International Journal of Cooperative Information Systems
Vol. 18, No. 2 (2009) 261–308
© World Scientific Publishing Company

MODEL DRIVEN DEVELOPMENT OF SEMANTIC WEB ENABLED MULTI-AGENT SYSTEMS

GEYLANI KARDAS*
International Computer Institute, Ege University
Bornova, 35100 Izmir, Turkey
geylani.kardas@ege.edu.tr

ARDA GOKNIL
Software Engineering Group, University of Twente
Enschede, 7500 AE, The Netherlands
a.goknil@ewi.utwente.nl

OGUZ DIKENELLI[†] and N. YASEMIN TOPALOGLU[‡]
Department of Computer Engineering, Ege University
Bornova, 35100 Izmir, Turkey
[†]*oguz.dikenelli@ege.edu.tr*
[‡]*yasemin.topaloglu@ege.edu.tr*

Semantic Web evolution brought a new vision into agent research. The interpretation of this second generation web will be realized by autonomous computational entities, called agents, to handle the semantic content on behalf of their human users. Surely, Semantic Web environment has specific architectural entities and a different semantic which must be considered to model a Multi-agent System (MAS) within this environment. Hence, in this study, we introduce a MAS development process which supports the Semantic Web environment. Our approach is based on Model Driven Development (MDD) which aims to change the focus of software development from code to models. We first define an architecture for Semantic Web enabled MASs and then provide a MAS metamodel which consists of the first class meta-entities derived from this architecture. We also define a model transformation process for MDD of such MASs. We present a complete transformation process in which the source and the target metamodels, entity mappings between models and the implementation of the transformation for two different real MAS frameworks by using a well-known model transformation language are all included. In addition to the model-to-model transformation, the implementation of the model-to-code transformation is given as the last step of the system development process. The evaluation of the proposed development process by considering its use within the scope of a real commercial software project is also discussed.

Keywords: Multi-agent System; model driven development; semantic web; model transformation.

*Corresponding author. Addr: Ege University International Computer Institute, Ege University Campus, 35100, Bornova Izmir Turkey. Tel.: +90-232-3423232-103; Fax.: +90-232-3887230.

1. Introduction

Software agents and Multi-agent Systems (MAS) are recognized as both useful abstractions and effective technologies for the modeling and building of complex distributed systems. In its most fundamental artificial intelligence definition, an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.[1]

Software agents are considered as autonomous software components which are capable of acting on behalf of their human users in order to perform a group of defined tasks. On the other hand, the study of MAS focuses on systems in which many intelligent software agents interact with each other. Their interactions can be either cooperative or selfish.[2] In other words, the agents can share a common goal or they can pursue their own interests (as in the free market economy).

MAS researchers develop communication languages, interaction protocols, and agent architectures that facilitate the development of MASs. They propose new methodologies (e.g. Gaia,[3] Tropos,[4] MaSE[5] and SODA[6]) and tools for agent-oriented software development because characteristics and challenges of agent-oriented software engineering (AOSE) stretch the limits of current software engineering methodologies as mentioned in Ref. 7. Since AOSE is distinct from object-orientation when agent *goal*, *role*, *context* and *messages* are considered as first class entities, various studies in MAS community[8,9] define agent metamodels that include these entities and their relations. Also, in recent past, Modeling Technical Committee (TC) of Foundation for Intelligent Physical Agents (FIPA)[a] had an effort to develop a notation to express relationships between agents, agent roles and agent groups in a MAS.[10] However, we believe that a significant deficiency exists in those noteworthy agent modeling and methodology studies when we consider their support on the Semantic Web[11] technology and its constructs.

Semantic Web evolution brought a new vision into agent research. This second generation Web aims to improve World Wide Web (WWW) such that web page contents are interpreted with ontologies. It is apparent that the interpretation in question will be realized by autonomous computational entities — so agents — to handle the semantic content on behalf of their human users. Surely, Semantic Web environment has specific architectural entities and a different semantic which must be considered to model a MAS within this environment. Therefore, agent modeling techniques and MAS development frameworks should support this new environment by defining new meta-entities and architectural components. Hence, in this study, we introduce a MAS development process which supports the Semantic Web environment during MAS development according to this vision. The developed MASs will be Semantic Web enabled; that means software agents are planned to collect Web content from diverse sources, process the information and exchange

[a]FIPA was accepted by the IEEE as its eleventh standards committee on June 2005 and became IEEE FIPA. We keep naming as FIPA throughout the paper since we reference this organization's previous works.

the results on behalf of their human users. Autonomous agents can also evaluate semantic data within these MASs and collaborate with semantically defined entities such as semantic web services by using content languages.

Our approach is based on Model Driven Development (MDD) which aims to change the focus of software development from code to models.[12] Design and implementation of MAS may become more complex and hard to implement when new requirements and interactions for new agent environments such as Semantic Web are considered. We believe that MDD would provide an infrastructure that simplifies the development of such MASs. To work in a higher abstraction level is of critical importance for the development of MASs since it is almost impossible to observe code level details of the MASs due to their internal complexity, distributedness and openness. Hence, such MDD application increases the abstraction level in MAS development.

MDD requires (1) definition of domain metamodels, (2) definition of system models conforming to those metamodels, (3) definition and application of model transformations between those models according to the entity mappings and (4) definition and application of model to text transformation for the automatic generation of software codes from output models. Our study presents a complete software development process that meets all of these MDD requirements to develop the Semantic Web enabled MASs.

The business domain in our study is agent systems working on the Semantic Web environment. So, we need a formal agent metamodel for such systems. The metamodel in question should be platform independent. That means, it should define first class entities and their relations for a Semantic Web enabled MAS which are all abstract from the physical agent deployment environments. However, current agent metamodels in literature have been generally used for only presenting concepts of their dedicated methodologies as mentioned in Ref. 13 and they support neither semantic representations of agent capabilities nor interactions between software agents and semantic web services. In our study, we first introduce a platform independent MAS metamodel in which agent organizations, agents, their roles and Semantic web extensions of the MAS are all modeled with their associations. Meta-entities and their relations of this MAS metamodel are derived from a conceptual MAS architecture which is also discussed in this paper.

Definition of such a model is a prerequisite to conduct a model transformation which is the key activity in MDD.[14] Hence in this paper, we also present a model transformation process in which a model conforming to the above MAS metamodel is transformed into models conforming to metamodels of two different real agent platforms. The designed Semantic Web enabled MAS can be implemented on these real platforms by applying the corresponding transformations. To do this, we first define source and target metamodels for each transformation and then provide mappings between entities of these source and target metamodels to derive transformation rules and constraints. We realize the whole transformation by using a pretty known model transformation language. Finally, automatic code generation from output MAS models assists MAS developers in implementing their systems on various agent platforms.

The paper is organized as follows: In Sec. 2, a brief explanation for MDD and its application on MAS development are given. In Sec. 3, we discuss a new conceptual software architecture for Semantic Web enabled MASs to identify new constructs in addition to the constructs of a traditional MAS. A metamodel for such MASs is discussed in Sec. 4. This metamodel is our source metamodel for the model transformations. Section 5 introduces metamodel of two target MAS development platforms and discusses the entity mappings required for the model transformations. Application of the real model transformations with appropriate transformation language and tool utilization is given in Sec. 6. Section 7 discusses the last step of the proposed development process: model to text transformation which provides automatic generation of software codes from MAS models. Section 8 includes the evaluation of the development process within the scope of a commercial software development project. Section 9 covers the related work on MDD of agent systems. Finally, conclusions and future work are given in Sec. 10.

2. Model Driven Engineering for MAS Development

MDD approach considers the models as the main artifacts of software development. We use *Model Driven Architecture* (*MDA*)[15] which is one of the realizations of MDD to support the relations between platform independent and various platform dependent agent artifacts to develop semantic web agents.

MDA defines several model transformations which are based on the Meta-Object Facility (MOF)[16] framework. In MDA, models are first-class artifacts, integrated into the development process through the chain of transformations to coded application. In order to enable this, MDA requires models to be expressed in a MOF-based language. This guarantees that the models can be stored in a MOF-compliant repository, parsed and transformed by MOF-compliant tools, and rendered into XML Metadata Interchange (XMI) for transport over a network.[15]

Transformations defined by MDA are structured in a three-layered architecture: *the Computation Independent Model* (*CIM*), *the Platform Independent Model* (*PIM*), and *the Platform Specific Model* (*PSM*). A CIM is a view of a system from the computation independent viewpoint.[15] For instance, the CIM for agent systems does not have any information about agents and semantic web services. However, according to the system requirements, entities in the CIM can later be used in order to derive agents and semantic web services in the PIM of the Semantic Web enabled MAS.

The PIM focuses on the operation of a system while it still hides the details necessary for the implementation of the system in a particular platform. The PIM specifies a degree of platform independency to be suitable for use with a number of different platforms of similar type.[15] In our perspective, the PIM of a Semantic Web enabled MAS should define the main entities and interactions which do not belong to a specific agent framework.

On the other hand the PSM includes details of the platform implementation. The platform independent entities in the PIM of the Semantic Web enabled MAS are transformed to the PSM of an implemented Semantic Web enabled agent framework like SEAGENT.[17] The flexible part of this approach is that the PIM enables to generate different PSMs of Semantic Web enabled agent frameworks automatically.

The development process and the MOF based transformations between the MDA models are given in Fig. 1. The transitions from the CIM to the PSM step by step are specified in the model transformation definitions based on the MOF metamodel.

In the transformation pattern depicted in Fig. 1, a source model *sm* is transformed into a target model *tgm*. The transformation is driven by a transformation definition written in a transformation language.[18–21] The source and target models and the transformation definition conform to their metamodels *SMM*, *TgMM* and *TMM* respectively. The transformations defined from CIM to PIM ($T1$) and PIM to various PSMs ($T2$ and $T2'$) use the metamodels of CIM, PIM and PSMs for source and target metamodels in corresponding transformation patterns. After completing model-to-model transformations according to that pattern, the next and final step is to provide system implementation by realizing model-to-code transformations ($T3$ and $T3'$) for the specific platforms.

Considering the study presented in this paper, we provide a PIM for Semantic Web enabled MASs and PSMs of various MAS development frameworks and we define model to model and model to code transformation steps which employ the related metamodels. Hence, a MAS developer builds the model of the desired MAS conforming to the defined PIM and provides this model as the input of the MDA process. After automatic execution of the related transformations the developer obtains the implementation of the designed system on different MAS environments. The developers do not deal with the transformation pattern or the internal execution of the transformations.

3. A Software Architecture for Semantic Web Enabled MASs

The first class entities of a Semantic Web enabled MAS, which constitute the application models, must be defined in order to apply model driven approaches for the development of these systems. We believe that these entities can be derived from a conceptual architecture of Semantic Web enabled MASs. For this reason, we introduce a MAS architecture in which autonomous agents can also evaluate the semantic data and collaborate with the semantically defined entities such as semantic web services by using content languages. The preliminary version of the architecture appeared in Ref. 22.

Our proposed conceptual architecture for the Semantic Web enabled MASs is given in Fig. 2. The architecture has a layered style and the relation among layers is *allowed-to-use* relation described in Ref. 23. For two layers having this relation, any module in the first is allowed to use any module in the second. The direction of the

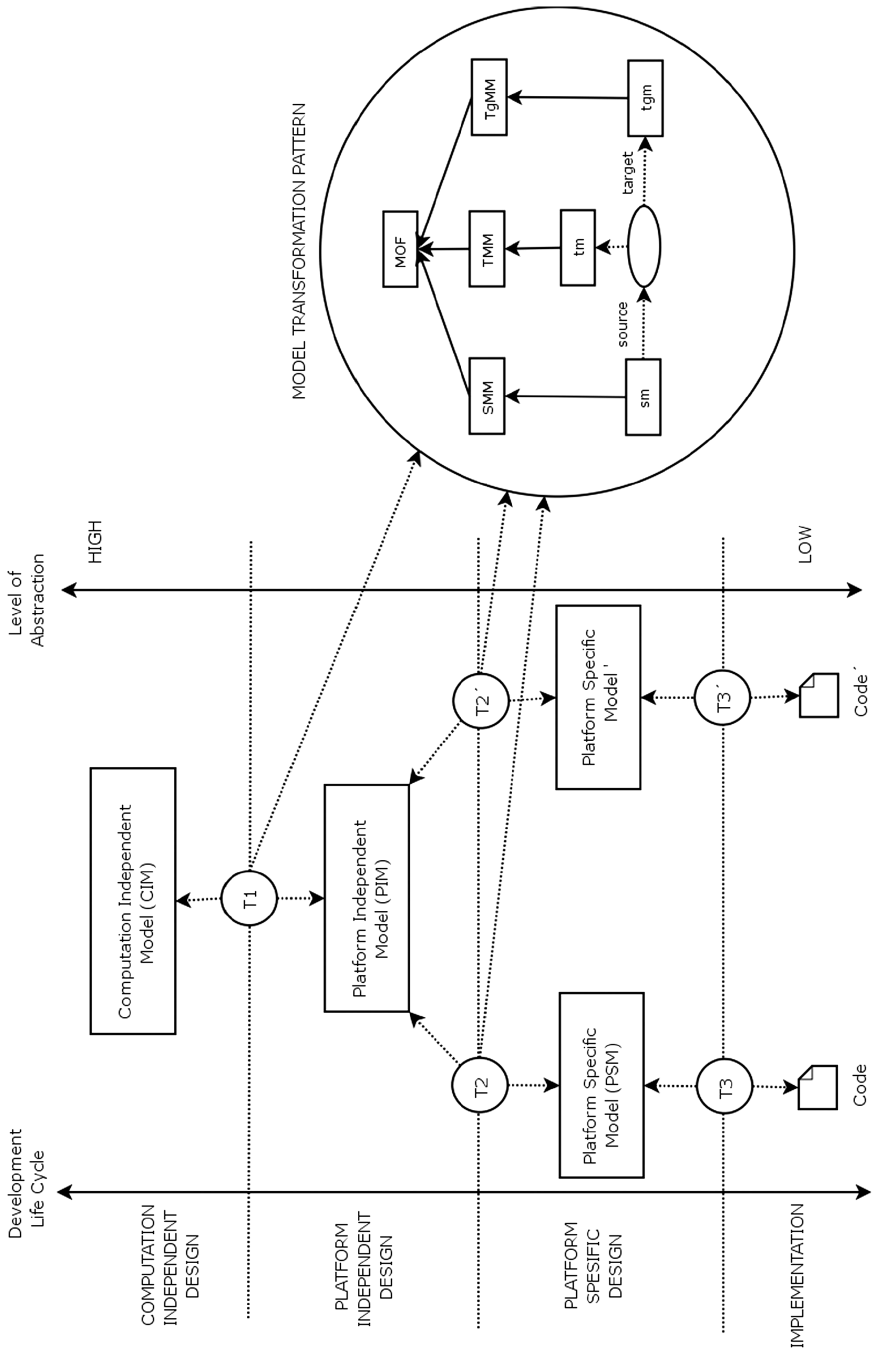

Development Life Cycle
Level of Abstraction
HIGH
LOW
COMPUTATION INDEPENDENT DESIGN
PLATFORM INDEPENDENT DESIGN
PLATFORM SPESIFIC DESIGN
IMPLEMENTATION
Computation Independent Model (CIM)
T1
Platform Independent Model (PIM)
T2
Platform Specific Model (PSM)
T3
Code
T2´
Platform Specific Model '
T3´
Code´
MODEL TRANSFORMATION PATTERN
MOF
SMM
TMM
TgMM
sm
tm
tgm
source
target

Fig. 1. Development process and model transformation mechanism in MDA.

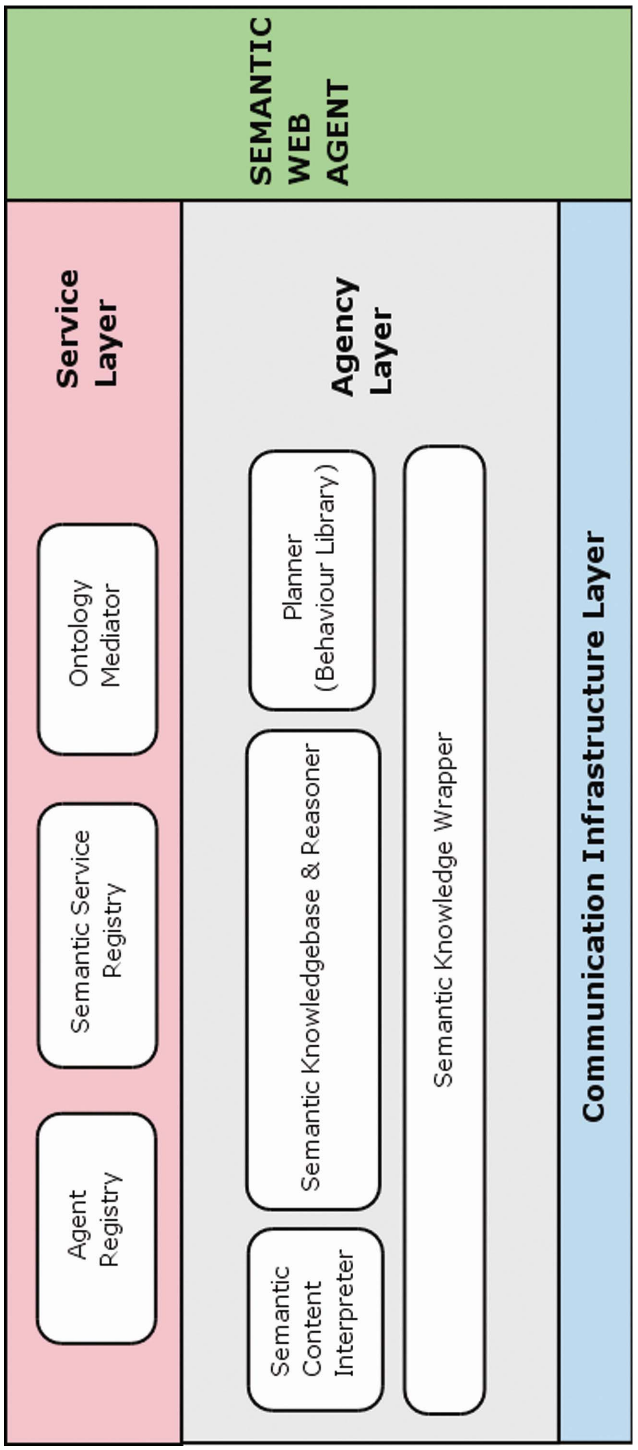

SEMANTIC WEB AGENT
Service Layer
Agent Registry
Semantic Service Registry
Ontology Mediator
Agency Layer
Semantic Content Interpreter
Semantic Knowledgebase & Reasoner
Planner (Behaviour Library)
Semantic Knowledge Wrapper
Communication Infrastructure Layer

Fig. 2. The conceptual architecture for Semantic Web enabled MASs.

relation is downward. That means only a higher layer is allowed to use facilities of a lower layer in our architecture. Reverse usage is not allowed. On the other hand, the architecture does not include any *layer bridging*.[23] Hence a higher layer uses only modules of the next-lower layer. We use a notation including layers with a sidecar and the *allowed-to-use* relation is denoted by geometric adjacency in the figure.

Due to its layered construction, the architecture defines three layers in its main stack: *Service Layer*, *Agency Layer* and *Communication Infrastructure Layer*. *Semantic Web Agent* is the sidecar component which uses modules of the above mentioned architecture layers. A group of system agents provides services defined in the Service Layer. Every agent in the system has an inner agent architecture described in the Agency Layer and they communicate with each other according to the protocols defined in the Communication Infrastructure. The layers of the architecture are discussed with their internal modules in the following subsections.

3.1. *Service layer*

In the Service Layer, services (and/or roles) of semantic web agents inside the platform are described. All services in the Service Layer use the capability of the Agency Layer. Besides domain specific agent services, yellow page and mediator services should also be provided.

Agent Registry is a system facilitator in which capabilities of agents are semantically defined and advertised for other platform members. During their task executions, platform agents may need services provided by the other agents. Hence, they query on this facilitator to determine relevant agents for interaction. No matter it is FIPA-compliant[24] or not, a traditional MAS owns one or more registries which provides yellow page services for system's agents to look for proper agent services. Certainly, the aforementioned registries are not simple structures and mostly implemented as directory services and served by some platform specific agents. For example there is a mandatory agent called *directory facilitator* (*DF*) in FIPA abstract architecture specification on which agent services are registered.[24] When an agent looks for a specific agent service, it gathers supplier data (agent's name, address, etc.) of the service from the DF and then it begins to communicate with this service provider agent to complete its task. However, capability matching becomes complex and has to be redefined when we take into consideration of MASs on Semantic Web environment. In case of agent service discovery, in such systems, we should define semantic matching criteria of service capabilities and design registration mechanisms (directory services) of agent service specifications according to those criteria. That makes matching of requested and advertised services more efficient by not only taking into consideration of identical service matching: New capability matching will determine type and degree of relation between two services (requested and advertised) *semantically*. Hence, the conceptual architecture includes an agent registry to provide capability matching on agent services. In Ref. 25, one such application of the semantic capability matching on FIPA-compliant agent systems is discussed.

On the other hand, agents of the platform may also need to interact with Semantic Web Services which are web services with semantic interface to be discovered and executed. Web services are commonly-known software systems identified by a Uniform Resource Identifier (URI) whose public interface and bindings are defined and described by XML.[26] By discovering these definitions, other software systems use related Web service. However, current Web service infrastructure focuses on only syntactic interoperability. Two popular standards are SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language) which supports XSD-based data structures. Again in Ref. 26, it is mentioned that such a definition does not allow for both semantic interoperability and automatic composition of Web services. To support such interoperability and composition, capabilities of web services are defined in service ontologies such as OWL-S[27] and WSMO.[28] In our approach, those service capabilities should also be advertised on proper registries to provide dynamic discovery and execution of the services by agents. Hence, we define a conceptual entity called *Semantic Service Registry* in the proposed architecture. This registry can also be modeled as a service matchmaker in which semantic interfaces of the platform's semantic web services are advertised to be discovered by the agents. Considering OWL-S services, agents may query on this facilitator by sending its requested semantic service OWL-S profile to the facilitator. The facilitator (or matchmaker) performs a semantic capability matching between the given request and advertised profiles and informs the agent about suitable services. Then the agent may interact with those services to complete its task. Engagement and invocation of the semantic web service is also performed according to service's semantic protocol definitions. Candidate implementations for capability matching of the semantic web services are discussed in Refs. 29 and 30.

A Semantic Web enabled agent interacts with agents within the different organization(s) and semantic web services may use knowledge sources handled by the different knowledgebase(s) and/or peer system(s). In such environment, it is obvious that, there exist more than one ontology and different entities may use different ontologies. So, there should be another architectural service in which translation and mapping of different ontologies are performed. We call this service as *Ontology Mediator* and it may be provided by one or more agents within the MAS. An Ontology Mediator may also behave as a central repository for the domain ontologies used within the platform and provide basic ontology management functionality such as ontology deployment, ontology updating and querying. Through the usage of the ontology translation support, any agent of the platform may communicate with MAS and/or services outside the platform even if they use different ontologies.

3.2. *Agency layer*

The middle layer of the architecture is the Agency which includes inner structural components of Semantic Web enabled agents. Every agent in the system has a *Semantic Knowledgebase* which stores the agent's local ontologies. Those ontologies are used by the agent during its interaction with other platform agents and semantic web services. Evaluation of the ontologies and primitive inference are realized by the *Reasoner*.

Semantic Knowledge Wrapper within the Agency provides utilization of above mentioned ontologies by upper-level Agency components. For example, during its task execution, the agent may need object (or any other programmatic) representation of a specific ontology individual. Or the content interpreter requests a query on one of the ontologies to reason about something. To meet up such requirements, the Semantic Knowledge Wrapper of the agent may form graph representations of the related ontologies within the runtime environment of the Agency. An example for this kind of wrapper use within an agent internal architecture is based on the JENA[31] framework which is discussed in Ref. 32.

The *Planner* of the Agency Layer includes necessary reusable plans with their related behavior libraries. The reusable agent plans are composed of tasks which are executed according to the agent's intentions. The planner is based on the reactive planning paradigm e.g. HTN (Hierarchical Task Network) planning framework presented in Ref. 33. In reactive planning a library of general pre-defined (may be defined at compile time) plans is provided to agent and the agent performs one or more of these plans in response to its perceptions of the environment.[34]

Semantic Content Interpreter module uses the logical foundation of semantic web, ontology and knowledge interpretation. During its communications, the agent receives messages from other agents or semantic services. It needs to evaluate the received message content to control its semantic validity and interpret the content according to its beliefs and intentions. Necessary content validity and interpretation takes place in this module.

3.3. *Communication infrastructure layer*

The bottom layer of the architecture is responsible of abstracting the architecture's communication infrastructure implementation. For example, it may be an implementation of FIPA's Agent Communication and Agent Message Transport specifications[24] to handle agent messaging. Hence, the layer transfers any content (including semantic knowledge) by using FIPA Agent Communication Language (ACL) and transport infrastructure. Physical communication may take place via well-known HTTP-IIOP (Internet Inter-ORB Protocol). However, the content language within the message infrastructure is crucial.

4. A Metamodel for Semantic Web Enabled MASs

In this section, we introduce an agent metamodel superstructure to define elements and their relationships of a Semantic Web enabled MAS depending on the preceding software architecture. Current metamodel is composed of a core model discussed in Ref. 22 and this core's supplements in order to support the use of model in MDD as a PIM. The final metamodel is an extension of FIPA Modeling TC's Agent Class Superstructure Metamodel.[10] On the other hand, we provide new constructs (e.g. Semantic Web constructs) for our metamodel by extending Unified Modeling Language (UML) 2.0 Superstructure[35] and Ontology UML Profile which is defined in Ref. 36.

FIPA Modeling TC's Agent Class Superstructure Metamodel (ACSM) has a specification which is based on — and extends — UML superstructure. It proposes a metamodel which defines the user-level constructs required to model agents, agent roles and agent groups. Although ACSM is in a preliminary phase, we believe that it neatly presents an appropriate superstructure specification that defines the user-level constructs required to model agents, their roles and their groups. By extending this superstructure we do not need to re-define basic entities of the agent domain. In fact, representing the MAS structure with these main meta-entities is not new and formerly proposed in AALAADIN MAS metamodel[37] but not as formal as FIPA Modeling TC's work. Also, ACSM models assignment of agents to roles by taking into consideration of group context. Hence, extending ACSM clarifies relatively blurred associations between related concepts in our core metamodel by appropriate inclusion of ACSM's Agent Role Assignment entity. More information about ACSM can be found in Refs. 10 and 38.

Due to space limitations, our Semantic Web enabled MAS metamodel is pictured in here by dividing it into two parts (Figs. 3(a) and 3(b) respectively). In order to provide traceability, the rest of each entity relation denoted with a bold letter at the end of the Fig. 3(a) is again denoted with the same bold letter in Fig. 3(b). The depicted metamodel is the enhanced version of the model introduced in Ref. 39 with new entities and revised associations and used as PIM during our MDD process.

A *Semantic Web Agent* is an autonomous entity which is capable of interaction with both other agents and semantic web services. It is a special form of the ACSM's Agent class. *Semantic Organization* is a composition of *Semantic Web Agent*s which is constituted according to organizational roles of those agents. A Semantic Organization is implemented as a composition of Semantic Web Agents. It also includes organizational roles and those roles are played by its agents. Taking into consideration those organizational roles, a Semantic Web Agent may be a member of different Semantic Organizations. That means, one agent may play more than one *Role* and one Role may be played by many Semantic Web Agents within the Semantic Organization context.

On the other hand, Semantic Web Organization is defined as a direct extension of the ACSM's Group Structure in the model. A Semantic Web Organization may or may not behave as a Semantic Web Agent in overall manner. Hence, it is not defined neither as Agentified nor Non-Agentified Group.

The Role concept in the metamodel is an extension of Agent Role Classifier due to its classification for roles the semantic agents are capable of playing at a given time. This conforms to the Agent — Agent Role Classifier association defined in

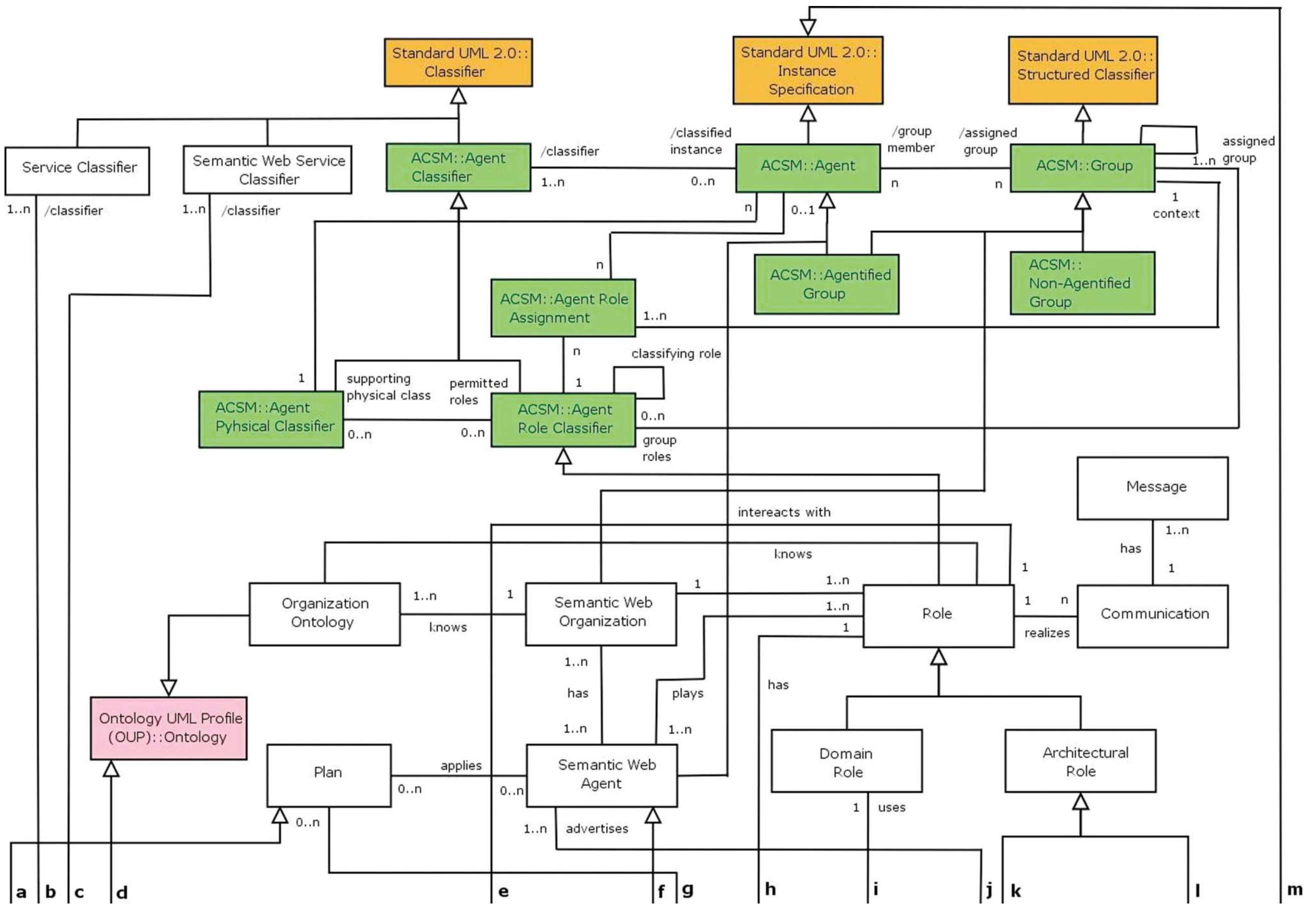


Fig. 3. (a) The first part of the metamodel for Semantic Web enabled MASs which extends FIPA Modeling TC's ACSM, UML 2.0 Superstructure and Ontology UML Profile.

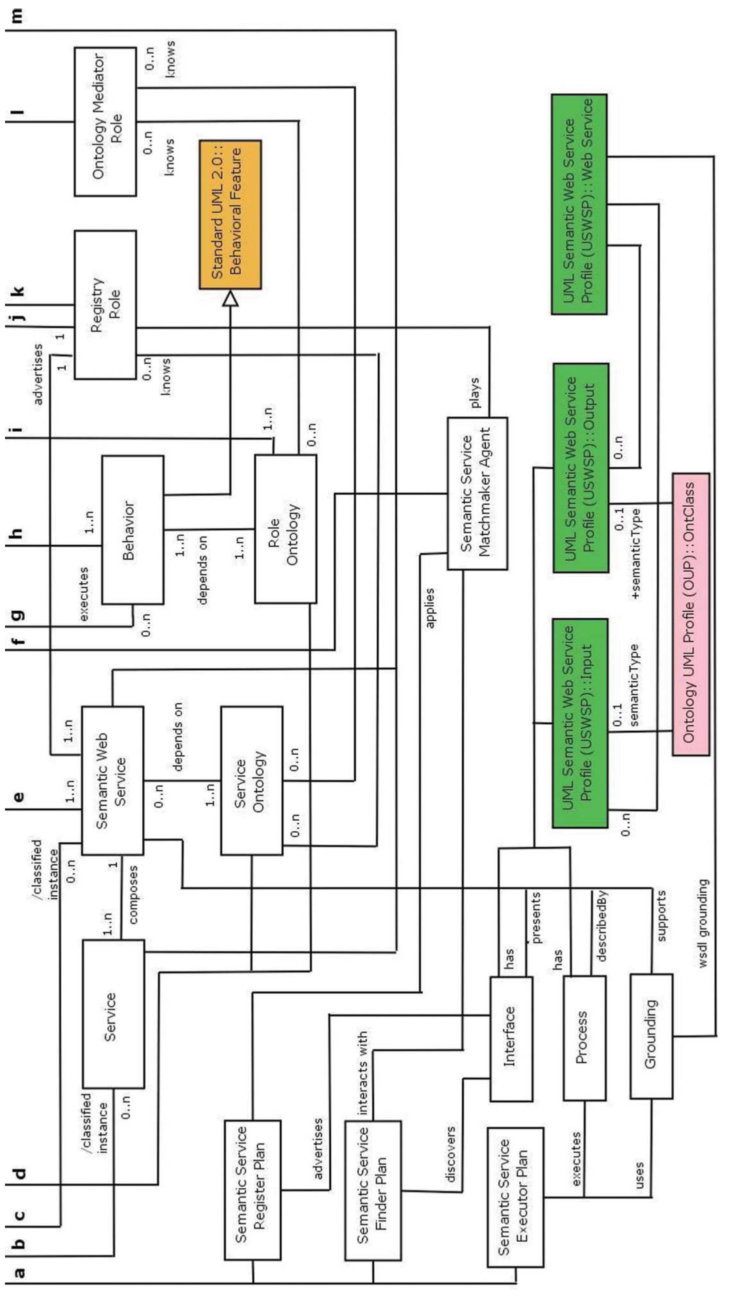

a
b
c
d
e
f
g
h
i
j
k
l
m
Ontology Mediator Role
Registry Role
Standard UML 2.0:: Behavioral Feature
Behavior
Role Ontology
Semantic Web Service
Service Ontology
Service
Semantic Service Matchmaker Agent
Semantic Service Register Plan
Semantic Service Finder Plan
Semantic Service Executor Plan
Interface
Process
Grounding
UML Semantic Web Service Profile (USWSP)::Web Service
UML Semantic Web Service Profile (USWSP)::Output
UML Semantic Web Service Profile (USWSP)::Input
Ontology UML Profile (OUP)::OntClass
advertises
knows
plays
executes
depends on
applies
/classified instance
composes
interacts with
discovers
has
presents
describedBy
supports
wsdl grounding
uses
+semanticType
semanticType
1
1..n
0..n
0..1

Fig. 3. (b) The second part of the metamodel for Semantic Web enabled MASs which extends FIPA Modeling TC's ACSM, UML 2.0 Superstructure and Ontology UML Profile.

ACSM[38]: *Semantic Web Agents can be associated with more than one Role* (*which is also an Agent Role Classifier*) *at the same point in time* (*multiple classification*) *and can change roles over time* (*dynamic classification*). The ACSM extensions provide clarification of the relations between Semantic Web Agent, Role and Semantic Web Organization in our model by employing ACSM's Agent Role Assignment ternary association between Agent, Agent Role Classifier and Group.

The Role is a general model entity and it should be specialized in the metamodel according to task definitions of architectural and domain-based roles: An *Architectural Role* defines a mandatory Semantic Web enabled MAS role that should be played at least one agent inside the platform regardless of the organization context whereas a *Domain Role* completely depends on the requirements and task definitions of a specific Semantic Organization created for a specific business domain.

Some of the organization agents must play architectural roles to provide services defined in the Service Layer of the conceptual architecture for other agents. Hence two specialization of the Architectural Role are also defined in the metamodel: *Registry Role* and *Ontology Mediator Role*. Registry Roles are played by one or more Semantic Web Agents which store capability advertisements of Semantic Web Agents or Semantic Web Services. Ontology Mediator Role in the metamodel defines basic ontology management functionality that should be supported by ontology mediator agents as discussed in the previous section.

One Role is composed of one or more *Behavior*s. Task definitions and related task execution processes of the Semantic Web agents are modeled inside the Behavior entities. Proper arrangement of those behaviors constitutes agent roles. The Behavior entity is defined in the metamodel as a UML 2.0 Behavioral Feature because it refers to a dynamic feature of a Semantic Web Agent (e.g. an agent task which realizes agent interaction with other agents).

According to played roles, agents inevitably communicate with other agents to perform desired tasks. Each *Communication* entity defines a specific interaction between two agents of the platform which takes place in proper to predefined agent interaction protocol. One Communication is composed of one or more *Message*s whose content can be expressed in a Resource Description Framework (RDF)[40] based semantic content language.

A *Semantic Web Service* represents any service (except agent services) whose capabilities and interactions are semantically described within a Semantic Web enabled MAS. A Semantic Web Service composes one or more *Service* entities. Each service may be a web service or another service with predefined invocation protocol in real-life implementation. But they should have a semantic web interface to be used by autonomous agents of the platform. It should be noted that association between the semantic web agents and the services is provided over the agent role entities in the metamodel. Because agents interact with semantic web services, depending on their roles defined inside the organization.

Like agents, semantic web services have also capabilities and features which could not be just based on the object-oriented paradigm. Hence, we define new

Classifiers and their related Instance Specifications in the metamodel by extending proper UML 2.0 Superstructure meta-entities in order to encapsulate the semantic web entities. We have applied classifier — classified instance association between *Semantic Web Service Classifier* and *Semantic Web Service*. Same is valid for *Ontology Classifier* — *Ontology* and *Service Classifier* — *Service* relationships.

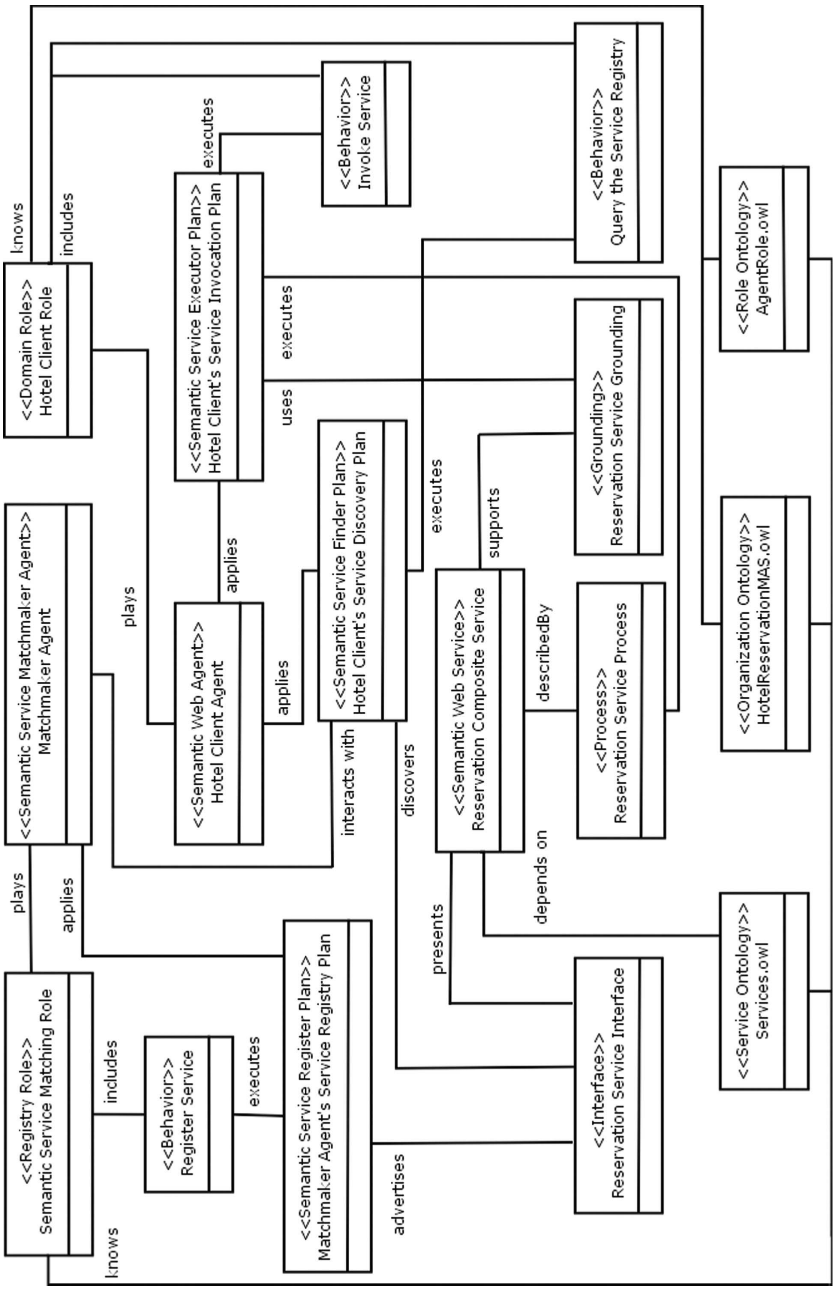
A Semantic Web enabled MAS is inconceivable without ontologies. An Ontology represents any information gathering and reasoning resource for MAS members. Collection of the ontologies creates knowledgebase of the MAS that provides domain context. Specializations of the Ontology called *Organization Ontology*, *Service Ontology* and *Role Ontology* are utilized by the related metamodel entities. For example semantic interface and capability description of services are formed according to the Service Ontology and this ontology is used by the Semantic Web Agents in order to discover and invoke Semantic Web Services.

Ontology entities in the proposed metamodel are defined as extensions of the *Ontology* element of the Ontology UML Profile (OUP) defined in Ref. 36. The OUP captures ontology concepts with properties and relationships and provides a set of UML elements available to use as semantic types in our metamodel. By deriving the semantic concepts from the OUP, we have already-defined UML elements to use as semantic concepts within the metamodel.

Semantic web service modeling languages (e.g. OWL-S[27]) mostly represent services by three semantic documents: *Service Interface*, *Process Model* and *Physical Grounding*. Service Interface is the capability representation of the service in which service inputs, outputs and any other necessary service descriptions are listed. Process Model describes internal composition and execution dynamics of the service. Finally, Physical Grounding defines invocation protocol of the web service. These Semantic Web Service components are given in the metamodel with *Interface*, *Process* and *Grounding* entities respectively. Semantic input, output and web service definitions used by those service components are imported from the UML Semantic Web Service Profile (USWSP) proposed in Ref. 41. Each semantic service input and output has a type which is a domain specific Ontology class. In order to represent those Ontology type entities in our UML 2.0 based metamodel, we also import the OntClass meta-entity defined in the OUP.[36]

Semantic Web Agents apply *Plan*s to perform their tasks. In order to discover and execute Semantic Web Services dynamically, two extensions of the Plan entity are defined in the proposed PIM. *Semantic Service Finder Plan* is a Plan in which discovery of candidate semantic web services takes place. During this plan execution, the agent communicates with the service matchmaker of the platform to determine proper semantic services. After service discovery, the agent applies the *Semantic Service Executor Plan* to execute appropriate semantic web services. Process model and grounding mechanism of the service are used within the plan.

On the other hand, agents need to communicate with a service registry in order to discover service capabilities. For this reason, the model includes a specialized agent entity, called *Semantic Service Matchmaker Agent*. This meta-entity



<<Domain Role>> Hotel Client Role
<<Semantic Service Matchmaker Agent>> Matchmaker Agent
<<Registry Role>> Semantic Service Matching Role
<<Semantic Service Executor Plan>> Hotel Client's Service Invocation Plan
<<Semantic Web Agent>> Hotel Client Agent
<<Behavior>> Register Service
<<Semantic Service Register Plan>> Matchmaker Agent's Service Registry Plan
<<Behavior>> Invoke Service
<<Semantic Service Finder Plan>> Hotel Client's Service Discovery Plan
<<Semantic Web Service>> Reservation Composite Service
<<Behavior>> Query the Service Registry
<<Grounding>> Reservation Service Grounding
<<Process>> Reservation Service Process
<<Interface>> Reservation Service Interface
<<Role Ontology>> AgentRole.owl
<<Organization Ontology>> HotelReservationMAS.owl
<<Service Ontology>> Services.owl
knows
includes
executes
plays
applies
uses
interacts with
discovers
supports
describedBy
depends on
presents
advertises

Fig. 4. An instance model for the agent — service interaction within a MAS working in Tourism domain.

represents the matchmaker agents which store the capability advertisements of semantic web services within a MAS and match those capabilities with service requirements sent by the other platform agents. These matchmaker agents apply the *Semantic Service Register Plan*.

An instance model of the above metamodel is given in Fig. 4 for the interaction between a Hotel Client Agent and a Reservation Service within a MAS working in Tourism domain. The client agent is a Semantic Web Agent which reserves hotel rooms on behalf of its human users. During its task execution, it needs to interact with a semantic web service called "Reservation Composite Service". The Hotel Client Agent applies its service discovery and invocation plans called "Hotel Client's Service Discovery Plan" and "Hotel Client's Service Invocation Plan" respectively. "Matchmaker Agent" is the service matcher of the related agent platform. It stores capability interfaces of the semantic services and matches them with client requests semantically by applying its register plan called "Matchmaker Agent's Service Registry Plan". Hence, "Hotel Client Agent" determines appropriate semantic service by asking the "Matchmaker Agent" in its service discovery plan and interacts with the selected semantic service by executing service's process description and using service's grounding. The model also includes related agent roles, behaviors and required ontologies for the domain.

5. Model Transformations for MDD of the Semantic Web Enabled MASs

Sendall and Kozaczynski[14] describe model transformation as the heart and soul of model driven software development. Indeed, definition of metamodels is required but not sufficient for a complete MDD process. We have to define transformations between those metamodels to obtain the main artifacts of the process: target models. We apply transformation between platform independent and platform specific MAS metamodels in order to achieve working model of Semantic Web enabled agent systems.[42]

Referring to Fig. 1, our source metamodel for the transformation is the PIM pictured in Figs. 3(a) and 3(b). The instance model conforming to this metamodel is the one given in Fig. 4. As mentioned before, this model is a platform independent Semantic Web enabled MAS model in which semantic agents reserve hotel rooms on behalf of their users by interacting with semantic web services. Now, we employ the transformation between PIM and various PSMs shown in Fig. 1 in order to facilitate the implementation of the specified agent system in different agent development environments. This can only be realized if we provide metamodels of the corresponding physical environments as platform specific metamodels and define transformation rules.

In this study, we employ two different agent development software frameworks called SEAGENT[17] and NUIN[43] for the implementation of our Tourism MAS. Therefore, metamodels of these frameworks are used as target metamodels (PSMs) in our MDD process. In the following subsections, metamodels of these frameworks and model transformation between our PIM and those metamodels are discussed.

5.1. *Model transformation for the SEAGENT framework*

Our first target platform for platform specific models is the SEAGENT. SEAGENT[17] is an agent development software framework and platform that is specialized for semantic web-based MAS development. The communication and plan execution infrastructure of the SEAGENT looks like other existing agent development frameworks such as DECAF,[44] JADE[45] and RETSINA.[46] However, as discussed in Ref. 32, in order to support and facilitate semantic web-based MAS development, SEAGENT includes the following built-in features which the existing agent frameworks and platforms do not have:

- SEAGENT provides a specific feature within the agent's internal architecture to handle the agent's internal knowledge using Web Ontology Language (OWL).[47]
- The directory service of the SEAGENT stores agent capabilities using specially designed OWL-based ontologies and it provides a semantic matching engine to find the agents with semantically related capabilities.
- Based on FIPA-RDF,[24] a content language called Seagent Content Language (SCL) has been defined to transfer semantic content within the agent communication language messages.
- SEAGENT introduces a new service called Ontology Management Service (OMS). The most important feature of this service is to define the mappings between the platform ontologies and the external ontologies. Then it provides a translation service to the platform agents based on these defined mappings.
- SEAGENT supports discovery and dynamic invocation of semantic web services by introducing a new platform service for semantic service discovery and a reusable agent behavior for dynamic invocation of the discovered services.

SEAGENT is implemented in Java and provides libraries to develop Semantic Web enabled MASs also in Java. The metamodel of SEAGENT considering HTN-based agent planner and service interaction is given in Fig. 5. The upper part of the metamodel includes the SEAGENT planner components. In the SEAGENT framework, agents execute their *Task*s according to HTN.[33] HTN planning creates plans by task decomposition as an AI planning methodology. This decomposition process continues until the planning system finds primitive tasks that can be performed directly. For example, semantic web service interaction of an agent may be modeled as a reusable plan which composes service discovery, service engagement and dynamic service invocation tasks. The Semantic Web agent executes those tasks in order to utilize a semantic web service.

As a requirement of HTN, tasks might be either complex (called *Behavior*s) or primitive (called *Action*s). Each plan consists of a complex root task consisting of sub-tasks to achieve a predefined goal. Behaviors hold a "reduction schema"

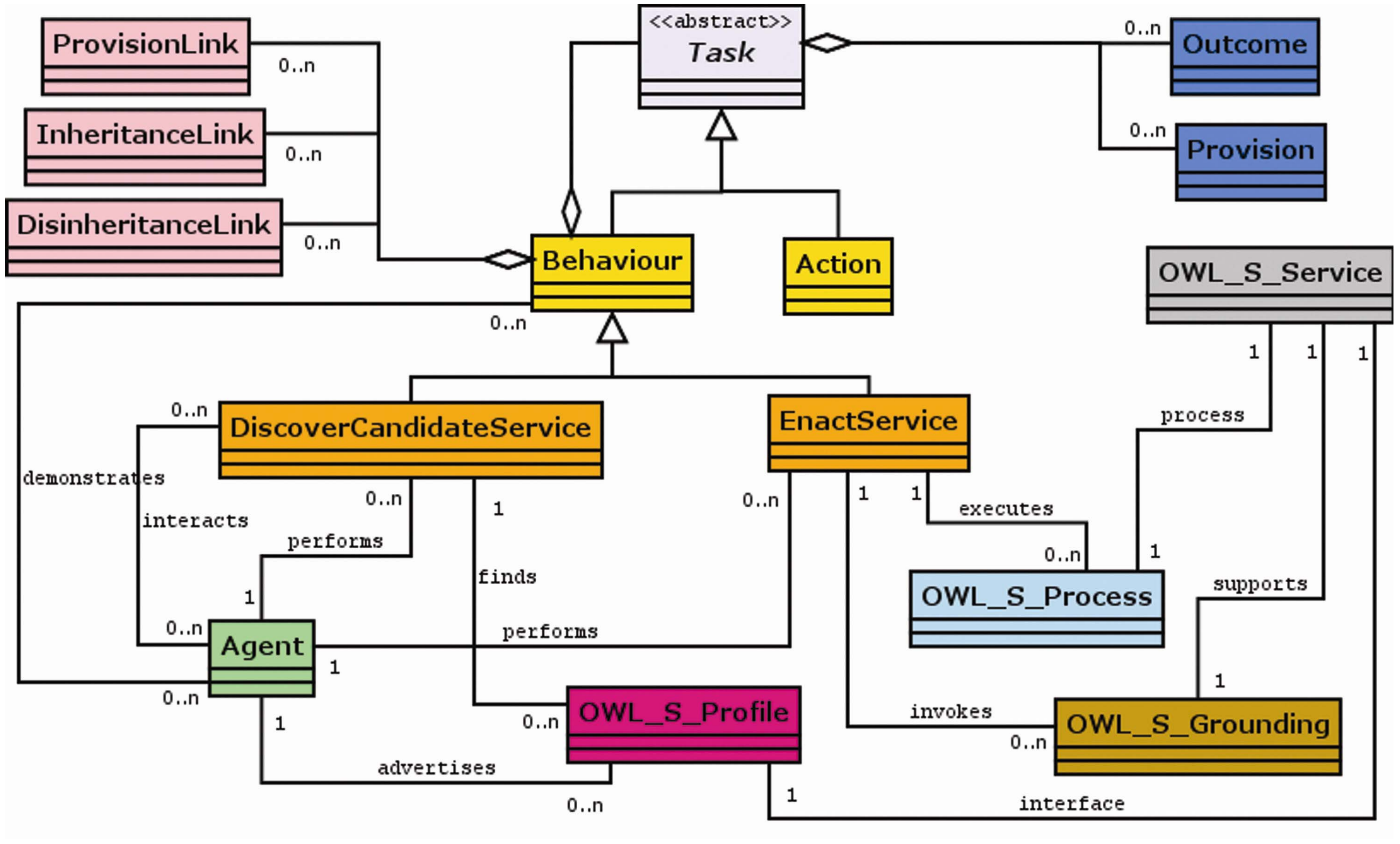


Fig. 5. The metamodel of SEAGENT framework in agent planner and service interaction viewpoints.

knowledge that defines the decomposition of the complex task to the sub-tasks and the information flow between these sub-tasks and their parent task. The information flow mechanism is as follows: each task represents its information need by a set of provisions and the execution of a task produces outcomes, and there are links that represent the information flow between tasks using these provision and outcome slots. There are three types of these links: *Provision Links* that connect outcomes to provisions, *Inheritance Links* that connect provision in a parent task to one of its sub-tasks and *Disinheritance Links* that connect outcome of a sub-task to its parent task.

Actions are primitive tasks that can be executed directly by the planner. Also, each task produces an outcome state after its execution. The outcome states are used to route the information flow between tasks. Tasks have a name describing what they are supposed to do and have zero or more *Provision*s (information needs) and *Outcome*s (execution results). The provision information is supplied dynamically during plan execution. Tasks are ready, and thus eligible for execution, when there is a value for each of its provisions. More detailed information about SEAGENT plan structure can be found in Ref. 48.

Discover Candidate Service and *Enact Service* are two predefined Behavior extensions which provide semantic web service interaction for *Agent*s. In Discover Candidate Service Behavior, an Agent performs discovery of semantic web services that matches with the agent's requirements while execution of the service according to service's process definition is realized in the Agent's Enact Service Behavior.

Semantic web services are implemented in SEAGENT as OWL-S Services.[27] Therefore each semantic web service is defined as an *OWL-S Service* instance in SEAGENT models and has *OWL-S Profile*, *OWL-S Process* and *OWL-S Grounding* constructs in order to provide service usage by the platform agents.

The above metamodel of SEAGENT is used as one of the PSMs in the proposed MDD process and transformation between PIM and this PSM is realized within the study. The crucial part of the transformation process is to define the transformation rules in a predefined transformation language. Those rules are based on the mappings between source and target model entities. The rules also include the formal representation of mapping constraints which are applied during the transformation. In our case, we have to define the mappings between entities of the PIM and the PSM of the SEAGENT framework. In Table 1, significant entity mappings are listed.

Mappings between our metamodel entities and the SEAGENT concepts are not always in one-to-one manner. For instance, the SEAGENT framework does not define a *Role* abstraction. That abstraction is built in the definition of the *Agent* class. Hence both the *Role* and the *Semantic Web Agent* entities of the metamodel are mapped into the *Agent* class of the SEAGENT metamodel.

On the other hand, the *Task* entity in the SEAGENT metamodel corresponds to the abstract *Plan*s of our PIM and the predefined behavior classes of the SEAGENT planner library correspond to the extended Plans of the PIM. For example, the SEAGENT planner includes the behavior class called *DiscoverCandidateService*

Table 1. Entity mappings between the PIM and the PSM of the SEAGENT.

PIM Entity	SEAGENT Entity	Explanation
Role, Semantic Web Agent (SWA), Semantic Service Matchmaker Agent (SSMA)	Agent	Role, SWA and SSMA in the metamodel correspond to the Agent in SEAGENT.
Plan	Task	Agent plans are mapped onto the HTN tasks in the SEAGENT framework.
Semantic Service Register Plan	Behavior	Register plan of the matchmaker agent is implemented as a Behavior in the SEAGENT model.
Semantic Service Finder Plan	DiscoverCandidateService	Service finder plan of the SWA is mapped into built-in Behavior subclass called DiscoverCandidateService.
Semantic Service Executor Plan	EnactService	Service executor plan of the SWA is mapped into built-in Behavior subclass called EnactService.
Behavior	Action	Agent behaviors defined in the metamodel can be implemented as Action instances in the SEAGENT Platform.
Semantic Web Service Interface Process Grounding	OWL-S_Service OWL-S_Profile OWL-S_Process OWL-S_Grounding	In SEAGENT, capabilities and process models of semantic web services are defined by using the OWL-S markup language.

which provides the discovery of semantic services according to their capability advertisements as mentioned before. When an agent acts according to that behavior, in fact it applies the Semantic Service Finder Plan which is defined in our PIM. Hence, that behavior class in the SEAGENT is the physical counterpart of our related metamodel entity.

Since capabilities and process models of the semantic web services are defined in SEAGENT by using the OWL-S markup language, we provide the translations derived from the mappings between our service metamodel entities (Interface, Process and Grounding) and OWL-S components of the SEAGENT as given in Table 1.

After the execution of the whole transformation process between the PIM and the SEAGENT PSM, we achieve the platform specific model of our tourism MAS. This output (target) model is given (in Fig. 8) and discussed at the end of Sec. 6.2 of this paper.

5.2. *Model transformation for the NUIN framework*

Our second target platform for implementing the desired MAS is the NUIN. NUIN[43] provides a flexible agent architecture designed to develop agents in Semantic Web applications based around Belief-Desire-Intention (BDI) principles. The implementations on the NUIN framework strongly embody the BDI theoretical foundation[49] which is perhaps the most commonly studied architecture for deliberative agents.

On the other hand, NUIN aims to make semantic web notations and representations central to the design of the agents. So, agents are able to operate as semantic web agents. While being a semantic web agent admits a number of interpretations as discussed in Ref. 50, it should be noted that NUIN agents can only provide some of those interpretations such as: They are configured using RDF[40] and in principle they can report this configuration in response to queries. NUIN agents can use semantic web information sources as agent belief bases, by including JENA model specifications[31] in the agent configuration, and they have a limited ability to query external semantic web information sources. Also, the NUIN language supports the semantic web notion of identity by requiring all symbolic constants to be URI's. These design basics of the NUIN framework and the "semantic web agent" vision that we share with the NUIN cause us to choose the NUIN as another candidate for the system development environment in exemplifying the applicability of our MDD process.

NUIN provides a Java API (Application Program Interface) for developers to build agent systems. Internally, the agent plans are Java objects, and can be generated in a variety of different ways, including by directly invoking the action constructors. However, the most convenient way, is to program agents by using an agent scripting language called Nuinscript, which is parsed into the Java objects used by the interpreter. For this reason, we first derived the metamodel of the Nuinscript agent scripting language from its Extended Backus-Naur Form (EBNF) specification given in Ref. 50 and employed this metamodel as another PSM during the MDD process. Figure 6 depicts this metamodel.

The whole discussion of the Nuinscript language is beyond the scope of this paper and interested readers may refer to Ref. 50 for this discussion and Ref. 51 for the complete script specification. However, a brief explanation on meta-entities of the language and their relations is given below in order to provide understanding of further model transformation and its implementation.

A Nuinscript script includes declaration of namespaces, knowledge stores and plans for a *NUIN Agent* configuration. There exist *Namespace Declaration*s in which URIs for namespaces are given to provide interoperation with other semantic web knowledge sources easier. A script can contain any number of collections of initial data in *Knowledgestore Declaration*s for the agent's knowledge bases. Those collections constitute the agent's beliefs. A knowledge store may be associated with one or more ontologies. Each *Ontology Specification* is identified by URI. Also, a knowledge store may specify a set of initial sentences that are asserted into the knowledge store before the agent starts. These sentences are denoted as *Axiom*s.

High-level behavior of the NUIN agents is specified by the *Plan*s that they can carry out. Plans are named by *Identifier*s. Identifiers are unique within the scope of a single agent. Each Plan contains *Plan Element*s which can be a *Trigger*

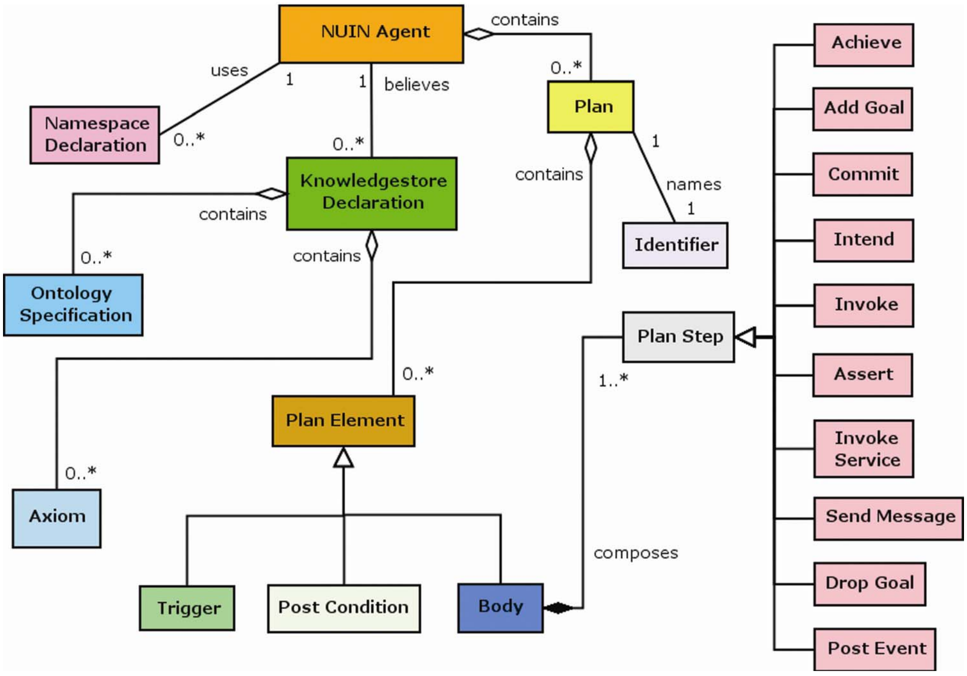


Fig. 6. The metamodel of the Nuinscript agent scripting language.

(a pattern over events that the agent may perceive), a *Post Condition* (a logical sentence denoting the condition that is stated to be achieved by the plan) and a *Body*. Body of an agent plan is composed of *Plan Step*s which represent actions. An agent's intention is represented with a series of actions that the agent will perform or a logical goal that it is attempting to achieve or a mixture of them.[51] Some platform dependent and predefined plan step extensions such as *Add Goal*, *Send Message* and *Invoke Service* are also included within the NUIN library and they are denoted in the rightmost side of the metamodel figure.

We have to define entity mappings between the PIM and NUIN PSM in order to write transformation rules and hence realize the model transformation for the source models in the same manner with the SEAGENT transformations. Mappings between metamodel elements of our PIM and the Nuinscript are listed in Table 2.

Agent entities in the PIM are mapped into NUIN Agents in the target metamodel as expected. *Ontology Specification*s which are used as knowledge stores in the NUIN platforms correspond to the Ontology meta-entities of the PIM. Conceptually, each *Plan* instance in the platform independent model is represented with a NUIN Plan in the target model. However, in fact, transformation between these elements is much more than a simple Plan-to-Plan mapping. For each PIM Plan, we have to create corresponding NUIN *Plan* instance and its *Identifier* and a *Body* as a container for the plan steps. Hence, the transformation for the Plan elements

Table 2. Entity mappings between the PIM and the Nuinscript.

PIM Entity	Nuinscript Entity	Explanation
Semantic Web Agent (SWA), Semantic Service Matchmaker Agent (SSMA)	NUIN Agent	NUIN Agent in the Nuinscript metamodel corresponds to SWA and SSMA in the PIM.
Ontology, Role Ontology, Service Ontology, Organization Ontology	Ontology Specification	Organizational, role or any other ontology in the metamodel are represented as ontological knowledge store declarations in the NUIN platform.
Plan, Semantic Service Finder Plan, Semantic Service Executor Plan, Semantic Service Register Plan,	Plan Identifier Body	Agent plans pertaining to semantic service interaction can be implemented as NUIN Plans and their identifiers. For each plan, a Body element is created in order to provide a container for plan steps.
Behavior	Plan Step	Agent behaviors defined in the PIM can be implemented as Plan Step instances in the NUIN Platform.

includes a mapping between the Plan meta-entity of the PIM and Plan, Identifier and Body elements of the Nuinscript metamodel. Agent *Behavior*s defined in the PIM are mapped into the *Plan Step* entities of the NUIN metamodel.

The platform specific model of our tourism MAS for the NUIN framework is obtained after the execution of the whole transformation process between the PIM and the Nuinscript PSM. This output (target) model is given (in Fig. 9) and discussed at the end of Sec. 6.2 of this paper.

Model transformation for the NUIN platform differs from the SEAGENT transformation in two major viewpoints: Completeness and Complexity of the transformations. Model transformations from PIM to SEAGENT PSM are more complete than the NUIN PSM. We can provide appropriate (or nearly exact) counterparts of the most of the metamodel entities and their associations in the SEAGENT PSM. Hence, in most cases, we can achieve the realization of the designed model defined in the platform independent layer by using the SEAGENT environment. However, we need to provide new entity definitions and Nuninscript entity extensions to complete the realization of the same model in the NUIN environment especially when we consider semantic web service structures. Semantic web support of the NUIN remains only in agent configuration and knowledge store declaration. In order to provide real implementation of the agent — semantic web service interactions, we defined external software constructs for the NUIN (both in agent plans and semantic web service descriptions) which are not currently included in the Nuinscript metamodel.

On the other hand, model transformation for the NUIN is more complicated and difficult to implement than SEAGENT transformations. Defined rules for the

SEAGENT PSM transformation mostly include attribute settings for the target entities and mapping of source entities into those target entities after recognition of related model instances at the source pattern. However model transformation for NUIN environment requires more than just entity mappings between our PIM and metamodel of the Nuinscript. As they are discussed at the following section, rules written for NUIN transformations include various dynamic model element creations, complex Object Constraint Language (OCL) queries[52] for detection of source elements on the pattern and selection of derived instances according to the relations on the target environment in addition to n-to-n entity mappings. Those differences in model transformations for NUIN and SEAGENT PSMs are naturally expected because SEAGENT platform and our PIM are compatible due to their abstractions, design mechanisms and environments that they model.

6. Implementation of the Model Transformations Using ATL

We implemented the whole model transformation processes (both for SEAGENT and NUIN PSMs) discussed in this study by using ATLAS INRIA & LINA research group's Atlas Transformation Language (ATL). ATL[20] is a widely accepted model transformation language, specified as both a metamodel and a textual concrete syntax. It is defined to perform general transformations within the MDA framework. It also provides a development environment as a plugin in Eclipse Open Source Development Platform.[53] These advantages are the main reasons of choosing ATL as our implementation language. The following subsections provide an overview about our transformation implementation in ATL and discuss the prepared model transformation rules for each PSM.

6.1. *Overview of the implementation in ATL*

The structure of an ATL[b] transformation is composed of the following elements[54]:

- a header section that defines some attributes that are relative to the transformation
- an optional import section that enables to import some existing ATL libraries
- a set of helpers that can be viewed as an ATL equivalent to Java methods
- a set of rules that defines how target models are generated from source models

ATL helpers can be viewed as the ATL equivalent to Java methods. They make it possible to define factorized ATL code that can be called from different points of an ATL transformation.

[b]There are two versions of ATL compiler in use now: ATL 2004 and ATL 2006. ATL 2004 allows only one pattern element in the query part of rules. Therefore, we can only define one-to-one and one-to-many transformations in a single rule. ATL 2006 allows multiple source elements. We intentionally used ATL 2004 because it had more detailed documentation and more stable execution environment at the time of this study. We use ATL to refer ATL 2004 throughout this paper.

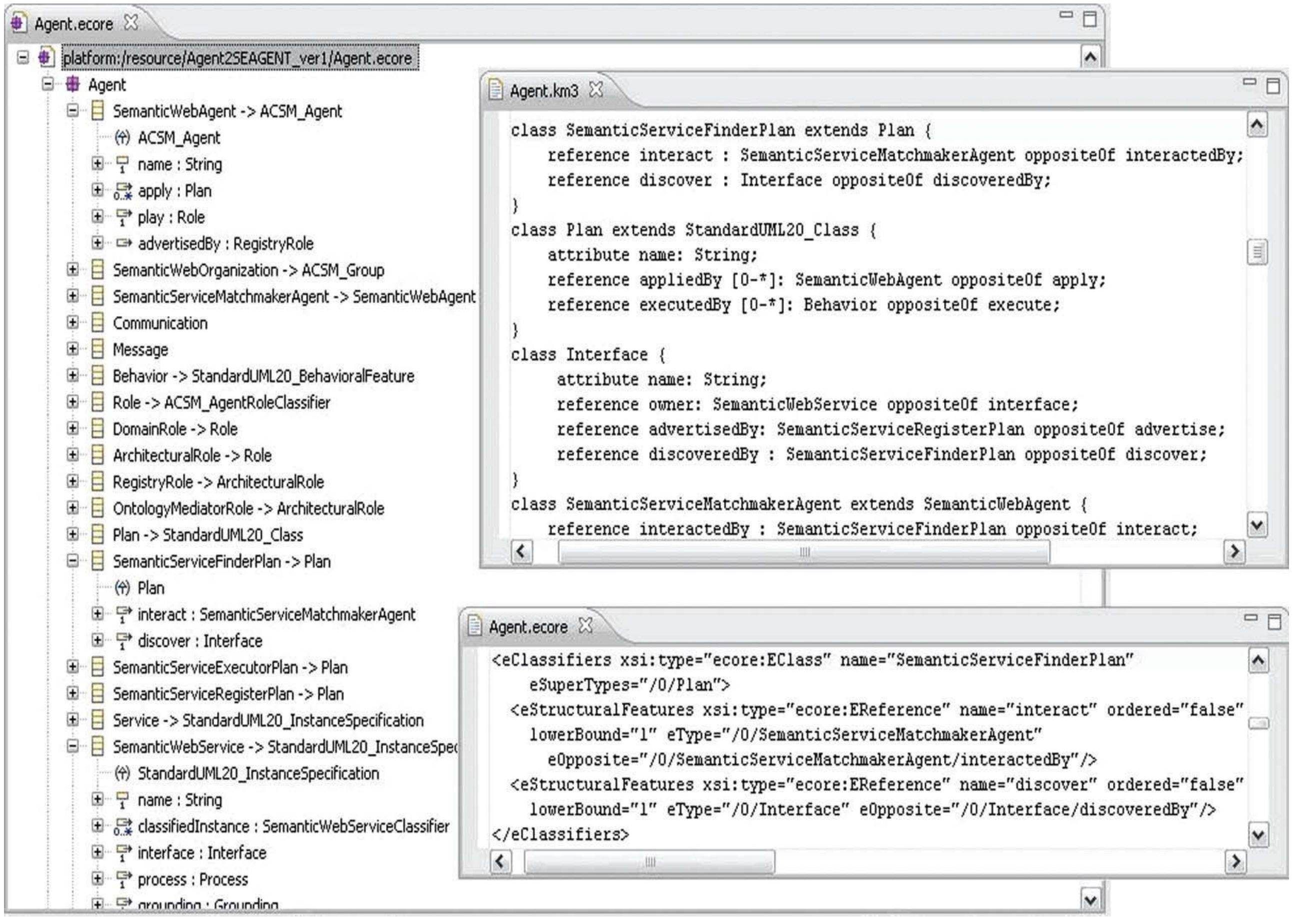
The transformation rule in ATL has the "*to*" section which allows only one element to query the model elements in the source part and the "*from*" section for the target part which allows multiple elements. Since ATL does not have a direct support for in-place model updates, it only considers create operations. Detailed information about these language features can be found in Refs. 20 and 54.

Referring back to the model transformation process depicted in Fig. 1, we use the MOF variant called Ecore[55] as the metametamodel (MMM) of all metamodels (SMM, TMM and TgMM) within the implementations. Our transformation metamodel (TMM) is the ATL and source metamodel (SMM) is the platform independent MAS metamodel discussed in Sec. 4. Our source model (sm) is the platform independent model of the tourism MAS pictured in Fig. 4. When we apply transformations (tm) into our source model, we aim to obtain the platform specific target models (tgm) of our tourism MAS in the SEAGENT and the NUIN frameworks which conform to the target metamodels (TgMMs) given in Figs. 5 and 6 respectively.

In order to use ATL engine, we need to prepare Eclipse Modeling Framework (EMF) encodings -ecore files- of both source and target metamodels. EMF provides its own file format (.ecore) for model and metamodel encoding. However manual edition of Ecore metamodels is particularly difficult with EMF. To make this common kind of editions easier, the ATL Development Tools (ADT) include a simple textual notation dedicated to metamodel edition: the Kernel MetaMetaModel (KM3).[56] This textual notation facilitates the edition of metamodels. Once edited, KM3 metamodels can be injected into the Ecore format using the ADT integrated injectors. More information about KM3 and Ecore injection can be found in Refs. 20 and 56. Hence, we first prepare the KM3 representations of the source metamodel and the target metamodels and then inject them into the Ecore format. The maximized editor window in Fig. 7 portrays the visual representation of our source metamodel (PIM) in the ATL environment after the source KM3 file has been injected into Ecore format.

UML superstructures used in our PIM are not used directly in model transformations. However they should exist as fundamental structures in Ecore representation of our PIM since PIM entities extend these structures. Therefore UML superstructure entities extended by our entities are also represented in KM3 notation of the PIM and when we inject this KM3 model into Ecore, we automatically achieve Ecore counterparts of these UML elements.

After both the source and the target metamodels are prepared for the transformation, heuristic rules for the transformation should be written in the transformation language (herein ATL) according to the entity mappings defined between source and target metamodels. Each ATL rule for the transformation defines a source model element in its source part and has the full definition of constraints to query the whole source pattern in the model. Then we have to prepare source model also in Ecore format for the transformation process. ATL engine applies the written rules on this source model and produces the target model conforming to each PSM again in Ecore format.



```
Agent.ecore
platform:/resource/Agent2SEAGENT_ver1/Agent.ecore
  Agent
    SemanticWebAgent -> ACSM_Agent
      ACSM_Agent
      name : String
      apply : Plan
      play : Role
      advertisedBy : RegistryRole
    SemanticWebOrganization -> ACSM_Group
    SemanticServiceMatchmakerAgent -> SemanticWebAgent
    Communication
    Message
    Behavior -> StandardUML20_BehavioralFeature
    Role -> ACSM_AgentRoleClassifier
    DomainRole -> Role
    ArchitecturalRole -> Role
    RegistryRole -> ArchitecturalRole
    OntologyMediatorRole -> ArchitecturalRole
    Plan -> StandardUML20_Class
    SemanticServiceFinderPlan -> Plan
      Plan
      interact : SemanticServiceMatchmakerAgent
      discover : Interface
    SemanticServiceExecutorPlan -> Plan
    SemanticServiceRegisterPlan -> Plan
    Service -> StandardUML20_InstanceSpecification
    SemanticWebService -> StandardUML20_InstanceSpec
      StandardUML20_InstanceSpecification
      name : String
      classifiedInstance : SemanticWebServiceClassifier
      interface : Interface
      process : Process
      grounding : Grounding
```

```
Agent.km3
class SemanticServiceFinderPlan extends Plan {
    reference interact : SemanticServiceMatchmakerAgent oppositeOf interactedBy;
    reference discover : Interface oppositeOf discoveredBy;
}
class Plan extends StandardUML20_Class {
    attribute name: String;
    reference appliedBy [0-*]: SemanticWebAgent oppositeOf apply;
    reference executedBy [0-*]: Behavior oppositeOf execute;
}
class Interface {
     attribute name: String;
     reference owner: SemanticWebService oppositeOf interface;
     reference advertisedBy: SemanticServiceRegisterPlan oppositeOf advertise;
     reference discoveredBy : SemanticServiceFinderPlan oppositeOf discover;
}
class SemanticServiceMatchmakerAgent extends SemanticWebAgent {
    reference interactedBy : SemanticServiceFinderPlan oppositeOf interact;
```

```
Agent.ecore
<eClassifiers xsi:type="ecore:EClass" name="SemanticServiceFinderPlan"
    eSuperTypes="/0/Plan">
  <eStructuralFeatures xsi:type="ecore:EReference" name="interact" ordered="false"
    lowerBound="1" eType="/0/SemanticServiceMatchmakerAgent"
      eOpposite="/0/SemanticServiceMatchmakerAgent/interactedBy"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="discover" ordered="false"
    lowerBound="1" eType="/0/Interface" eOpposite="/0/Interface/discoveredBy"/>
</eClassifiers>
```

Fig. 7. Visual representation of the source metamodel in the ATL environment. The upper right window shows a fragment of the source metamodel in KM3 format while the lower right window shows the textual ecore representation of the related metamodel fragment.

Due to space limitations, it is not feasible to show whole KM3 representations and ATL rule definitions of our implementation. However, to give some flavor of the implementation in here, we describe transformation of the "Semantic Service Finder Plan" source instance into their corresponding entities in the SEAGENT and the NUIN frameworks. Again in Fig. 7, at the upper right window, the part of the source KM3 file in which "Semantic Service Finder Plan" is represented with its associations for "Interface" and "Semantic Service Matchmaker Agent" entities. In its Java-like syntax, KM3 representations provide the definition of every source metamodel entity and its relationships with other entities. Using ADTs injection feature we prepare Ecore format of the above given KM3 representation automatically. The lower right window in Fig. 7 shows the definition of "Semantic Service Finder Plan" meta-entity again but this time in Ecore format.

The source model should also be given in Ecore format as the input for the transformation processes. In our case study, we have to provide Ecore representation of the tourism MAS model conforming to the source metamodel (PIM). Considering our plan transformation, this model includes the following model instance in which the Semantic Service Finder Plan called "Hotel Client's Service Discovery Plan" is defined with its associations to other model elements. Numbers refer to the exact order of the model instances in the full source model Ecore specification. References to the other instances are omitted and specified with "...".

```
<xmi:XMI xmi:version="2.0" xmlns:xmi=http://www.omg.org/XMI xmlns="Agent">
  <SemanticServiceFinderPlan name="Hotel Client's Service Discovery Plan"
       appliedBy="{\#}/0" executedBy="{\#}/10" interact="{\#}/1" discover="{\#}/7"/>
  <SemanticWebAgent name="Hotel Client Agent" apply="{\#}/4 ..." play="..."/>
  <Behavior name="Query the Service Registry" includedBy="..."
       execute="{\#}/4" knowRoleOntology="..."/>
  <SemanticServiceMatchmakerAgent name="Matchmaker Agent" apply="..."
       play="..." interactedBy="{\#}/4"/>
  <Interface name="Reservation Service Interface" owner="..."
       advertisedBy="..." discoveredBy="{\#}/4"/>
</xmi:XMI>
```

6.2. *Rules for the model transformation in ATL*

According to the entity mappings, heuristic rules for the transformation should be given in ATL as mentioned in Sec. 6.1. For instance, we write two ATL rules called `SemanticServiceFinderPlan2DiscoverCandidateService` and `SemanticServiceFinderPlan2Plan` in order to transform Semantic Service Finder Plan instances in source models into their platform specific counterparts in SEAGENT and NUIN frameworks respectively. The Semantic Service Finder Plan class in the source part of those rules needs the full constraint definition of the source pattern to match in the model because the constraint part requires constraints of other source pattern elements related to the Semantic Service Finder Plan class to bind the appropriate model element. The helper rules are required in the constraint part to define the relationships between the pattern elements.

Following is the `SemanticServiceFinderPlan2DiscoverCandidateService` ATL rule:

```
1 rule SemanticServiceFinderPlan2DiscoverCandidateService {
2       from fndpln: Agent!SemanticServiceFinderPlan (
3                 fndpln.partofPatternforFinderPlan )
4       to plnf: SEAGENT!DiscoverCandidateService (
5         name <- fndpln.name,
6         containTask <- Sequence{Agent!Behavior->allInstances()->
           asSequence()->select(bhv|bhv.execute->exists(pln|pln=fndpln))},
7         interact <- Sequence {Agent!SemanticServiceMatchmakerAgent->
                                allInstances()->asSequence()},
8         find <- Sequence {Agent!Interface->allInstances()->asSequence()}
  )
9 }
```

In the above rule, the transformation of the `SemanticServiceFinder Plan` entity into its corresponding target model entity (DiscoverCandidateService class of the SEAGENT) is realized. We need to call the helper rule called `partof PatternforFinderPlan` for the relations of the `SemanticServiceFinderPlan` entity with its attributes. These helpers are the realization of the constraints to query the models. The constraints in ATL are specified as OCL constraints.[52] Same helper rules and constraint repetitions may be required both for other rules in the same target model transformation or other PSM transformations (e.g. NUIN). Hence this kind of rule decomposition makes the definitions easier. The helper `partofPatternforFinderPlan` called in line 3 of the above rule is given below:

```
1 helper context Agent!SemanticServiceFinderPlan def:
2  partofPatternforFinderPlan : Boolean =
3    if self.appliedBy->forAll(agnt|
        not agnt.oclIsTypeOf(Agent!SemanticServiceMatchmakerAgent)
4       and not agnt.play.oclIsTypeOf(Agent!RegistryRole))
5       and self.interact.play.oclIsTypeOf(Agent!RegistryRole)
6       and self.interact.apply->
             select(p|p.oclIsTypeOf(Agent!SemanticServiceRegisterPlan))->
             forAll(p|p.advertise->exists(intfc|intfc.discoveredBy=self))
7       and self.discover.advertisedBy.appliedBy->
            exists(sma|sma.interactedBy=self)
8  then true
9  else false
10 endif;
```

The helpers correspond to the constraint part of the related rules. There are two types of helper in our transformations as discussed in Ref. 42. The first type helpers like `partofPatternforFinderPlan` are used to check if the model element is the part of the pattern or not. The second type helpers are used to select the elements for creating relations between target elements. As a first type helper, `partofPatternforFinderPlan` is used to select the Semantic Service Finder Plan instances in pattern matching. The conditions between line 3 and line 7 check the relations of Plan instances with other instances like Semantic Service Matchmaker Agent, Registry Role and Interface. They determine if the matched plan is a part of the pattern or not. If the conditions in the helper are satisfied by the instance, the helper returns the boolean value "true".

During the transformation process, the ATL engine applies the above rule (`SemanticServiceFinderPlan2DiscoverCandidateService`) in order to transform "Hotel Client's Service Discovery Plan" into a SEAGENT DiscoverCandidateService instance. The Ecore representation of this obtained target instance is given below. Reference numbers to the other generated model instances are omitted again and denoted with "...":

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="SEAGENT">
  <DiscoverCandidateService name="Hotel Client's Service Discovery Plan"
     interact="..." find="..." performedBy="...">
      <containTask xsi:type="Action" name="Query the Service Registry"/>
  </DiscoverCandidateService>
</xmi:XMI>
```

On the other hand, following rule called `SemanticServiceFinderPlan2Plan` is written for the transformation of Semantic Service Finder Plan instances into NUIN Plan entities for the NUIN PSM. Above mentioned `partofPatternforFinderPlan` helper is again used in here (line 3) to provide selection of Semantic Service Finder Plan instances in the source model. However, transformation rule now becomes complicated when we compare it with the one written for the SEAGENT transformation. As discussed in Sec. 5.2, Plan entities in our PIM are mapped onto "Plan", "Identifier" and "Body" entities in the Nuinscript metamodel. Therefore this rule includes one source part but three target parts. For each finder plan, a NUIN Plan instance and its Identifier are created first and transformation of other model associations is realized (lines between 5 and 11). Then a NUIN Body is created for the plan (lines between 12 and 16) in order to provide a container for the related plan steps which are counterparts of the behaviors of this plan in the source model.

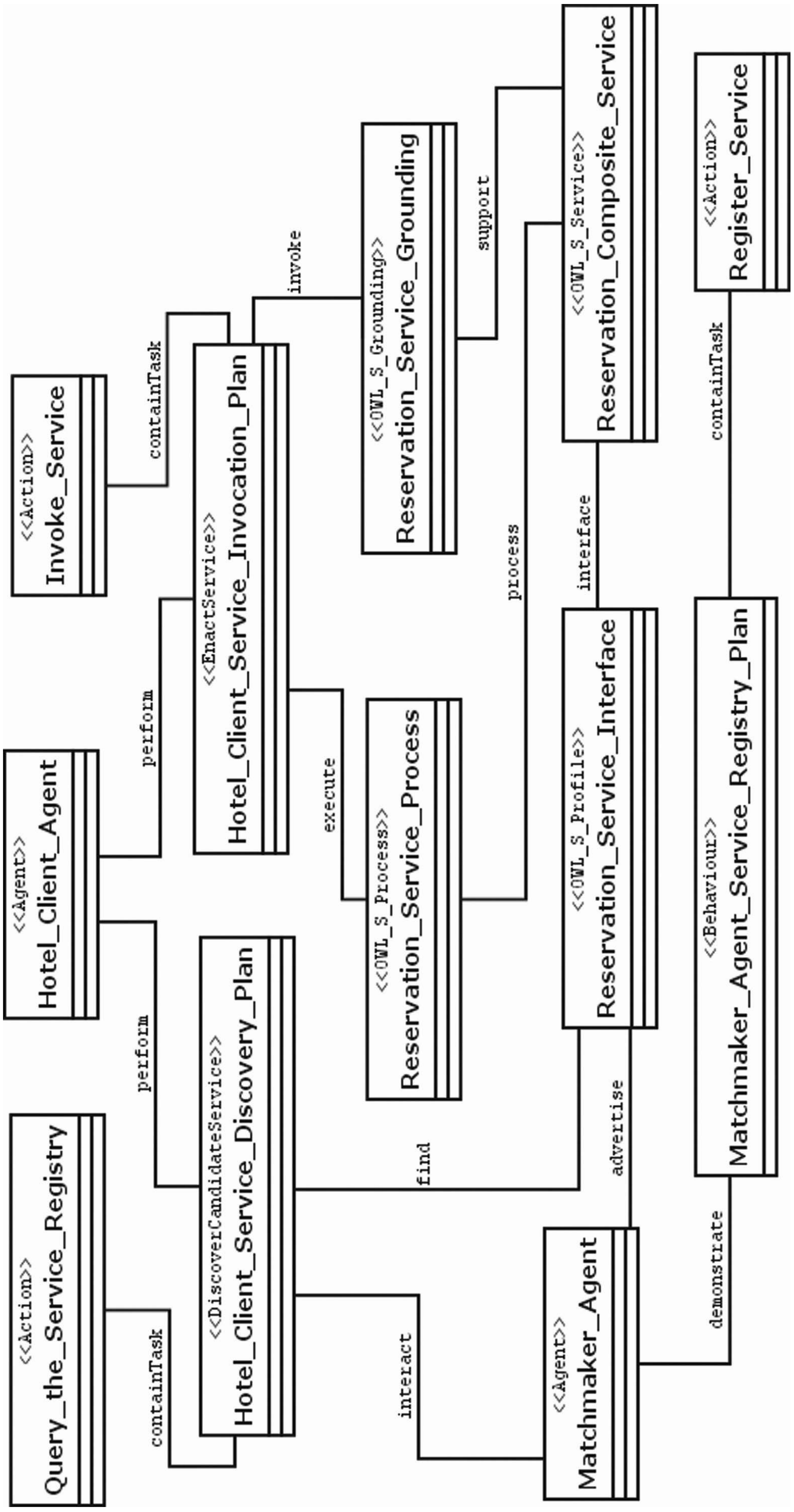
```
1 rule SemanticServiceFinderPlan2Plan {
2       from fndpln: Agent!SemanticServiceFinderPlan (
3          fndpln.partofPatternforFinderPlan )
4        to
5       plnf: NUIN!Plan (
6         containedBy <- Agent!SemanticWebAgent->allInstances()->
                         asSequence()->select(agt|agt.apply->
                           exists(pln|pln=fndpln))->first()
7         ),
8         identifier: NUIN!Identifier (
9              id <- fndpln.name,
10             name <- Agent!SemanticServiceFinderPlan->allInstances()->
                        asSequence()->select(pln|pln=fndpln)->first()
11        ),
12        plnfBody: NUIN!Body (
13             name <- 'Finder Plan Body',
14             containedBy <- plnf,
15             compose <- Sequence{Agent!Behavior->allInstances()->
                           asSequence()->select(bhv|bhv.execute->
                             exists(pln|pln=fndpln))}
16        )
17 }
```

When the ATL engine applies the above rule, the target instance of the “Hotel Client’s Service Discovery Plan” in the NUIN framework is obtained. Notice that the related plan and its required plan step instance are represented as inner elements within the definition of the target NUIN Agent instance with <contain> and <compose> tags respectively. The identifier of the plan is represented with a standalone element. Such kind of Ecore element representation is also correct and it is just the preference of the ATL engine for the automatic generation of the output model:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="NUIN">
  <NUINAgent name="Hotel Client Agent" believe="...">
    <contain namedBy="/4">
      <containPlanElement xsi:type="Body" name="Finder Plan Body">
        <compose name="Query the Service Registry"/>
      </containPlanElement>
    </contain>
  </NUINAgent>
  <Identifier id="Hotel Client's Service Discovery Plan"
     name="/0/@contain.1"/>
</xmi:XMI>
```

In our complete implementations, we first provided KM3 representations of all metamodels: our PIM for Semantic Web enabled MASs and SEAGENT and Nuinscript metamodels as PSMs. We also supplied the model of our tourism MAS in Ecore format as the transformation input. Then we prepared model transformation rules written in ATL. Finally, after execution of the whole process in ATL environment, we obtained the platform specific models of the tourism MAS.

The model in Fig. 8 describes components and their relations of our tourism MAS implemented in the SEAGENT environment. The model includes the `Hotel_Client_Agent` that discovers hotel reservation services with semantic capability interfaces according to its `Hotel_Client_Service_ Discovery_Plan`. It communicates with the `Matchmaker_Agent` of the system during execution of this plan. The register plan of the matchmaker agent (`Matchmaker_Agent_Service_Registry_Plan`) is represented as a SEAGENT behavior instance in the output model. Discovery plan extends `DiscoverCandidateService` behavior. This behavior is the corresponding entity for the “Semantic Service Finder Plan” meta-entity given in our PIM. Similarly, agent’s service execution plan (`Hotel_Client_Service_Invocation_Plan`) is an `EnactService` behavior and this behavior is the counterpart of our PIM’s “Semantic Service Executor Plan” meta-entity. Agent behaviors defined in the source model are now represented as SEAGENT plan actions in the target model. Semantic web services are OWL-S services in SEAGENT. Hence, our reservation service becomes an `OWL_S_Service` with its profile, process and grounding descriptions after the transformation as expected.



<<Action>> Query_the_Service_Registry
<<Agent>> Hotel_Client_Agent
<<Action>> Invoke_Service
<<DiscoverCandidateService>> Hotel_Client_Service_Discovery_Plan
<<EnactService>> Hotel_Client_Service_Invocation_Plan
<<OWL_S_Process>> Reservation_Service_Process
<<OWL_S_Grounding>> Reservation_Service_Grounding
<<Agent>> Matchmaker_Agent
<<OWL_S_Profile>> Reservation_Service_Interface
<<OWL_S_Service>> Reservation_Composite_Service
<<Behaviour>> Matchmaker_Agent_Service_Registry_Plan
<<Action>> Register_Service
containTask
perform
perform
containTask
invoke
execute
support
process
find
interact
advertise
interface
demonstrate
containTask

Fig. 8. The target model of the tourism MAS obtained after the PIM to SEAGENT PSM transformation.

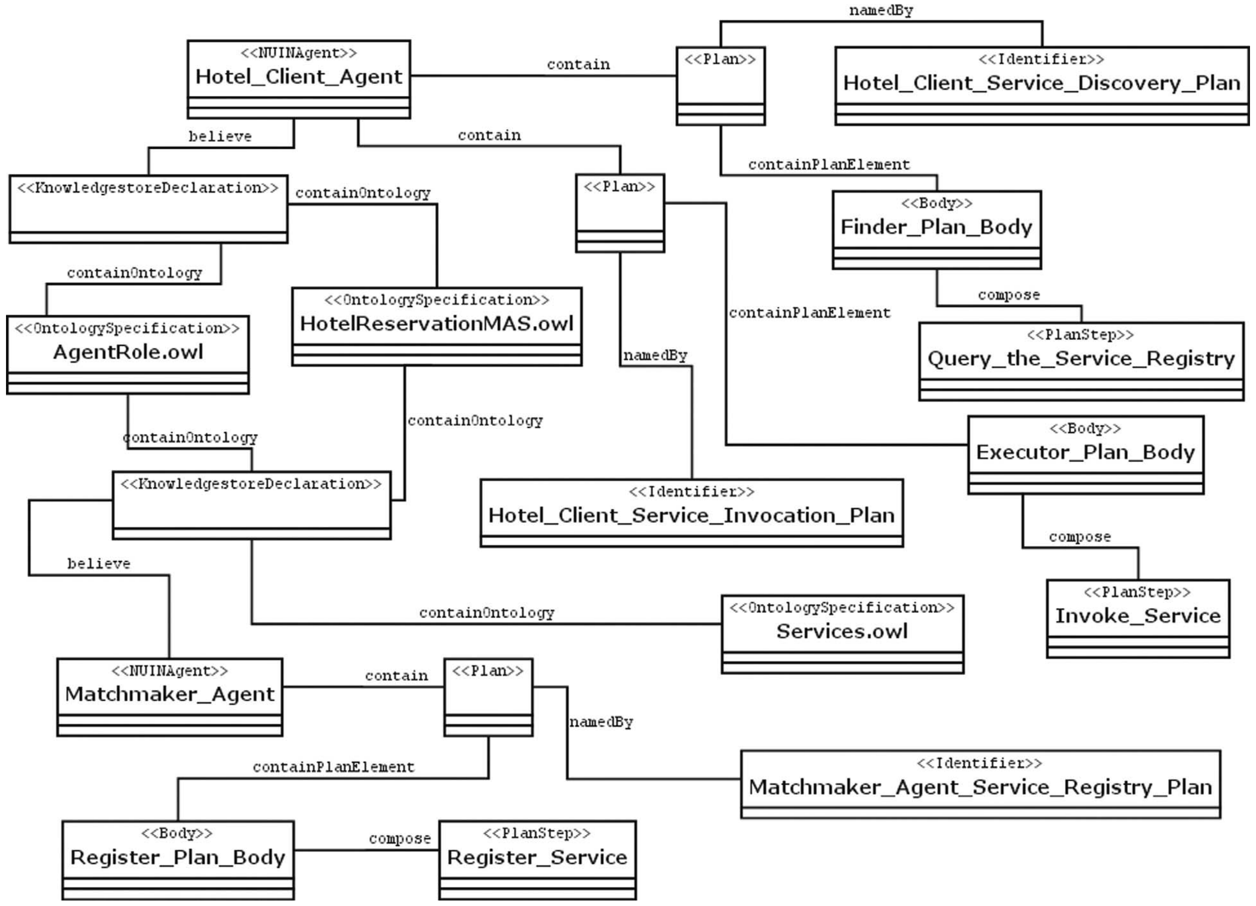


Fig. 9. The target model of the tourism MAS obtained after the PIM to Nuinscript PSM transformation.

The NUIN specific model of our tourism MAS, that is obtained after application of the whole model transformation process, is given in Fig. 9. This time our platform agents are represented with NUIN `Agent` instances and their plans as NUIN `Plans`. For each BDI agent, a `KnowledgestoreDeclaration` instance is created in order to store the beliefs of agents as `OntologySpecifications`. Plan instances of the source model are given with their identifiers and bodies according to the Nuinscript specifications. Behaviors defined in the source model are now given as `PlanStep` instances for the related agent plans (e.g. `Invoke_Service` for the `Hotel_Client_Service_Invocation_Plan`).

In fact, design and preparation of the source model conforming to our PIM is the only thing that a system developer needs to do to achieve working counterparts of the model in various PSMs according to our proposed MDD process. For example, considering the SEAGENT and NUIN environments, there is no need to rewrite metamodel definitions and abovementioned ATL rules in case of different system designs. Since metamodels and transformation rules are defined at the time of our MDD process is defined, developers only deal with the design of their MAS and provide that system's Ecore or XMI representation as an input to the introduced model transformation process.

7. Beyond the Model to Model Transformation: Code Generation for Semantic Web Enabled MASs

Although model to model transformation facilitates development of MASs by providing a higher abstraction level, it is not sufficient for real life implementations of such systems. System designers should inevitably write software codes for the systems. Proposed software development methodologies or development processes should provide a step in order to assist developers for code generation. Hence, in this section, the last step of our MDD process is discussed: Code generation from PSMs.

An Eclipse plugin for the generation of HTN-based agent plans was developed for the SEAGENT framework. The developed HTN[c] editor provides a Graphical User Interface (GUI) for the system developers to assist generation of software codes using the SEAGENT library. MAS developers prepare agent plan models in this GUI environment and the tool generates template codes for the planners of the agents. It is also possible to test designed plan model and execute generated codes inside a real SEAGENT platform via this Eclipse plugin.

Above introduced editor is based on the Eclipse Graphical Editing Framework (GEF). GEF allows developers to create a rich graphical editor from an existing application model.[57] The *Model*, *EditParts* and *Figures* are the three aspects of the GEF. These aspects are mapped to the widely-known Model-View-Controller

[c]This Eclipse plugin and also SEAGENT MAS development environment can be downloaded from the SEAGENT Project website at: http://seagent.ege.edu.tr.

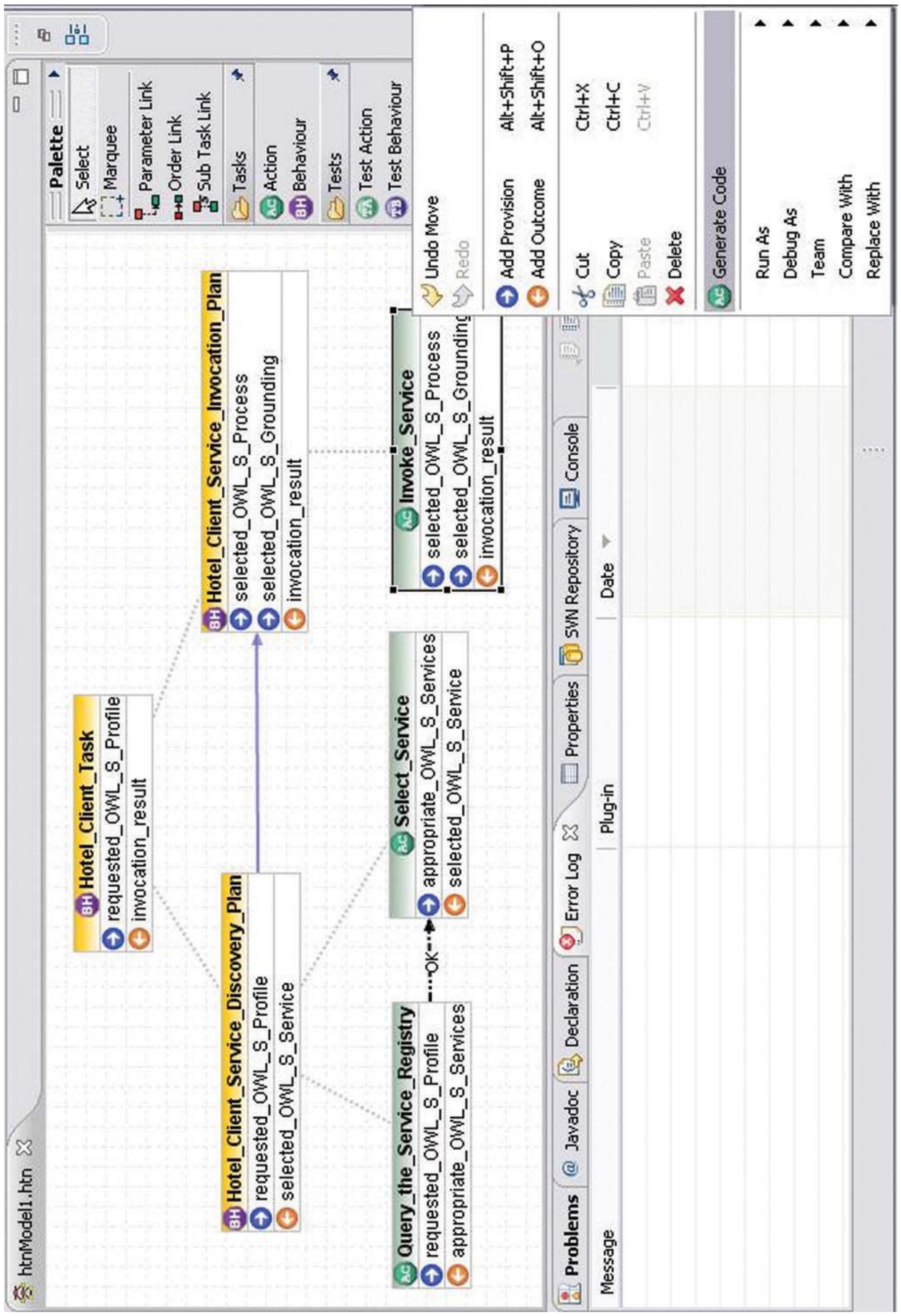


Fig. 10. The HTN Planner editor which provides plan code generation for agents working on the SEAGENT framework.

(MVC) architecture. GEF Model maps to "Model" of MVC and definition of model is left to user. GEF EditParts map to "Controller" of MVC and GEF figures map to "View" aspect of MVC. When we consider the HTN editor for the SEAGENT models, the GEF Model is composed of the SEAGENT planner meta-entities (Behavior, Action, Provision, Outcome, etc.). GEF EditParts provides the communication between model and figures. When a modification exists in the model, this modification is reflected to the corresponding figure element(s) via the EditParts or vice versa.

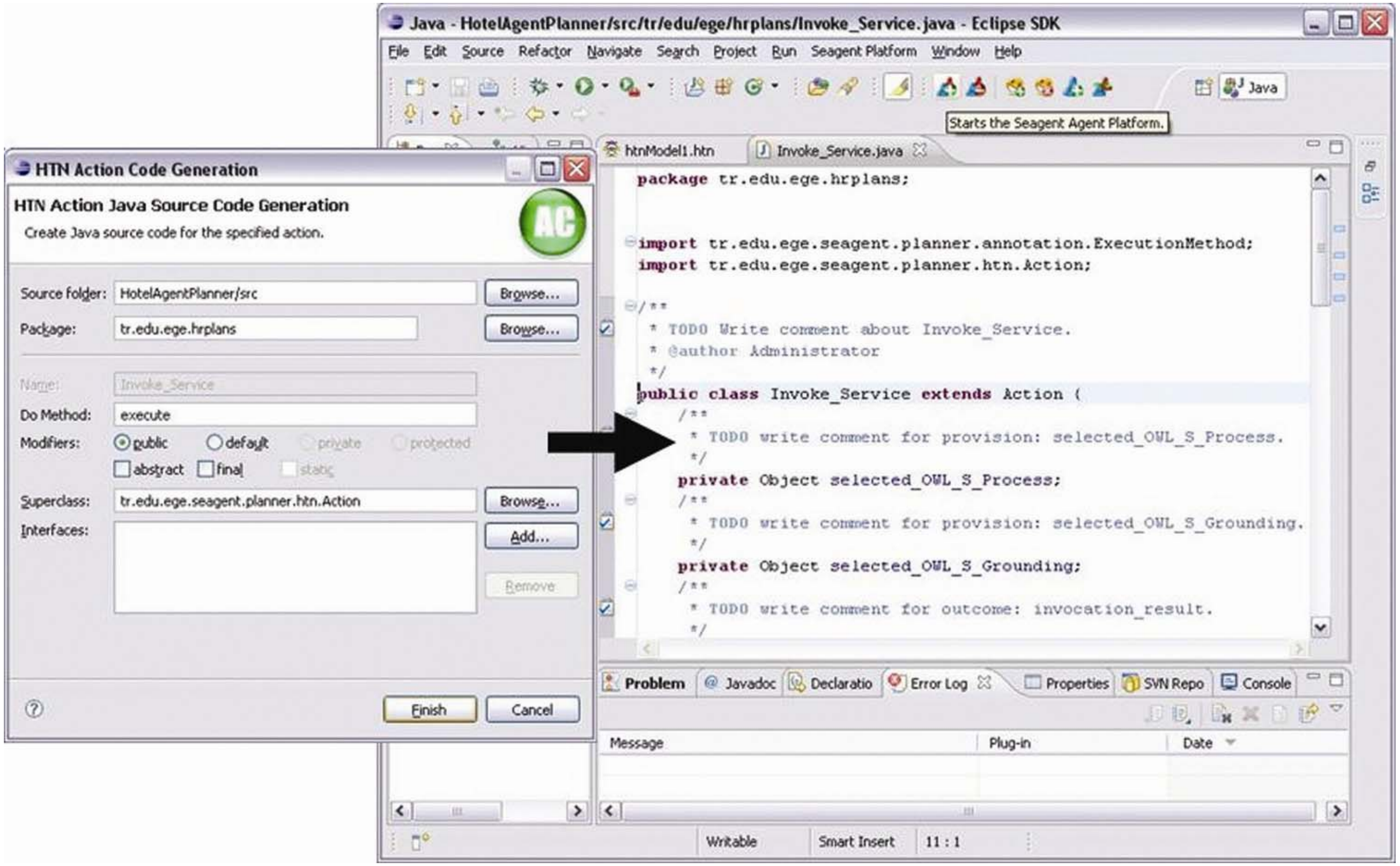
The editor employs the Abstract Syntax Tree (AST) and related parser in the Eclipse Java Development Tools (JDT) for automatic generation of HTN Plan codes. For example, there exists a component of the editor called `ActionCodeGen` which is the source code generator for HTN Action classes. It provides code generation as a compilation unit from a specified action code model and supports for writing generated code to a file or an output stream.

The screenshot given in Fig. 10 depicts generation of agent plan codes for our tourism MAS by using the editor. After visualization of the platform specific model for the Hotel Client Agent's plans, SEAGENT specific codes for each Action and Behavior components are prepared using the proper menus. It is also possible to modify Plan components (e.g. addition of new provisions and outcomes) using the editor palette.

In our example, we want to generate SEAGENT code for an Action (e.g. Invoke_Service in here). After setting specific properties of the generation process (e.g. source folder for the generated code file and name of the package) via a dialogue (shown in left side of Fig. 11), the code generator parses the model element, determines outcome, provision and any other associations and finally generates the template code for the related Action (shown in the right side of Fig. 11). The developer completes the code (e.g. by filling the body of the generated Action methods). Likewise, code generation of all actions are completed. According to the hierarchy, the next step is to generate codes of the agent behaviors which compose actions. Codes for actions of a behavior should be created first in order to generate the behavior codes, otherwise the editor warns the developer. After setting system specific properties again, the behavior codes are generated by taking into consideration provision — outcome relations of its sub-tasks (actions). Code generation for the upper levels of the agent plan is realized in the same manner.

The developer may also test execution of the generated plan in a SEAGENT MAS platform which can be initialized by using the SEAGENT Eclipse plugin.

Considering the NUIN framework, we provided a model to text transformation in order to generate *Nuinscript*s from the platform specific MAS models conforming to NUIN PSM. We implemented the related transformation by using the well-known MOF model to text transformation language called MOFScript.[58] MOFScript is a language specifically designed for the transformation of models into text files and it deals directly with metamodel descriptions (Ecore files) as input. Also, it provides a tool as an Eclipse plugin[59] in which MOFScript transformations can be written


Java - HotelAgentPlanner/src/tr/edu/ege/hrplans/Invoke_Service.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Seagent Platform Window Help
Java
Starts the Seagent Agent Platform.
htnModel1.htn
Invoke_Service.java
Problem
Javadoc
Declaratio
Error Log
Properties
SVN Repo
Console
Message
Plug-in
Date
Writable
Smart Insert
11 : 1
HTN Action Code Generation
HTN Action Java Source Code Generation
Create Java source code for the specified action.
AC
Browse...
Browse...
Browse...
Add...
Remove
Finish
Cancel

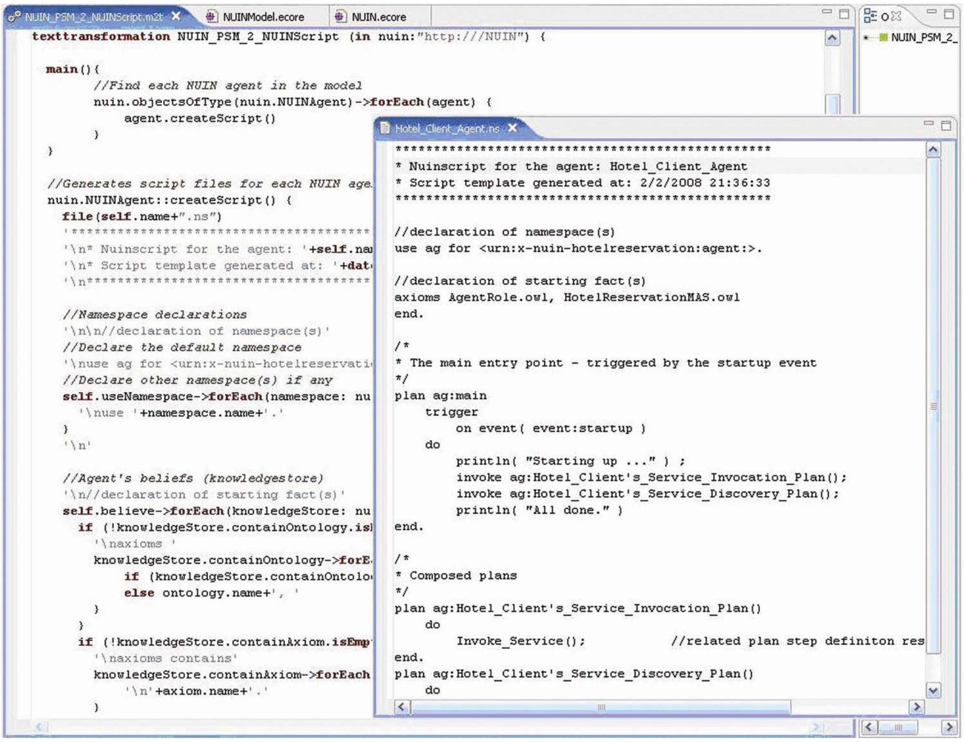
Source folder: HotelAgentPlanner/src
Package: tr.edu.ege.hrplans
Name: Invoke_Service
Do Method: execute
Modifiers: [✓ public] default private protected
abstract final static
Superclass: tr.edu.ege.seagent.planner.htn.Action
Interfaces:

```
package tr.edu.ege.hrplans;


import tr.edu.ege.seagent.planner.annotation.ExecutionMethod;
import tr.edu.ege.seagent.planner.htn.Action;

/**
 * TODO Write comment about Invoke_Service.
 * @author Administrator
 */
public class Invoke_Service extends Action (
    /**
     * TODO write comment for provision: selected_OWL_S_Process.
     */
    private Object selected_OWL_S_Process;
    /**
     * TODO write comment for provision: selected_OWL_S_Grounding.
     */
    private Object selected_OWL_S_Grounding;
    /**
     * TODO write comment for outcome: invocation_result.
     */
```

Fig. 11. Code generation for the SEAGENT Action class called Invoke_Service.



```
texttransformation NUIN_PSM_2_NUINScript (in nuin:"http:///NUIN") {

 main(){
       //Find each NUIN agent in the model
       nuin.objectsOfType(nuin.NUINAgent)->forEach(agent) {
           agent.createScript()
       }
 }

 //Generates script files for each NUIN age
 nuin.NUINAgent::createScript() {
   file(self.name+".ns")
   '****************************************
   '\n* Nuinscript for the agent: '+self.na
   '\n* Script template generated at: '+dat
   '\n**************************************

   //Namespace declarations
   '\n\n//declaration of namespace(s)'
   //Declare the default namespace
   '\nuse ag for <urn:x-nuin-hotelreservati
   //Declare other namespace(s) if any
   self.useNamespace->forEach(namespace: nu
     '\nuse '+namespace.name+'.'
   }
   '\n'

   //Agent's beliefs (knowledgestore)
   '\n//declaration of starting fact(s)'
   self.believe->forEach(knowledgeStore: nu
     if (!knowledgeStore.containOntology.is
       '\naxioms '
       knowledgeStore.containOntology->forE
           if (knowledgeStore.containOntolo
           else ontology.name+', '
       }
     }
     if (!knowledgeStore.containAxiom.isEmp
       '\naxioms contains'
       knowledgeStore.containAxiom->forEach
           '\n'+axiom.name+'.'
       }
```

```
************************************************
* Nuinscript for the agent: Hotel_Client_Agent
* Script template generated at: 2/2/2008 21:36:33
************************************************

//declaration of namespace(s)
use ag for <urn:x-nuin-hotelreservation:agent:>.

//declaration of starting fact(s)
axioms AgentRole.owl, HotelReservationMAS.owl
end.

/*
* The main entry point - triggered by the startup event
*/
plan ag:main
    trigger
        on event( event:startup )
    do
        println( "Starting up ..." ) ;
        invoke ag:Hotel_Client's_Service_Invocation_Plan();
        invoke ag:Hotel_Client's_Service_Discovery_Plan();
        println( "All done." )
end.

/*
* Composed plans
*/
plan ag:Hotel_Client's_Service_Invocation_Plan()
    do
        Invoke_Service();           //related plan step definiton res
end.
plan ag:Hotel_Client's_Service_Discovery_Plan()
    do
```

Fig. 12. The MOFScript transformation for the generation of Nuinscripts from NUIN PSMs and the template Nuinscript generated for the NUIN Agent called Hotel Client Agent.

and directly executed from the Eclipse environment. These advantages caused us to prefer MOFScript as our implementation language for NUIN model to Nuinscript code generations.

In Fig. 12, an excerpt from the prepared MOFScript transformation is given in the background. This text transformation uses the same metamodel of the Nuinscript in Ecore which is employed as one of the PSMs in our above discussed model to model transformations. The transformation, in here, reads a MAS model conforming to the metamodel of the NUIN and creates Nuinscripts for each NUIN Agent in the model. When the transformation is executed on a NUIN MAS model (again given in Ecore), each NUIN Agent instance is determined and related script files with ".ns" extensions are created.

Template scripts for each agent are generated in appropriate Nuinscript syntax by taking into consideration agent's namespace declarations, beliefs (given as knowledgestores in the model) and intentions (given as NUIN Plans with composition of Plan Steps in the model). A developer completes this generated template script for the exact implementation. For instance, we executed the above MOFScript transformation on the NUIN MAS model, which is the counterpart of our platform independent tourism MAS, obtained after the model to model transformation previously discussed in Sec. 6. After the execution of the model to text transformation, we achieved the template Nuinscript file for each NUIN Agent in the model. Again in Fig. 12, template Nuinscript generated for the Hotel Client Agent is listed.

8. Evaluation

Our MDA approach on Semantic Web enabled MAS development was used in a commercial project in which the design and the implementation of a tourism system based on the SEAGENT framework were realized. The project included adaptation of an existing hotel reservation system into the Semantic Web environment. The existing system was one of the products of a national software company called *ODEON*.[d] The company's main business domain is hotel management systems. The system had been previously based on the web service architecture and the project aimed to provide both semantic interfaces of the web services in use and to realize an online system in which software agents reserve hotel rooms on behalf of their human users.

The objectives were the evaluation of the approach's usability in a commercial project and optimization of the development process if required according to the user feedbacks. The developers used the metamodels and the model to model transformation process discussed in this study during the design phase of the project. Since code generation step was not included in our MDD process (and also supporting software tool was not implemented) at the time of that system development took place, it can be said that the process evaluation only considered generation of the platform specific MAS model.

At first, the developers determined the domain-based meta-constructs which could be considered as agents, roles, semantic services, etc. and associated those constructs with the elements of our PIM. Afterwards they designed the ecore model of the MAS conforming to the PIM. Finally, the target model of the system in the SEAGENT environment was obtained by applying the transformation. The output model was used both in the system modeling and further phases of the project.

The development team was composed of 5 software engineers with system development experiences ranging from 1 to 6 years. Following is the feedback on the application of our MDD process gained from the team. The feedback was obtained from the conversations with the team members.

First of all, the developers in general found the MAS architecture and the related metamodel especially helpful in the derivation of the real system's components. They also used a slightly different version of our PIM in documenting their software. The metamodel was found as an interesting extension of FIPA's ACSM with respect to agent planning. Some of the developers claimed that the approach in question justified the role of MDA in current software engineering practices and particularly for AOSE. Finally, the developers agreed that applying the transformation and

[d]ODEON Hotel Management Systems (http://www.myodeon.com).

generating the software model of the system as a result of the model transformation were helpful in their agile code development style due to our MDA approach's support for clarification and determination of entity relationships between agent and service structures.

However, some of the developers complained about the use of the ATL environment in model representation during model transformations. They found the source model representation in Ecore a bit confusing and error prone. They expressed the requirement of a tool in which the user can simply draw model elements and their associations according to the PIM. After the model is pictured, the tool automatically generates the codes for the Ecore representation of the model for the model transformation process. They also suffered from the problems encountered during the installation of the ATL plug-in. Although those complaints are directly related to the ATL and its environment, we also included them in the evaluation discussed here as one of the drawbacks of our approach. This is because model transformation is vital in our process and current application of the transformation is realized by the ADT within the process.

Furthermore some of the developers also suggested enrichment of the process by introducing interaction/collaboration diagrams in addition to the Semantic Web enabled MAS metamodel. Since the metamodel is both based on and extends the UML 2.0 superstructure; these developers expressed that inclusion of the appropriate UML diagram usage into the process would also support the metamodel. In fact, the developers agreed that the collaborations in question were already described implicitly within the heuristic rules prepared for the model transformation and hence that exposed another benefit of our approach. However, they also denoted that documenting of those relations explicitly within the corresponding UML diagrams would also strengthen the design process.

The application of the proposed development process within the abovementioned project also provided us to evaluate the approach as its owners. We examined that a MDA-based MAS development extended the learning and process adaptation curves for MAS developers. Like other MDD projects, adaptation of the developers — who are naturally accustomed to code centric development — into a model centric development also needed a great effort and took time in our case. Although some of the developers were a bit familiar with the MDD, the rest of them got newly acquainted to the metamodeling and model transformations. Hence, persuasion and adaptation of those developers was a big challenge and a change was required within the software development organization.

On the other hand, we experienced how the definition of reusable metamodels and model transformations facilitate system development. The metamodel reusability is particularly crucial. If the MAS metamodels are not designed in a reusable manner, we need to reorganize those metamodels and possibly we have to redesign the whole transformation process in every change of the application domain. This causes a more time consuming and an expensive development process when we compare it with the traditional code centric software development. Fortunately, our defined metamodel is designed by taking into account of a generic MAS architecture and it does not depend on the application domain. Hence, choosing such an architectural approach instead of the domain centric process definition provides reusability of the model transformation in various system developments with slight modifications. For instance, obtaining the platform specific model of a MAS working on the banking domain instead of the aforementioned tourism system, needs changes only in the source model, not in our architecture centric metamodels and the transformation process due to their domain independency.

9. Related Work

Recently, model driven approaches have been recognized and become one of the major research topics in AOSE community. As briefly discussed below, some of the studies intend to apply the whole MDD process for MAS development while some of them only utilize either metamodels or model transformation as needed. Also some studies focus on the conceptual MDA definitions and MDA-based MAS research directions e.g. Refs. 60–62.

Depke *et al.*[60] first introduced an approach to agent-oriented modeling based on UML notation and concepts of typed graph transformation systems. The theory of graph transformation also provides the mathematical background to formalize their study. Bauer and Odell discussed the use of UML 2.0 and MDA for agent-based systems in Ref. 61. They also evaluated which aspects of a MAS could be considered at the CIM and the PIM. The Cougaar MDA introduced in Ref. 62 provides a higher application composition for agent systems by elevating the composition level from individual components to domain level model specifications in order to generate software artifacts.

Jayatilleke *et al.*[63] described a conceptual framework of domain independent component types that can be used to formulate and modify an agent system. They provide a toolkit for their approach in Ref. 64 in order to make their approach consistent with MDD and use agent models to generate executable codes.

Defining meta-models and model transformations for MDD of MASs seems to be an emerging agent research track. Some of the studies also include interoperability of agent systems and service-oriented architectures (SOA). For instance, Zinnikus, *et al.*[65] proposed a new framework for rapid prototyping of SOAs. It contains a modeling part concerned with applying MDD techniques, a flexible communication platform for Web services and an autonomous agent part for negotiation and brokering in SOAs. Their framework follows the MDA approach and defines a PIM for SOA and PSM for BDI agents. Likewise in Ref. 66, the focus is on PIM to PSM transformation development in which a PIM for SOAs and a PSM for agent technologies are presented. The transformation mechanism introduced in that study has similarities with our transformation process in the way of defining metamodels, granting mappings and implementing the transformation. However, they provide a transformation from an agent-free SOA domain (PIM4SOA) to a MAS domain and deals with the interoperability between agents and web services. On the other hand, we define a transformation between two domains that both include agents as the first class entities and we support the interoperability between autonomous agents and semantic web services.

The study defined in Ref. 67 applies the transformation pattern of MDA which is previously depicted in Fig. 1. Perini and Susi[67] use TEFKAT model transformation language[19] to implement the transformation process in automating conversions from Tropos[4] structures to UML models. The Malaca UML Profile discussed in Ref. 68 provides the stereotypes necessary to create Malaca models on UML modeling tools. In their transformation pattern, Tropos design model and Malaca model are considered as PIM and PSM respectively.

While the abovementioned studies cover metamodeling and model transformation for agent systems, some of the agent development methodology owners reorganize their approaches according to the MDA. For example, in Ref. 13, Pavon and his friends reformulate their agent-oriented methodology called INGENIAS in terms of the MDD paradigm. This reformulation increases the relevance of the model creation, definition and transformation in the context of MASs. A similar MAS methodology revision is discussed in Ref. 69. Ideas and standards from MDA are adopted in refining the modeling process algorithm and building tools of Tropos methodology[4] within this study. Likewise, the study presented in Ref. 70 aims to add a MDD phase to ADELFE methodology[71] according to adaptive MAS paradigm by considering two adaptation levels called functional and operational. The functional level is application dependent and close to the decision process of agents while operational level is related to elementary skills of agents.

Regarding all of the above studies, we can conclude that the current application of the MDD on MAS development is in its preliminary phase. Neither a complete MDD process nor a common MAS metamodel has been developed. On the other hand, Semantic Web[11] technology and its required constructs on MASs are not supported within those studies as mentioned before. We believe this shortage in question is crucial when development of future MASs is considered. Therefore providing a Semantic Web enabled MDD process for MAS development is the key difference between our study and those previous studies.

On the other hand, the interactions between agents and semantic web services should be considered during environment modeling. The study depicted in Ref. 41 is a good example to apply MDA techniques to semantic web service development. Grønmo *et al.*[41] propose a UML profile for semantic web services that enables the use of high-level graphical models as an integration platform for semantic web services. This UML profile enables generating different platform dependent semantic web service language documents like OWL-S[27] and WSMO[28] documents from the PIM. As discussed in related sections of this paper, we utilize this UML profile also in our PIM. Within the same perspective, Pahl[72] extends the use of web ontology languages in MDD frameworks by presenting a layered, ontology transformation-based semantic modeling approach for software services. MDD is adapted to service architecture using ontology technology as the integrating tool. Although those studies provide brilliant model driven approaches for semantic web service development, they do not consider the development of a system which integrates agents and semantic web services as directly supported in our work.

10. Conclusion and Future Work

The constructs and development steps for MDD of Semantic Web enabled MASs are discussed in this study. We first define an architecture for Semantic Web enabled MASs and then provide a metamodel which consists of the first class meta-entities derived from this architecture in order to model such MASs. We believe that the metamodel in here helps to bridge the gap of modeling agent and Semantic Web constructs in a single environment by defining entities of a Semantic Web enabled MAS at first time.

The study herein also presents a whole model transformation process in which source and target metamodels, entity mappings and implementation of the transformations for two real MAS frameworks called SEAGENT[17] and NUIN[43] are all included. The realization platform in here may differ. In that case we only need to prepare metamodel of this new platform and utilize this metamodel as the PSM in our MDD process. Model to text transformation step of our process is also discussed by exemplifying Java code and Nuinscript generations for the SEAGENT and the NUIN PSMs respectively.

Utilization of the process in a real commercial project shows the practical relevance of our approach. Evaluation of the process within this context also helps us to determine future work for our study. We aim to support the MAS metamodel with related collaboration diagrams for the entities considering the interaction aspect of the study. The first attempt will be the definition of the entity interactions by employing the Agent UMLs (AUML) sequence, activity and collaboration diagrams and statecharts. AUML[73] provides agent-based extensions for UMLs package, template, sequence diagrams, activity diagrams and class diagrams. As mentioned in the evaluation section, the appropriate inclusion of such diagrams seems easy since our metamodel is already based on and extends UML 2.0 superstructure. However, AUML does not include Semantic Web components and probably we will need to bring new extensions for the AUMLs related interaction diagrams.

Requirement engineering for the business domain and traceability of system specifications from CIM to PIM composes an important step of our MDD process. However, the present state of this process step is immature and hence it is not covered in this paper in order to provide consistency. Currently, we are working on reformulation of this step by taking into account of specification, validation and formatting of system requirements as a CIM.

Meanwhile, we also intend to improve mappings and model transformations introduced in the study. These improvement efforts will cover the issues like elaborating mappings in entity attribute level and clarifying input/output and precondition/effect representations of semantic web service entities on the model.

On the other hand, we believe that a tool support for our MDD process in model generation will ease preparation of the metamodels for the transformations. Therefore, our aim is to provide another software tool with a GUI in which users can draw their model elements and specify related entity associations visually. Upon completion of the model preparation in graphics, the tool automatically generates corresponding KM3 representation or EMF encodings of the model which will be used by the ADT during the model transformation.

The improvement of the code generation tool for the SEAGENT framework introduced in this paper is also an important future work. As discussed in Sec. 7, the editor for the code generation is currently GEF based. There exists an ongoing study to provide integration of the EMF into the editor infrastructure by utilizing Eclipse Graphical Modeling Framework (GMF)[74] instead of GEF. One important benefit of this new release will be the full support for Ecore models and automatic integration with the ATL environment.

Acknowledgments

This work is partially funded by The Scientific and Technological Research Council of Turkey (TUBITAK) Electric, Electronic and Informatics Research Group (EEEAG) under grant 106E008 and TUBITAK National Scholarship Program for PhD Students.

The authors wish to thank the software development team of the SEAGENT laboratory and the ODEON Co. for their great effort and collaboration in this study. The authors also thank Dr. E. Turhan Tunali for his helpful comments on formatting the early drafts of the paper. Finally, the authors thank anonymous reviewers for their accurate comments on the initial version of the paper. We were able to improve both our study and the paper significantly by taking their critical comments into account.

References

1. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (Pearson Education, USA, 2003).
2. K. Sycara, Multiagent systems, *AI Magazine* **19**(4) (1998) 79–92.
3. M. Wooldridge, N. R. Jennings and D. Kinny, The Gaia methodology for agent-oriented analysis and design, *Autonomous Agents and Multi-Agent Systems* **3**(3) (2000) 285–312.
4. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia and J. Mylopoulos, Tropos: An agent-oriented software development methodology, *Autonomous Agents and Multi-Agent Systems* **8**(3) (2004) 203–236.
5. S. A. DeLoach, M. F. Wood and C. H. Sparkman, Multiagent systems engineering, *Int. J. Software Engineering and Knowledge Engineering* **11**(3) (2001) 231–258.

6. A. Omicini, SODA: Societies and infrastructures in the analysis and design of agent-based systems, *Lecture Notes in Computer Science* **1957** (2000) 185–193.
7. F. Bergenti, M.-P. Gleizes and F. Zambonelli, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook* (Kluwer Academic Publishers, Boston, USA, 2004).
8. C. Bernon, M. Cossentino, M.-P. Gleizes, P. Turci and F. Zambonelli, A study of some multi-agent meta-models, *Lecture Notes in Computer Science* **3382** (2005) 62–77.
9. A. Molesini, E. Denti and A. Omicini, MAS Meta-models on Test: UML vs OPM in the SODA Case Study, *Lecture Notes in Artificial Intelligence* **3690** (2005) 163–172.
10. FIPA Modeling TC, Agent Class Superstructure Metamodel (2004), http://www.omg.org/docs/agent/04-12-02.pdf
11. T. Berners-Lee, J. Hendler and O. Lassila, The semantic web, *Scientific American* **284**(5) (2001) 34–43.
12. B. Selic, The pragmatics of model-driven development, *IEEE Software* **20** (2003) 19–25.
13. J. Pavon, J. Gomez and R. Fuentes, Model driven development of multi-agent systems, *Lecture Notes in Computer Science* **4066** (2006) 284–298.
14. S. Sendall and W. Kozaczynski, Model transformation — the heart and soul of model-driven software development, *IEEE Software* **20** (2003) 42–45.
15. OMG, MDA Guide Version 1.0.1., OMG Document Number: omg/2003-06-01 (2003), http://www.omg.org/docs/omg/03-06-01.pdf
16. OMG, Meta Object Facility (MOF) Specification, OMG Document Number: AD/97-08-14 (1997), http://www.omg.org/docs/ad/97-08-14.pdf
17. O. Dikenelli, R. C. Erdur, O. Gumus, E. E. Ekinci, O. Gurcan, G. Kardas, I. Seylan and A. M. Tiryaki, SEAGENT: A platform for developing semantic web based multi agent systems, in *Proc. the Fourth Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2005)* (Utrecht, the Netherlands, 2005) (ACM Press), pp. 1271–1272.
18. A. Agrawal, G. Karsai, S. Neema, F. Shi and A. Vizhanyo, The design of a language for model transformations, *Software and Systems Modeling* **5**(3) (2006) 261–288.
19. K. Duddy, A. Gerber, M. Lawley, K. Raymond and J. Steel, Model transformation: A declarative, reusable patterns approach, in *Proc. Seventh IEEE Int. Enterprise Distributed Object Computing Conf. (EDOC'03)* (IEEE Computer Society 2003), pp. 174–185.
20. F. Jouault and I. Kurtev, Transforming models with ATL, *Lecture Notes in Computer Science* **3844** (2006) 128–138.
21. A. Kalnins, J. Barzdins and E. Celms, Model transformation language MOLA, *Lecture Notes in Computer Science* **3599** (2005) 62–76.
22. G. Kardas, A. Goknil, O. Dikenelli and N. Y. Topaloglu, Metamodeling of semantic web enabled multiagent systems, in *Proc. the Multiagent Systems and Software Architecture (MASSA), Special Track at Net.ObjectDays - NODe 2006* (Erfurt, Germany, 2006), pp. 79–86.
23. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, *Documenting Software Architectures: Views and Beyond* (Pearson Education, USA, 2003).
24. FIPA, Foundation for Intelligent Physical Agents (FIPA) Specifications (2002), http://www.fipa.org
25. G. Kardas, Ö. Gümüs and O. Dikenelli, Applying semantic capability matching into directory service structures of multi agent systems, *Lecture Notes in Computer Science* **3733** (2005) 452–461.

26. K. Sycara, M. Paolucci, A. Ankolekar and N. Srinivasan, Automated discovery, interaction and composition of Semantic Web Services, *J. Web Semantics* **1** (2003) 27–46.
27. OWL-S Coalition, OWL-S: Semantic Markup for Web Services (2004), http://www.daml.org/services/owl-s/1.1/overview/
28. WSMO Working Group, Web Service Modeling Ontology (WSMO) (2005), http://www.wsmo.org/index.html
29. M. Paolucci, T. Kawamura, T. R. Payne and K. Sycara, Semantic matching of web services capabilities, *Lecture Notes in Computer Science* **2342** (2002) 333–347.
30. L. Li and I. Horrocks, A software framework for matchmaking based on semantic web technology, in *Proc. the WWW'2003* (Budapest, Hungary, 2003), pp. 331–339.
31. JENA, A semantic web framework for Java (2003), http://jena.sourceforge.net.
32. O. Dikenelli, R. C. Erdur, G. Kardas, O. Gümüs, I. Seylan, O. Gurcan, A. M. Tiryaki and E. E. Ekinci, Developing multi-agent systems on semantic web environment using SEAGENT platform, *Lecture Notes in Artificial Intelligence* **3963** (2006) 1–13.
33. M. Williamson, K. Decker and K. Sycara, Unified information and control flow in hierarchical task networks, in *Proc. the AAAI-96 Workshop* (California, USA, 1996), pp. 142–150.
34. I. Dickinson and M. Wooldridge, Agents are not (just) web services: Considering BDI agents and web services, in *Proc. the Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE 2005)* (Utrecht, the Netherlands, 2005).
35. OMG, Object Management Group UML 2.0 Superstructure Specification (2004), http://www.omg.org/technology/documents/formal/uml.htm.
36. D. Djuric, MDA-based Ontology Infrastructure, *Computer Science Information Systems* **1**(1) (2004) 91–116.
37. J. Ferber and O. Gutknecht, A meta-model for the analysis and design of organizations in multi-agent systems, in *Proc. the Third International Conference on Multi-Agent Systems* (IEEE Computer Society, 1998), pp. 128–135.
38. J. Odell, M. Nodine and R. Levy, A metamodel for agents, roles and groups, *Lecture Notes in Computer Science* **3382** (2005) 78–92.
39. G. Kardas, A. Goknil, O. Dikenelli and N. Y. Topaloglu, Modeling the interaction between semantic agents and semantic web services using MDA approach, *Lecture Notes in Artificial Intelligence* **4457** (2007) 209–228.
40. W3C, World Wide Web Consortium Resource Description Framework (RDF) (2004), http://www.w3.org/RDF/.
41. R. Grønmo, M. C. Jaeger and H. Hoff, Transformations between UML and OWL-S, *Lecture Notes in Computer Science* **3748** (2005) 269–283.
42. G. Kardas, A. Goknil, O. Dikenelli and N. Y. Topaloglu, Model transformation for model driven development of semantic web enabled multi-agent systems, *Lecture Notes in Artificial Intelligence* **4687** (2007) 13–24.
43. I. Dickinson and M. Wooldridge, Towards practical reasoning agents for the semantic web, in *Proc. the Second Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2003)* Melbourne, Australia (ACM Press, 2003), pp. 827–834.
44. J. R. Graham, K. S. Decker and M. Mersic, DECAF — A flexible multi-agent systems infrastructure, *Autonomous Agents and Multi-Agent Systems* **7**(1-2) (2003) 7–27.
45. F. Bellifemine, A. Poggi and G. Rimassa, Developing multi-agent systems with a FIPA-compliant agent framework, *Software Practice and Experience* **31** (2001) 103–128.
46. K. Sycara, M. Paolucci, M. Van Velsen and J. Giampapa, The RETSINA MAS Infrastructure, *Autonomous Agents and Multi-Agent Systems* **7**(1-2) (2003) 29–48.
47. D. L. McGuiness and F. van Harmelen, OWL Web Ontology Language Overview (2004), http://www.w3.org/TR/owl-features/.

48. Ö. Gürcan, G. Kardas, Ö. Gümüs, E. E. Ekinci and O. Dikenelli, An MAS infrastructure for implementing SWSA-based semantic services, *Lecture Notes in Computer Science* **4504** (2007) 118–131.
49. A. Rao and M. Georgeff, BDI Agents: From theory to practice, in *Proc. the First International Conference on Multi-Agent Systems* (ICMAS-95), San Francisco, USA (1995), pp. 312–319.
50. I. Dickinson, *BDI Agents and the Semantic Web: Developing User-Facing Autonomous Applications*, PhD Thesis, University of Liverpool (2006).
51. NUIN, NUIN Framework (2006), http://www.nuin.org.
52. J. Warmer and A. Kleppe, *Object Constraint Language: The Getting Your Models Ready for MDA* (Pearson Education, USA, 2003).
53. Eclipse Community, Eclipse Open Development Platform (2003), http://www.eclipse.org.
54. ATLAS Group, ATL User manual (2006), http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf.
55. Eclipse Community, Eclipse Modeling Framework (2004), http://www.eclipse.org/emf.
56. F. Jouault and J. Bezivin, KM3: A DSL for metamodel specification, *Lecture Notes in Computer Science* **4037** (2006) 171–185.
57. Eclipse Community, Graphical Editing Framework (2006), http://www.eclipse.org/gef.
58. J. Oldevik, T. Neple, R. Grønmo, J. Aagedal and A. J. Berre, Toward standardised model to text transformations, *Lecture Notes in Computer Science* **3748** (2005) 239–253.
59. Eclipse Community, MOFScript model to text transformation language and tool (2005), http://www.eclipse.org/gmt/mofscript/.
60. R. Depke, R. Heckel and J. M. Küster, Agent-oriented modeling with graph transformations, *Lecture Notes in Computer Science* **1957** (2001) 105–120.
61. B. Bauer and J. Odell, UML 2.0 and Agents: How to build agent-based systems with the new UML standard, *Engineering Applications of Artificial Intelligence* **18**(2) (2005) 141–157.
62. D. Gracanin, H. L. Singh, S. A. Bohner and M. G. Hinchey, Model-driven architecture for agent-based systems, *Lecture Notes in Artificial Intelligence* **3228** (2005) 249–261.
63. G. B. Jayatilleke, L. Padgham and M. Winikoff, Towards a component-based development framework for agents, *Lecture Notes in Computer Science* **3187** (2004) 183–197.
64. G. B. Jayatilleke, L. Padgham and M. Winikoff, Evaluating a model driven development toolkit for domain experts to modify agent based systems, *Lecture Notes in Computer Science* **4405** (2007) 190–207.
65. I. Zinnikus, G. Benguria, B. Elvesæter, K. Fischer and J. Vayssière, A model driven approach to agent-based service-oriented architecture, *Lecture Notes in Artificial Intelligence* **4196** (2006) 110–122.
66. C. Hahn, C. Madrigal-Mora, K. Fischer, B. Elvesæter, AJ. Berre and I. Zinnikus, Meta-models, models, and model transformations: Towards interoperable agents, *Lecture Notes in Artificial Intelligence* **4196** (2006) 123–134.
67. A. Perini and A. Susi, Automating model transformations in agent-oriented modeling, *Lecture Notes in Computer Science* **3950** (2006) 167–178.
68. M. Amor, L. Fuentes and A. Vallecillo, Bridging the gap between agent-oriented design and implementation using MDA, *Lecture Notes in Computer Science* **3382** (2004) 93–108.
69. L. Penserini, A. Perini, A. Susi and J. Mylopoulos, From stakeholder intentions to software agent implementations, *Lecture Notes in Computer Science* **4001** (2006) 465–479.

70. S. Rougemaille, F. Migeon, C. Maurel and M.-P. Gleizes, Model driven engineering for designing adaptive multi-agent systems, in *Proc. the Eighth Annual International Workshop on Engineering Societies in the Agents World (ESAW 2007)* (Athens, Greece, 2007).
71. C. Bernon, M.-P. Gleizes, S. Peyruqueou and G. Picard, ADELFE: A methodology for adaptive multi-agent systems engineering, *Lecture Notes in Artificial Intelligence* **2577** (2003) 156–169.
72. C. Pahl, Semantic model-driven architecting of service-based software systems, *Information and Software Technology* **49** (2007) 838–850.
73. B. Bauer, J. P. Muller and J. Odell, Agent UML: A formalism for specifying multiagent software systems, *Int. J. Software Engineering and Knowledge Engineering* **11**(3) (2001) 207–230.
74. Eclipse Community, Eclipse Graphical Modeling Framework (2007), http://www.eclipse.org/gmf/.