65. OMG. *Model Driven Architecture (MDA) Specification*. 2003 [cited 2015 August]; Available from: http://www.omg.org/mda/.
66. Hahn, C. and Fischer, K., *The Formal Semantics of the Domain Specific Modeling Language for Multiagent Systems*, in *Agent-Oriented Software Engineering IX, Lecture Notes in Computer Science*, M. Luck and GomezSanz, J.J., Editors. 2009. p. 145-158.
67. Smith, G., *The Object-Z specification language*. 2000: Kluwer Academic Publication.
68. Boudiaf, N., Mokhati, F. and Badri, M., *Supporting formal verification of DIMA multi-agents models: towards framework based on MAUDE model checking.* International Journal of Software Engineering and Knowledge Engineering, 2008. **18**(7): p. 853-875.
69. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J. and Quesada, J.F., *Maude: specification and programming in rewriting logic.* Theoretical Computer Science, 2002. **285**(2): p. 187-243.
70. Hilaire, V., Koukam, A., Gruer, P. and Müller, J.-P., *Formal specification and prototyping of multi-agent systems*, in *Engineering Societies in the Agents World, Lecture Notes in Artificial Intelligence*. 2000, Springer. p. 114-127.
71. Brandão, A.A., Alencar, P. and de Lucena, C.J., *AgentZ: extending Object-Z for multi-agent systems specification*, in *Agent-Oriented Information Systems II, Lecture Notes in Artificial Intelligence*. 2005, Springer. p. 125-139.
72. Dastani, M., Hindriks, K.V. and Meyer, J.-J., *Specification and verification of multi-agent systems*. 2010: Springer Science & Business Media. 405.
73. El Fallah-Seghrouchni, A., Gomez-Sanz, J.J. and Singh, M.P., *Formal methods in agent-oriented software engineering*, in *Agent-Oriented Software Engineering X, Lecture Notes in Computer Science*. 2011, Springer. p. 213-228.
74. Jackson, D., *Alloy: a lightweight object modelling notation.* ACM Transactions on Software Engineering and Methodology, 2002. **11**(2): p. 256-290.
75. Jackson, D., *Software Abstractions: Logic, Language, and Analysis*. 2012, Cambridge, MA: MIT Press. 376.
76. Podorozhny, R., Khurshid, S., Perry, D. and Zhang, X., *Verification of multi-agent negotiations using the Alloy analyzer*, in *Integrated Formal Methods, Lecture Notes in Computer Science*, J. Davies and Gibbons, J., Editors. 2007, Springer. p. 501-517.

77. Haesevoets, R., Weyns, D., Torres, M.H.C., Helleboogh, A., Holvoet, T. and Joosen, W., *A Middleware Model in Alloy for Supply Chain-wide Agent Interactions*, in *Agent-Oriented Software Engineering XI, Lecture Notes in Computer Science*. 2011, Springer. p. 189-204.
78. Kardas, G., Demirezen, Z. and Challenger, M., *Towards a DSML for semantic web enabled multi-agent systems*, in *Proceedings of the International Workshop on Formalization of Modeling Languages (FML), held in conjunction with the 24th European Conference on Object-Oriented Programming (ECOOP 2010)* 2010, ACM: Maribor, Slovenia. p. 1-5.
79. Challenger, M., Demirkol, S., Getir, S., Mernik, M., Kardas, G. and Kosar, T., *On the use of a domain-specific modeling language in the development of multiagent systems.* Engineering Applications of Artificial Intelligence, 2014. **28**: p. 111-141.
80. Sarasa-Cabezuelo, A. and Sierra, J.-L., *The grammatical approach: A syntax-directed declarative specification method for XML processing tasks.* Computer Standards & Interfaces, 2013. **35**(1): p. 114-131.
81. Mernik, M., *An object-oriented approach to language compositions for software language engineering.* Journal of Systems and Software, 2013. **86**(9): p. 2451-2464.
82. Ceh, I., Crepinšek, M., Kosar, T. and Mernik, M., *Ontology driven development of domain-specific languages.* Computer Science and Information Systems, 2011. **8**(2): p. 317-342.
83. Frakes, W., Prieto-Diaz, R. and Fox, C., *DARE: Domain analysis and reuse environment.* Annals of Software Engineering, 1998. **5**(1): p. 125-141.
84. Weiss, D.M. and Lai, C.T.R., *Software product-line engineering: a family-based software development process*. 1999: Addison-Wesley Professional.
85. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S., *Feature-oriented domain analysis (FODA) feasibility study*. 1990: Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. p. 148.
86. Falbo, R.d.A., Guizzardi, G. and Duarte, K.C., *An Ontological Approach to Domain Engineering*, in *14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*. 2002. p. 351-358.
87. Štuikys, V. and Damaševičius, R., *Measuring complexity of domain models represented by feature diagrams.* Information Technology and Control, 2009. **38**(3): p. 179-187.
88. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenič, M., Gray, J. and Wu, H., *Automatic generation of language-based tools using the LISA system.* IEE Proceedings-Software, 2005. **152**(2): p. 54-69.
89. de Jonge, M. and Visser, J., *Grammars as feature diagrams*, in *Proceedings of the Workshop on Generative Programming at the 7th International Conference on Software Reuse*. 2002. p. 23-24.
90. Naur, P., Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J. and Perlis, A.J., *Revised report on the algorithmic language Algol 60.* Communications of the ACM, 1963. **6**(1): p. 1-17.
91. Porubän, J., Forgáč, M., Sabo, M. and Běhálek, M., *Annotation based parser generator.* Computer Science and Information Systems, 2010. **7**(2): p. 291-307.
92. Eclipse. *Xtext Language*. 2006 [cited 2015 August]; Available from: http://www.eclipse.org/Xtext/.
93. Alanen, M. and Porres, I., *A relation between context-free grammars and meta object facility metamodels*. 2003: TUCS Technical Report No 606, Turku Centre for Computer Science, Åbo Akademi University.
94. Mernik, M., Crepinsek, M., Kosar, T., Rebernak, D. and Zumer, V., *Grammar-based systems: definition and examples.* Informatica, 2004. **28**(3): p. 245-255.
95. Kosar, T., Martinez Lopez, P.E., Barrientos, P.A. and Mernik, M., *A preliminary study on various implementation approaches of domain-specific language.* Information and Software Technology, 2008. **50**(5): p. 390-405.
96. Attali, I. and Parigot, D., *Integrating natural semantics and attribute grammars: the Minotaur System*, in *INRIA Technical Report RR-2339*. 1994.
97. Fister , I.J., Mernik, M., Fister, I. and Hrnčič, D., *Implementation of EasyTime formal semantics using a LISA compiler generator.* Computer Science and Information Systems, 2012. **9**(3): p. 1019-1044.
98. Rao, A.S. and Georgeff, M.P., *BDI Agents: From Theory to Practice*, in *The 1st International Conference on Multi-Agent Systems (ICMAS-95)*. 1995: San Francisco. p. 312-319.
99. W3C. *OWL-S: Semantic markup for web services*. 2004 [cited 2015 August]; Available from: http://www.w3.org/Submission/OWL-S/.
100. Hsu, I.-C., Tzeng, Y.K. and Huang, D.-C., *OWL-L: An OWL-based language for Web resources links.* Computer Standards & Interfaces, 2009. **31**(4): p. 846-855.
101. Fowler, M., *UML distilled: a brief guide to the standard object modeling language*. 2003: Addison-Wesley Professional. 208.
102. Macikenas, E. and Makunaite, R., *Applying agent in business evaluation systems.* Information Technology and Control, 2008. **37**(2): p. 101-105.
103. Shih, D.-H., Huang, S.-Y. and Yen, D.C., *A new reverse auction agent system for m-commerce using mobile agents.* Computer Standards & Interfaces, 2005. **27**(4): p. 383-395.

104. Pesovic, D., Vidakovic, M., Ivanovic, M., Budimac, Z. and Vidakovic, J., *Usage of Agents in Document Management.* Computer Science and Information Systems, 2011. **8**(1): p. 193-210.
105. Teket, K.D., Sayit, M. and Kardas, G., *Software agents for peer-to-peer video streaming.* IET Software, 2014. **8**(4): p. 184-192.
106. Yildirim, O. and Kardas, G., *A multi-agent system for minimizing energy costs in cement production.* Computers in Industry, 2014. **65**(7): p. 1076-1084.
107. Demirkol, S., Getir, S., Challenger, M. and Kardas, G., *Development of an Agent based E-barter System*, in *International Symposium on Innovations in Intelligent Systems and Applications (INISTA 2011)*. 2011: Istanbul, Turkey. p. 193-198.
108. Kardas, G., Challenger, M., Yildirim, S. and Yamuc, A., *Design and implementation of a multiagent stock trading system.* Software-Practice & Experience, 2012. **42**(10): p. 1247-1273.
109. Google Inc., *Google Talk computer to computer voice and video chat software*. 2005 [cited 2015 August]; Available from: http://www.google.com/talk/
110. Nonoh. *Nonoh VoIP calls*. 2007 [cited 2015 August]; Available from: http://www.nonoh.net/
111. Jajah. *Jajah Communication Solutions*. 2006 [cited 2015 August]; Available from: http://www.jajah.com
112. Oovoo. *Oovoo Free video chats*. 2007 [cited 2015 August]; Available from: http://www.oovoo.com
113. Toahchoodee, M. and Ray, I., *Using Alloy to analyse a spatio-temporal access control model supporting delegation.* IET Information Security, 2009. **3**(3): p. 75-113.
114. Warmer, J. and Kleppe, A., *The Object Constraint Language: Getting Your Models Ready for MDA*. Second ed. 2003, USA: Pearson Education.
115. OMG. *Object Constraint Language (OCL)*. 2012 [cited 2015 August]; Available from: http://www.omg.org/spec/OCL/2.3.1/.
116. Cadavid, J.J., Combemale, B. and Baudry, B., *An analysis of metamodeling practices for MOF and OCL.* Computer Languages, Systems & Structures, 2015. **41**: p. 42–65.
117. Saritas, H.B. and Kardas, G., *A model driven architecture for the development of smart card software.* Computer Languages, Systems & Structures, 2014. **40**(2): p. 53-72.
118. Mernik, M. and Žumer, V., *Incremental programming language development.* Computer Languages, Systems & Structures, 2005. **31**(1): p. 1-16.
119. Fister, I., Jr., Kosar, T., Fister, I. and Mernik, M., *Easytime++: A Case Study Of Incremental Domain-Specific Language Development.* Information Technology and Control, Kaunas, Technologija, 2013. **42**(1): p. 77-85.
120. Vacchi, E. and Cazzola, W., *Neverlang: A framework for feature-oriented language development.* Computer Languages, Systems & Structures, 2015. **43**: p. 1-40.