6-2007

Agent Design of SmArt License Management System Using Gaia Methodology

Qian Zhao
University of Western Ontario, qianzhao@csd.uwo.ca

Yu Zhou
University of Western Ontario, yuzhou@alumni.uwo.ca

Mark Perry
University of Western Ontario, mperry@uwo.ca

Agent Design of SmArt License Management System Using Gaia Methodology

Qian Zhao, Yu Zhou, Mark Perry
Department of Computer Science
University of Western Ontario
London, ON, Canada
Email: {qianzhao@csd|yuzhou@alumni|markp@csd}.uwo.ca

***Abstract*— Modern software services and data centers require a license management system to regulate the agreements that have been reached between subscriber and provider. License management helps to track usage and protect service from abuse. License agreements provide the basis for enforcement and regulation. The automation of license agreements is desired by providers and subscribers to improve transaction efficiency, give flexibility, and minimize unwanted cost.**

We have proposed a framework, called SmArt (Semantic Agreement) system, that enables agreement automation in the autonomic computing context using ontology and agent technologies. This paper applies the SmArt system to the domain of license management and presents its agent design with Gaia methodology.

I. Introduction

To increase revenue and give more flexibility to users, the "on demand" concept has been adopted by the Information Technology (IT) industry. Analogous to their counterparts in other industries, such as cable, phone and hydro, IT vendors provide their customers the possibility of leasing software, storage, and/or hardware at a relatively low expense or even on a pay-per-use basis. Setting up a data center is a common example in IT leasing. IT vendors and customers negotiate, and eventually agree on how to use an IT service provided by the data center. The result of negotiation is then articulated in an agreement and signed by both parties. As in a traditional leasing/subscribing scenario, it is easy to see that it is a basic requirement for the management of a data center to include license control and apply it effectively and accurately. The cost of skilled personnel in this field makes automation of the interpretation and enforcement of these license agreements a very attractive proposition.

License agreement contains license terms and provisions of a varying complexity. During the automation of license agreements, one key challenge is: how to make automatic management decision based on license agreements?

The SmArt framework in [1] leverages ontology concepts to represent license agreement and other shared knowledge. Ontology is a set of *subject-property-object* tuples, where both *subject* and *object* are concepts (also called *vocabulary* or *class*) and *property* (or *predicate*) relates the subject and object together. As all necessary concepts of the domain are defined and their relations are shown as tuples, ontology discloses semantics of these concepts so that computer programs can "understand" the meaning of knowledge written in this way. By combining ontology with agent technologies, agents understand the ontology-based domain knowledge, including that of license agreements, and carry out management tasks autonomously based on what they understand, with little or no interruption from human. As we can see, the SmArt framework is able to meet the key challenges of agreement automation and thus can be adopted to build a license management system for data centers. This paper focuses on using the Gaia [2] methodology for the agent analysis and design of a SmArt license management system and shows how the task ontology guides the agent design.

The rest of this paper is organized as follows: Section 2 discusses the ontology of license management, focusing on the task ontology that expresses the workflow of reusable tasks using a set of associated relations between concepts; Section 3 addresses the agent design and analysis using Gaia; Section 4 briefly talks about the implementation; and the last section wraps up this paper with the conclusion and suggested future work.

II. Ontology of License Management

In the SmArt framework, ontology is used to represent knowledge that can be shared among multiple agents and/or programs. One important part of knowledge is modeling concepts in the license control domain as domain ontology. Domain ontology is the result of taxonomy. There are five basic concepts in the agreement automation as discussed in [1]: Agreement, Service, Resource, Constraint, and Request. Their definitions and other related concepts in license management scenario are discussed in [3]. To better explain the agent design and its foundation — task ontology, we summarize the concept of LA (License Agreement) here briefly in BNF (Backus-Naur Form).

<LA>::= <Agreement>
<Agreement>::= <Service><Customer><Vendor>
{<ProvisionItem>}
<ProvisionItem>::= <LATerm> | ... //ProvisionItem derives different agreement terms
<LATerm>::= <LATerm><AggregatingOperator><LATerm>|
<LicenseTerm>
<AggregatingOperator>::= *and* | *or*
<LicenseTerm>::= <LicenseMetrics>[*then* {<Action>} [*else* {<Action>}]]

Another important part of knowledge is the task ontology. Task ontology expresses reusable workflows of license management processes and is the foundation of the agent design. When applying SmArt framework in the domain of license management, the SmArt-based license agreement automation has four task ontologies that model four subsystems respectively: Service Management, Service Subscription, Service Configuration and Service Observation [1][3].

- Service Management works as the interface between customers and the SmArt-based system. It dispatches different customer request to corresponding functional component inside the system.
- Service Subscription handles service subscription request from customers and updates SKC (SmArt Knowledge Core) every time a subscription is made.
- Service Configuration responses to a user's request of service. It verifies the request against relate license agreement from SKC before bringing the service into the ready-to-serve state.
- Service Observation monitors a service, analyzes its usage data and controls the service according to the related license agreement from SKC, preventing service abuse and generating usage report.

The example in Figure 1 shows the task ontology for Service Configuration. This task ontology is built on relations between concept pairs, in the form of *subject-property-object* tuples, and models the workflow of the configuration process after a service request is received and dispatched to the configuration subsystem.

Some relations can be further broken down to show more details at lower level processes, bringing a hierarchy of task ontologies. However, here we stop at this level of granularity to avoid losing task generality, and to also give developers the freedom to choose how each property-mapping-its-two-concepts can work.

III. Agent Design

AI research has shown agent technologies closely integrating with ontology [4][5]. As agreement automation requires a certain level of intelligence and AI practices already show some machine intelligence can be created by combining agent and ontology together, it is natural that this research adopts agent technologies to fulfill the implementation.

This section uses the Gaia methodology to analyze each task ontology that represents the workflow of a subsystem of SmArt system, and carry out the agent design. We use the service configuration subsystem as the example throughout this analysis and design to illustrate how agents cooperate with task ontology of the subsystem it works for.

Following Gaia's notation, this section starts with the role model definition and then defines the associated interaction model. These two models form the analysis phase of Gaia methodology, which is then followed by the design phase that generates the agent model, the service model and the acquaintance model. In short, the analysis phase is from roles' view while the design phase is from agents' view.

A. Role Model

Role Model introduces all necessary roles existing in the service configuration. Each is expressed in a role schema where its individual properties such as protocols and activities are presented. Among these properties, protocols are actions that contain interactions with other roles, and activities, indicated by the underline, are actions where no interaction with other roles is involved. Permissions specify the resources that a role has access to. Though some of these resources can only be read without modification, others can be changed and some are new resources generated during a role's life cycle. Responsibilities define what a role can do. It has two aspects: *liveness* tells what a role will do, using an expression where '$\cdot$' means happening in sequence, '$|$' means either one before or after '$|$' happens but not both, and '[ ]' means actions inside are optional; and *safety* shows preconditions that have to be maintained to make a role alive.

The task ontology of service configuration shown in Figure 1 contains 9 task concepts of the subsystem, which correspond to 9 roles. As a result, properties between two task concepts are going to be mapped to the initiating role's protocols and properties between task concept and domain concept are data processing that could be included as part of the task concept's activities or protocols. The role of SvcConfigurer works as the mediator and carries out a major part of the service configuration task ontology. This subsection uses it as an example. The role schema of SvcConfigurer is shown in Table I.

Other roles are:

- AgreementRegistry: This role manages agreements represented in the domain ontology and is able to register new agreements.
- AgreementParser: This role parses an agreement to extract the license agreement term (LATerm) and verifies a request before starting up the requested service. Note this role is generalized so that it is able to derive parsers for various agreement types.
- LicenseTermDispatcher: This role is responsible for correlating/consolidating responses of an aggregated LATerm associated with the service, and propagating actions in its consisting atomic terms (LicenseTerm).
- LicenseTermVerifier: This is an generalized role that verifies a license term. Different license terms may have their perticular verifiers derived from this one. It provides the potential of accommodating more license terms that emerge later.
- ServiceRegistry: This role manages services, maintain their state and is able to register new services.
- ResourceMgr: This role is in charge of resource management, including registration, retrieval, assigning, returning and information updating of resource.
- ConstraintsVerifier: This role verifies if using a resource is acceptable, or if adding a consisting resource to a service/agreement is acceptable. The latter situation occurs during the service subscription when the customer wants

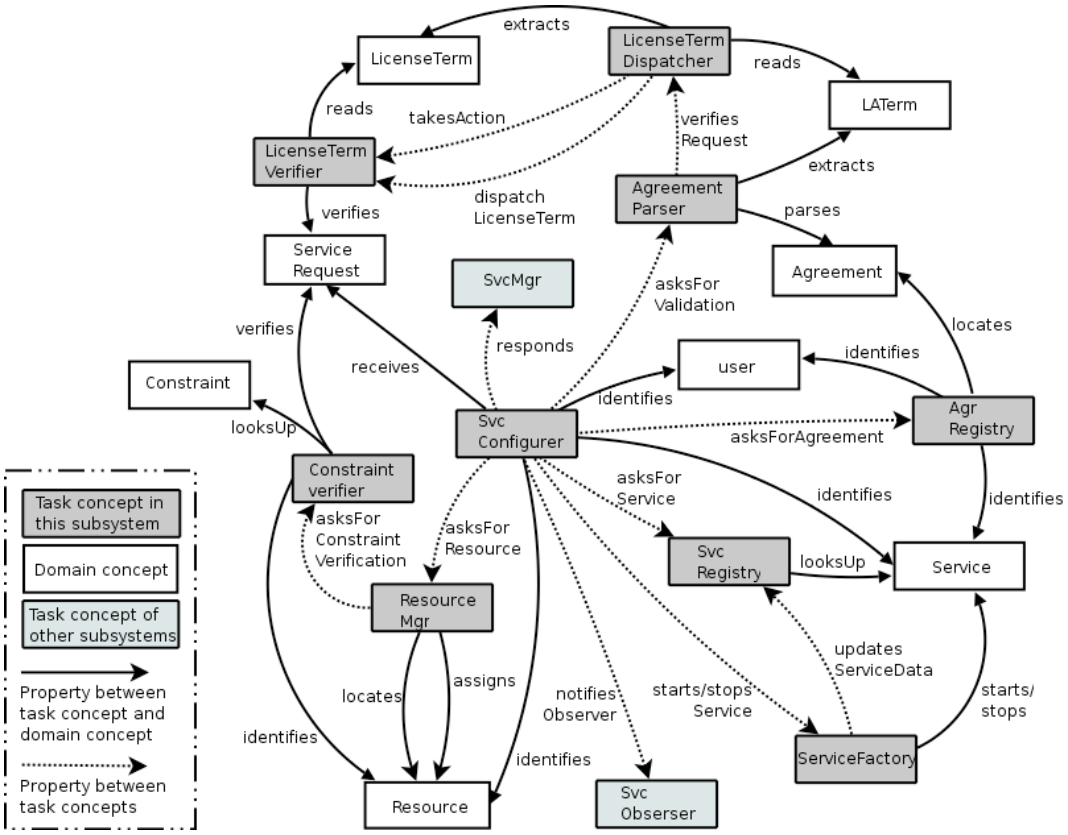

Fig. 1. The task ontology of service configuration

TABLE I
SCHEMA FOR ROLE SVCCONFIGURER

Role Schema: SvcConfigurer
Description: This role accepts the request of service and brings the service to the state of 'ready' at the end of processing.
Protocols and Activities: LocateAgreement, VerifyRequest, LocateService, RequestResource, StartService, NotifyObserver, RespondRequest, StopService
Permissions: reads *serviceRequest* // user wants to use a service *serviceRequestIsLegal* // the request to service is legal generates *serviceIsReady* // the requested service is ready to serve requests
Responsibilities Liveness: SvcConfigurer = (LocateAgreement · VerifyRequest · [LocateService · [RequestResource] · StartService · NotifyObserver] · RespondRequest) | (StopService · [RespondRequest]) Safety: •service.isLegal() = true

to build a new service or a new agreement, of which the details are not covered in this research, although a stub activity (VerifyAdding) is created to make it possible to accommodate this scenario later.

- ServiceFactory: This role controls the life cycle of services. It has the knowledge of how to start or stop a service.

B. Interaction Model

Interaction model explains the interaction between roles introduced in the role model. Related information like initiator, responder, input and output of an interaction is presented in this model. Here we focus on the interactions of role SvcConfigurer, shown in Figure 2, which happens during the service configuration. For each interaction, the top block shows its purpose. The left of the middle level block is the initiator of an interaction while the right one is the responder. The

bottom block briefly addresses the process being performed in the interaction. The line between top and middle blocks indicates the input and the one between middle and bottom blocks indicates the output. Some interactions will trigger other interactions and this triggering is indicated by an arrow.

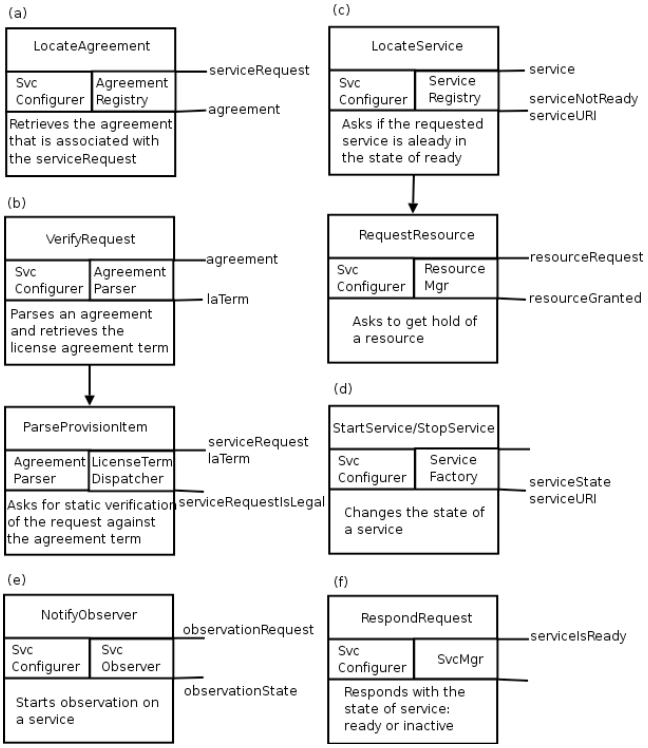

Fig. 2. Interaction of SvcConfigurer role: (a) LocateAgreement, (b) VerifyRequest, (c) RequestResource, (d) StartService/StopService, (e) NotifyObserver, (f) RespondRequest

C. Agent Model

As the first step in the design phase, the agent model shows all necessary agent types for the configuration subsystem and the roles they assume. Normally it is a one-to-one relation between a role and an agent type, but this does not always have to be the case. Depending on specific circumstances, an agent type may assume more than one roles. An agent type may have many instances during the running of system. The qualifier decides how many agent instances a role type can have. Qualifier is represented in a similar way of cardinality in UML diagram: “n” means n instances; “m··n” means the number of instance is between m and n; “+” means one or more instances; ”∗” means zero or more instances.

In this subsystem, each role corresponds to an agent type and thus there are nine agent types involved, shown in Figure 3. The SvcConfigurerAgent may have one or more instances where more than one instances may help balance the load. The AgreementParserAgent, LicenseTermDispatcherAgent and LicenseTermVerifierAgent may have zero instance when there is no license agreement is attached with requested service.

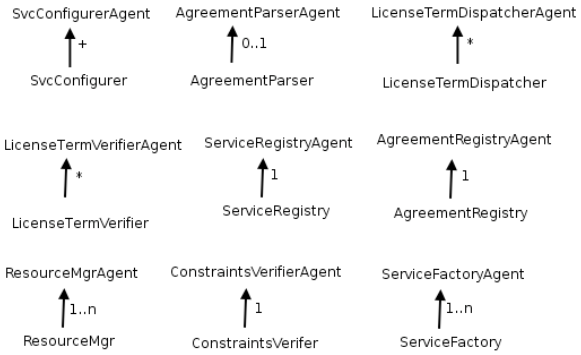

Fig. 3. The agent model of service configuration

D. Service Model

The service model lists all the functions an agent type engages, which come mostly from the protocols and activities and liveness of the roles this agent carries but not limited to them. For each function (service of the service model), Table II shows its input, output, precondition and postcondition. Here again, only the service model of the SvcConfigurerAgent are listed in the table below.

E. Acquaintance Model

At last, the acquaintance model presents how the communication among these agents take place. The communication paths of roles in the configuration subsystem are shown below in Figure 4. SvcConfigurerAgent interacts with five other agent types, while LicenseTermVerifierAgent interacts with LicenseTermDispatcherAgent only and ConstraintsVerifierAgent interacts with ResourceMgrAgent only.

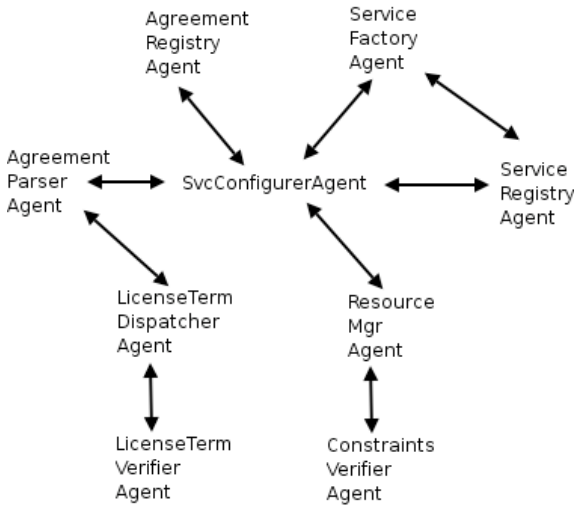

Fig. 4. The acquaintance model of service configuration

After the acquaintance model is generated, these two phases of Gaia methodology, producing five models, complete the agent design.

TABLE II
SERVICE MODEL FOR SVCCONFIGURERAGENT

Service	Input	Output	Pre-condition	Post-condition
parse service request	serviceRequest	user, service	serviceRequest $\neq$ nil	user $\neq$ nil $\wedge$ service $\neq$ nil
locate agreement	user, service	agreement	user $\neq$ nil $\wedge$ service $\neq$ nil	agreement $\neq$ nil
verify request statically	agreement	serviceRequestIsLegal	agreement $\neq$ nil	serviceRequestIsLegal $\in$ (true, false)
locate service	service	serviceNotReady serviceURI	service $\neq$ nil	($\neg$serviceNotReady $\wedge$ serviceURI $\neq$ nil) $\vee$ serviceNotReady
identify resource	service	resource	serviceNotReady	resource $\neq$ nil
request resource for service	resource	resourceGranted resourcesURI	resource $\neq$ nil	($\neg$resourceGranted $\wedge$ resourceURI = nil) $\vee$ (resourceGranted $\wedge$ resourceURI $\neq$ nil)
start service	resourceURI service	serviceURI serviceState	serviceNotReady $\wedge$ resourceGranted	serviceURI $\neq$ nil $\wedge$ serviceState = ready
start service observation	serviceURI	oberservationState	serviceURI $\neq$ nil	observationState $\in$ (started, ended)
stop service	observationState serviceURI	serviceState	oberservationState = ended	serviceState $\in$ (ready, inactive)
respond with result	serviceURI serviceState service		(serviceURI $\neq$ nil $\wedge$ observationState = started) $\vee$ (service $\neq$ nil $\wedge$ observationState = ended)	true

IV. IMPLEMENTATION

For the implementation of the SmArt License Management System, we started on the IBM Agent Building and Learning Environment (ABLE) [6] that provides a multiple agent system (MAS) platform with tools and a variety of implemented algorithms as building blocks to build the MAS. It also complies with the FIPA [7] standard of agent, which means that an ABLE agent will be able to communicate with other FIPA-compliant agents even if they are not based on ABLE.

We discuss some implementation and designs in this section, starting with the underlying mechanisms, and then explaining the implementation from two essential aspects of the agent — knowledge and bahavior. Discussion is kept brief as many details will not fit in this paper, but will cover various approaches of integrating ontology into MAS, applying Gaia analysis in MAS design, and implementing agents under guidance of task ontology and the Gaia desgin.

A. Knowledge Model and Ontology Manipulation

Most of the knowledge is represented as ontologies, e.g. SmArt Domain Ontology and SmArt Task Ontology, in OWL [8]/RDF [9]. OWL/RDF are XML [10] based, easy to extend, and make ontology expansion both possible and easy. The XML-basis also makes the ontology useable throughout networks, which is a must for management systems of data centers that provide services online.

This brought up an interesting question: how to let SmArt Agents understand and use ontologies effectively? The simplest approach may be to interpret OWL statements in realtime using toolkit like Jena [11] when implementing agents. However, this approach has obvious limits in many aspects, e.g. unnecessary complexity for low level OWL manipulation in MAS development.

Therefore we adopt another approach: extending ABLE with a set of essential ontology manipulation classes as the Ontology Extension (OE) for ABLE, which allows us to build a Knowledge Model (KM) for each agent in a MAS with ease. An overview of this approach is shown in Figure 5, where the OE is based on ABLE and each individual SmArt agent that holds the KM as its belief and has some behavior as its functionalities is built on top.

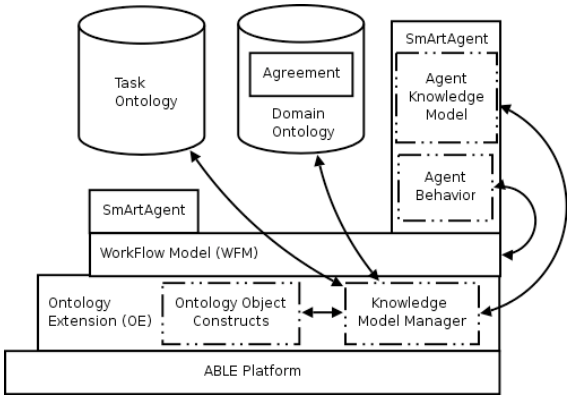

Fig. 5. The overview of implementation

Inside the OE, there are a series of abstract classes that can be used to construct ontology objects, which are actually java objects representing the *Concept* and the *Property* (*Predicate*) from OWL tuples, and representing the *Action* that is a special type of concept about doing things. These ontology objects can be manipulated directly as normal Java Beans. The KM in an agent consists of these ontology objects and reflects the belif and knowledge of the associated agent. By this means, when building agents MAS developers can work at a higher level of abstraction without having to worry about the manipulation

and interpretation of OWL statements at the lower level.

Another important component in OE is the KM Manager (KMM) that is capable of reading and interpreting an OWL document, and then creating KM on-the-fly based on the knowledge in that document. This will help to hide from MAS developers the complexity of transferring knowledge from an OWL document into ontology objects. Developers are able to work at the ontology object level. Also, KMM helps to serialize and deserialize ontology objects in a KM back and forth, translating knowledge to and from various formats, e.g. OWL, Plain Text, XML, or agent commmunication language (ACL) [12] as standardized in FIPA. In other words, KMM can help regulate communication among multiple agents.

B. From Domain Ontology to Agent Knowledge Model

When OE for ABLE is in place, we started to build our experimental SmArt MAS. One thing is to build for each SmArt agent a KM that represents its belief and knowledge according to the SmArt Domain Ontology. After an agent is built based on its role model and agent model from the Gaia analysis above, corresponding knowledge should be loaded at the very beginning of its lifecycle and such knowledge maintains and evolves throughout the whole lifecycle of this agent.

For the license management system, a variety of SmArt agents are being built upon the five models of Gaia methodology. SmArt agreements, along with the rest of the SmArt Domain Ontology and their concrete instances, are broken down into pieces. These knowledge pieces are to be loaded into related SmArt agents by the KMM, residing inside the agent's KM in the form of ontology object.

If one agent wants another to do something for it, the former can build an inquiry with details represented as ontology objects. As mentioned above, KMM is able to translate ontology objects into different formats for inter-agents communication. This means KMM helps to translate the inquiry containing ontology objects from the first agent's KM into appropriate ACL, and vice versa, translate ACL-based inquiry back to ontology objects and put them in the second agent's KM. Thus, in turn, an agent can understand inquiries from fellow agents by simply navigating through its own KM via direct interactions with ontology objects inside. Here, Domain Ontology, mainly as the type system, will be used as the guideline to create and parse these inquiries.

C. From Task Ontology to Agent Behaviors

SmArt MAS requires the construction of agent behavior. As shown in the implementation overview, each agent has some unique behavior. Behavior is based on the workflows described in the Role Liveness (RL) and the Service Model (SM) of that agent, and the RL and SM, at the same time, are compliant with the task ontology to which this agent is committed. When an agent is loaded with knowledge and put into the runtime enviroment, it is capable of responding to an inquiry by taking those steps of the workflow supported by the loaded knowledge.

So, agent behaviors are expressed on three levels of granularity, task ontology, agent design, and agent programming, at each of which flexibility needs to be considered and achieved as much as possible.

First, SmArt Task Ontology is the highest level abstraction of agent behaviors. Task ontology itself, as mentioned above, may have different levels of granularity, but no matter what abstraction level it is at, to its committed agents, the task ontology represents the basic workflows that consist of most roughly divided steps. At the Task Ontology level, the flexibility is inherent from the nature of ontology technology and OWL.

Secondly, during Gaia analysis and design of agent, task ontology based behavior is refined. Some complicated high level tasks from the task ontology are broken down and explicit sequencial information of workflows is added when RL and SM are produced. Therefore the RL and SM represent the refined workflows of an agent behavior. At the same time, to better serve the need of modeling RL and SM flexibly, we designed a data structure, the WorkFlow Model (WFM) as shown in Figure 5, which includes structs to represent tasks, control flows, data flows and conditions. The WFM is written in XML to express models from Gaia design, especially the RL and SM, which allows the RL/SM information to be used in developement easily. Based on WFM, we are in the process of building a rich client editing environment of modeling RL and SM based on task ontology and other Gaia models. Also, we are exploring the possibility of extending our base WFM into an RL/SM ontology.

Finally, after being refined from task ontology level to agent design level, agent behavior has to be further developed into implementable workflow and integrate detailed business logics of individual steps in the RL/SM-based workflow. A flexible mechanism is desired to facilitate possible changes of detailed business logic, which means updating the business logic of a workflow step after the system is deployed is possible. Such a flexible mechanism may be realized in the following ways:

1) Writing detailed logic directly in host language, e.g. Java;
2) Writing detailed logic in scripting languages, e.g. Javascript, Python, etc.;
3) Using REI (Rule Engine Integration) proposed in [13] and [14] that maps terms in a business logic rule to java objects using an XML-based mapping schema.

We started with the first approach and evolved the system with the other two. For the latter two approaches, atomic operations are implemented separatedly as functions or object methods and are made available for scripting languages or rule engines and a series of mapping schemas are also designed to integrate these method-based atomic operations into rule engines dynamically.

V. Conclusions and Future Works

This paper use Gaia methodology to analyze and design a license management MAS based on our SmArt framework. Though it could not present all the details of design and

implementation due to space limitations, the paper gives a broad picture of how the ontology and agents work together to provide the intelligent license management. The service configuration subsystem and the role of SvcConfigurer is used as an example, although the system can be used for a variety of setups.

Although we are getting encouraging results, this research is still developing. More challenging work is planned:

- Language to specify rules as ontology to make agents smarter and more generic.
- Generating agreements with conflicts between agreement items detected automatically.
- User friendly interface for service subscription.
- More license types to be defined and incorporated in the ontology.

ACKNOWLEDGMENT

The authors thank Natural Science and Engineering Research council of Canada and IBM Center for Advanced Studies for their support and cooperation for this research.

REFERENCES

[1] Q. Zhao, Y. Zhou, and M. Perry, "Agreement-aware Semantic Management of Services," in *Proceedings of International Conference on Autonomic and Autonomous Systems*, International Conference on Autonomic and Autonomous Systems 2006. IEEE, 2006.

[2] M. Wooldridge, N. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 3, pp. 285–312, 2000.

[3] Q. Zhao, Y. Zhou, and M. Perry, "Ontology of SmArt License Management System," 2007, prepublished manuscript.

[4] G. Capraro, G. Berdan, R. Liuzzi, and M. Wicks, "Artificial Intelligence and Sensor Fusion," in *Proceedings of International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, International Conference on Integration of Knowledge Intensive Multi-Agent Systems. IEEE, 2003, pp. 591– 595.

[5] M. Panteleyev, D. Puzankov, P. Sazykin, and D. Sergeyev, "Intelligent Educational Environments Based on the Semantic Web Technologies," in *Proceedings of the 2002 IEEE International Conference on Artificial Intelligence Systems*, the 2002 IEEE International Conference on Artificial Intelligence Systems. IEEE, 2002, pp. 457–462.

[6] M. Meyer, "The features and facets of the Agent Building and Learning Environment (ABLE)," International Business Machines Corporation, Tech. Rep., 2004.

[7] F. S. Organization, "FIPA Specifications," The Foundation for Intelligent Physical Agents, Tech. Rep., 2005.

[8] P. F. Patel-Schneider, P. Hayes, and I. H. eds., "OWL Web Ontology Language Semantics and Abstract Syntax," W3C Recommendation, Tech. Rep., 2004.

[9] G. Klyne and J. C. eds., "Resource Description Framework (RDF): Concepts and Abstract Syntax," W3C Recommendation, Tech. Rep., 2004.

[10] T. Bray, J. Paoli, and C. M. S.-M. eds., "Extensible Markup Language (XML) 1.0," W3C Recommendation, Tech. Rep., 1998.

[11] B. McBride, "Jena: Implementing the RDF Model and Syntax Specification," in *Semantic Web Workshop*. WWW2001, 2001.

[12] F. S. Organization, "Agent Communication Language Specifications," The Foundation for Intelligent Physical Agents, Tech. Rep., 2002.

[13] Y. Zhou, "On Demand Service Level Agreement: Architecture and Eforcement," Master's thesis, The University of Western Ontario, 2004.

[14] Y. Zhou, Q. Zhao, and M. Perry, "Reasoning over Ontologies for SLAs," in *Proceedings of The IEEE International Conference on e-Technology, e-Commerce and e-Service*, EEE2005. IEEE, 2005.