Proceedings of The 6th IEEE International Symposium on Service Oriented System Engineering (SOSE 2011)

Unifying Domain Ontology with Agent-Oriented Modeling of Services

Zhi Jin
Key Lab. of High Confidence Software Tech.
School of Elect. Eng. and Comp. Sci., Peking University
Beijing 100871, China
Email: zhijin@sei.pku.edu.cn

Hong Zhu
Dept. of Comp. & Comm. Technologies
Oxford Brookes University
Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk

Abstract— **Modeling plays a crucial role in model-driven development of service-oriented systems. This paper proposes a framework for service-oriented modeling that combines an agent-oriented software development methodology with an ontology-based domain analysis technique. It aims at improving the dynamic composability of services at requirements and design stages through modeling. The framework consists of an architectural structure of service models and a process of modeling. The architecture combines agent-oriented models of software systems in which service providers and requesters are regarded as autonomous entities (and called agents), and domain ontology, which specifies the entities in the application domain and their dynamic behaviors. The domain ontology extends classic ontology by including causal and symbolic entities as well as autonomous entities. The approach is illustrated by an example of online auction service.**

Keywords—Service-oriented computing, Software modeling, Service engineering, Software agents, Domain ontology.

I. Introduction

Service-oriented computing (SOC) aims at realizing distributed applications through developing services as basic computing entities and composing available services dynamically with highest flexibility [1]. The one of the key characteristics of service-oriented architectures is the dynamic service composition through discovering and invoking available services that are published and registered. How to develop services to achieve this mission is still one of the main challenges in service-oriented system engineering.

A. Related Work

Among various approaches proposed in the literature of service-oriented engineering, the model-driven approach offers a high degree of automation and high level of abstraction [2,3,4,5]. The past few years have seen a rapid growth of research in this area. Existing works fall into the following two classes.

- *Extension of object-oriented and component-based modeling.*

Among the most well-known works of this type, Levi and Arsanjani [6] and Endrei, Ang and Arsanjani [7] used goals to guide behavioral specification for components and extend component-based analysis beyond traditional object-oriented analysis and design to model service-oriented applications. Zimmermann, Krohdaul and Gee [8] considered service-oriented modeling as a hybrid approach including a set of traditional techniques that incorporate object-oriented analysis and design, business process modeling and enterprise architecture description. Arsanjani [9] advocated an iterative and incremental SOA modeling process that consists of identification, specification and realization of services, components as well as workflows. Zhang et al. have also developed a software development environment and tools SOMA-ME [2] that facilitate model-driven development of services based on the SOMA methodology [2]. It extends UML 2.0 by a profile that defines the concepts related to services and extends the IBM Rational Software Architect product to provide a development environment for designing SOA solutions in a model-driven fashion.

- *Workflow models for service choreography and orchestration.*

Modeling languages have been proposed and standardized for service choreography and orchestration, such as WS-BPEL (Web Services Business Process Execution Language) [10], WS-CDL (Web Services Choreography Description Language) [11], etc. These languages directly aim at the composition of web services. However, although they are widely regarded as process modeling languages, strictly speaking, they are not suitable for model driven service development due to their relatively low level of abstraction. A number of researchers have investigated workflow specification and modeling based on various formal notations, such as Petri-nets, process algebra, finite state machine and automata.

B. The Open Problems

Despite of these efforts, existing works on service-oriented modeling (SOM) have not adequately addressed the following important issues of SOC.

First of all, as Stal pointed out [12], service oriented architecture (SOA) is fundamentally different from the traditional distributed computing technologies. Many researchers have further argued (C.f. [13, 14]) that the services in a service-oriented system differ from traditional components by their autonomous and social behaviors. They are autonomous in the sense that they control their own resources and their own behaviors. They may demonstrate social ability by collaborating with each other through dynamic discovery and invocation of services unknown at design time. Therefore, they should be regarded as agents. Here, by the word agent we mean a computational entity that encapsulates its states, operations, behavior rules and an explicitly described environment. They are the computational entities that provide services, and often are also consumers of other services. This is just like estate agents provide services of buying and selling estate properties, and travel agents provide services of buying and selling air tickets, tourism products, etc.

Secondly, to realize the full power of SOC, it does not only require the interfaces between integrated entities syntactically compatible, but more importantly, the interactions must be semantically correct. It is currently a major problem in the development of service-oriented applications to enable dynamic search of semantically correct services and to understand required services with correct meanings. The domain knowledge of the application shared by services is essential to the semantic correctness of dynamic discovery and invocation of services. Much research has been reported in the development of infrastructure and enabling technologies addressing this issue under the title of Semantic Web Services [15]. However, little has been done in the research on modeling SOA that takes ontology and domain knowledge into consideration.

Finally, how to develop new services efficiently, say by reusing existing software components or services, have not been taken into consideration, too. In general, existing work on service development only focuses on enabling dynamic service composition at runtime. Languages for business process modeling including WS-BPEL and WS-CDL have been proposed to support runtime compositions. But, how to model service composition at a high level of abstraction remains a challenge to the development of service oriented applications.

C. *Previous Work*

In our previous work, we have addressed these issues separately. In particular, considering services as the functionality provided by agents, Zhu and Shan [16] proposed an agent-oriented approach to SOM based on an agent-oriented software development methodology [17]. The agent-oriented modeling language CAMLE and its supporting automated modeling tool is applied to the construction of models, the checking of consistency between the models of different vendors and the transformation of graphic models into formal specifications. However, it relies on formal notations to specify the semantics of each operations provided by the services. Its main drawback is that using and developing the standardized domain ontology were not taken into account. They are of vital importance for SOC.

To address the semantics issue, Wang and Jin *et al.* [18,19] proposed an ontology-based domain knowledge approach to the specification of service capabilities. The semantics of the operations can be specified at a high level of abstraction. It addressed the development and uses of domain ontology issue and extended the traditional vocabulary oriented ontologies to domain ontologies for effective definition of the semantics of services. However, modeling the structure of service-oriented applications, the dynamic behavior of services and service interactions is left open.

More recently, in [20], Wu and Jin *et al.* proposed a reuse-based approach to the development of services. It emphasizes on identifying and collecting reusable assets contained in service-oriented systems, specifying these reusable assets in an ontology, and reusing those assets whenever possible during service-oriented software development. Their experiments show that this modeling approach is effective in facilitating software asset reuse and reducing the modeling time. However, modeling collaborations between services owned by different vendors is still an open problem.

D. *Contributions of This Paper*

This paper intends to unify our previous work and aims at providing a systematic methodology, which includes a controllable process and a modeling language. The main essences of the approach proposed in this paper include:

- Using agents to capture the autonomy and activeness of the participants in service-oriented computing so that the corresponding language facilities, such as caste, which is the classifier of agents [21] just like class is the classifier of objects, can be used as the basic building block for the construction of service oriented applications;
- Employing domain ontology to provide sharable knowledge and terminology for defining the semantics of services at a high level of abstraction and to provide a standardized vocabulary to facilitate the communications between these participants from different vendors; and
- Devising a process model to provide guidelines to the model-driven development of service oriented applications.

This paper is organized as follows. A framework of agent-based SOM is outlined in Section II, which includes a discussion of the general principles of SOM, the architecture of models and the process of modeling. The modeling of service capability based on domain ontology is addressed in Section III. Section IV integrates the ontology-based service capability modeling into the agent-oriented structural and behavioral service modeling. Section V discusses the consistency and completeness checking of models. Two sets of constraints on model's consistency and completeness are presented. Section VI concludes the paper with a summary of the contributions of the paper.

II. Overview of the Framework

In this section, we extend the framework of agent oriented service modeling [13,16] to incorporate domain ontology. The framework includes an architecture of models and a modeling process.

A. *Architecture of Service Models*

The original architecture of agent oriented service models contains three types of models, which are structural model, collaboration model and behavior model. Each model is divided into three parts: (a) Specification of provided services; (b) Specification of expected collaborators; and (c) Specification of internal designs. There is no reference to domain ontology in this architecture. Here, we extend it by adding domain ontology as a new type of model and revise the behavior model. As shown in Figure 1, the new architecture consists of the following views.

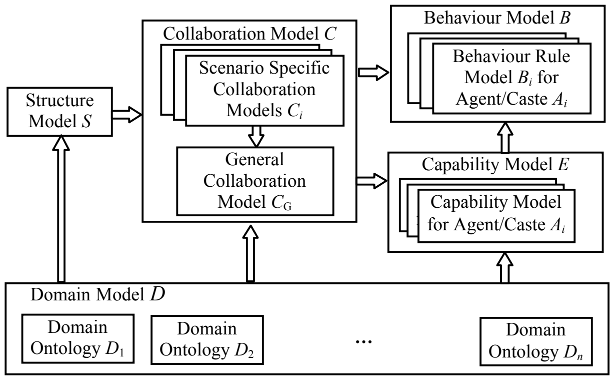

Collaboration Model C
Behaviour Model B
Scenario Specific Collaboration Models C_i
Behaviour Rule Model B_i for Agent/Caste A_i
Structure Model S
General Collaboration Model C_G
Capability Model E
Capability Model for Agent/Caste A_i
Domain Model D
Domain Ontology D_1
Domain Ontology D_2
...
Domain Ontology D_n

Figure 1 Architecture of Service Models

- *Domain Model*: It is a model of the application domain of the services. It defines the types of entities in an application domain and their state spaces and life cycles so that a standard vocabulary of the domain can be defined as the base for specifying the semantics of services. A service application may involve more than one domain, thus the domain model may contain multiple domain ontologies.
- *Structure Model*: It is a model of the service oriented system. It consists of a set of entities involved in the service application, which include autonomous entities (such as other services), causal entities (such as the objects and equipments in the real world that the service operates) and symbolic entities (such as values that represent the states of objects and services). The static relationships between them are depicted in two diagrams: a caste diagram for the relationships between service roles and an entity relationship diagram for the effects of service operations on the real world entities.
- *Capability Model*: It defines the capabilities of the services by specifying the effects imposed by the service on the entities in the system. The definitions of the capabilities are based on the domain model.
- *Collaboration Model*: It specifies the collaborations between the participants of the service. It may include an overall model of collaborations and a number of scenario-specific collaboration models. The overall collaboration model defines the interaction messages among services. The meanings of the messages are defined based on the domain model. Each scenario-specific model defines the interaction process in a specific scenario by indicating the sequence of messages passing between the participants of the collaboration.
- *Behavior Model*: It provides detailed specification of each service's behavior from an individual perspective in the form of a set of behavior rules. These rules specify how the participant should behave in the interaction with others. A behavior model can also be divided into two parts: internal behavior model and external behavior model. The former is to define the service provider's internal decision making process and hidden from the outside. The external behavior model specifies the behavior of the provided service as the others can expect. It also specifies the expected behaviors of the outside services that it relies on.

B. *Modeling Process*

To incorporate domain ontology into service modeling, the overall framework of model-driven service development proposed in [16] is extended by adding domain analysis and revised by modifying other activities as shown in Figure 2, where modified and new activities are in grey boxes.

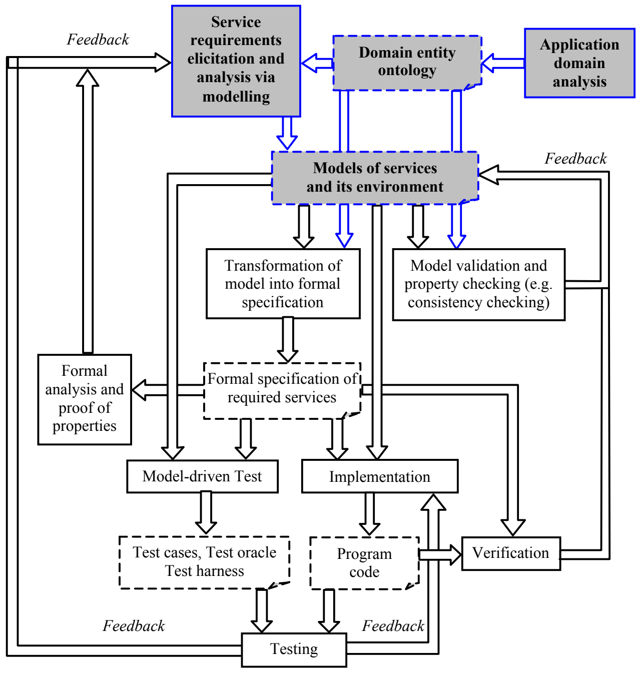

Feedback
Service requirements elicitation and analysis via modelling
Domain entity ontology
Application domain analysis
Models of services and its environment
Feedback
Transformation of model into formal specification
Model validation and property checking (e.g. consistency checking)
Formal analysis and proof of properties
Formal specification of required services
Model-driven Test
Implementation
Test cases, Test oracle Test harness
Program code
Verification
Feedback
Testing
Feedback

Figure 2 Overall framework of model-driven development of SOCA

Accordingly, as depicted in Figure 3, the original modeling process in [13,16] is now extended with two additional phases *domain analysis and modeling* and *capability modeling*. The original activities in *structure modeling*, *collaboration modeling* and *behavior modeling* phases are also revised.

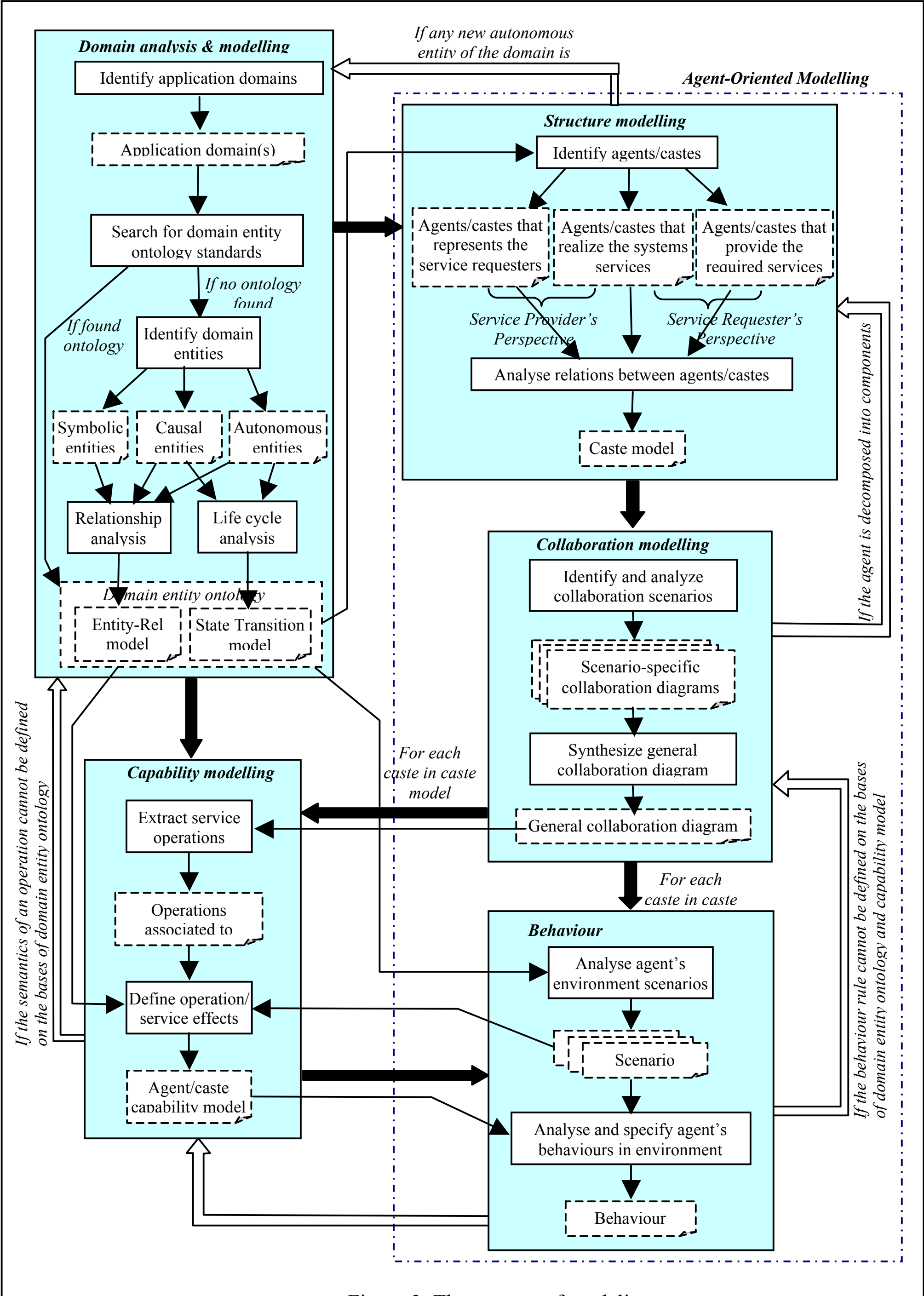
The new process starts with *domain analysis and modeling* activities. The first is to identify the application domains involved in the service under development. Then, for each application domain, its standard domain ontology is searched for in a repository. If a suitable ontology is not available, further domain analysis and modeling activities must be carried out, which is described in more detail in Section III.

The original *structure modeling* activities are now supported by the results of domain analysis and modeling, which provides a standard vocabulary of the application domains and the common models of the entities and agents in the application domain so that the structural model of a service conforms to the existing ontologies of the application domain(s). On the other hand, the domain ontology does not match the requirements of the service under development, feedbacks to further domain analysis and modeling activities will be passed to domain analysis to revise the domain ontology. For example, when a new type of agents or entities is identified as essential and necessary for an application, but not in the ontology, a revision of the domain ontology will take place.

In the original process model, *behavior modeling* has two goals: to define behavior rules for each caste of agents and to define the semantics of each action that an agent can take. These two goals are now divided into two iterative phases. The *capability modeling* phase defines the semantics of actions, while *behavior modeling* now focus on behavior rules that determine when an action is to be taken.

The capability modeling activities depend on collaboration modeling to find the set of actions that the agents of a caste can take, and then to define their semantics according to the domain ontology by specifying the effects of each action; see Section III.B for details. Feedbacks on domain analysis and modeling can be obtained from capability modeling if the required capability of an action cannot be defined on the basis of the domain entity ontologies. In such a case, the domain ontology must be revised accordingly. If such revision is not practical, it means that the required collaboration among agents is not feasible, thus needs modification.

The behavior modeling activities use both capability model and collaboration model to define behavior rules. It can give feedback to collaboration modeling if behavior rules cannot be defined based on the capability definition due to, for example, an infeasible collaboration among agents is required. Therefore, by using domain ontology and capability modeling, we provide a practical guideline for the iteration of collaboration modeling and behavior modeling.



Domain analysis & modelling
Identify application domains
Application domain(s)
Search for domain entity ontology standards
If found ontology
If no ontology found
Identify domain entities
Symbolic entities
Causal entities
Autonomous entities
Relationship analysis
Life cycle analysis
Domain entity ontology
Entity-Rel model
State Transition model
If any new autonomous entity of the domain is
Agent-Oriented Modelling
Structure modelling
Identify agents/castes
Agents/castes that represents the service requesters
Agents/castes that realize the systems services
Agents/castes that provide the required services
Service Provider's Perspective
Service Requester's Perspective
Analyse relations between agents/castes
Caste model
If the agent is decomposed into components
Collaboration modelling
Identify and analyze collaboration scenarios
Scenario-specific collaboration diagrams
Synthesize general collaboration diagram
General collaboration diagram
For each caste in caste model
For each caste in caste
Capability modelling
Extract service operations
Operations associated to
Define operation/ service effects
Agent/caste capability model
If the semantics of an operation cannot be defined on the bases of domain entity ontology
Behaviour
Analyse agent's environment scenarios
Scenario
Analyse and specify agent's behaviours in environment
Behaviour
If the behaviour rule cannot be defined on the bases of domain entity ontology and capability model

Figure 3. The process of modeling

III. MODELING DOMAIN ENTITIES

As discussed in the previous section, modeling domain entities plays a vital role in specification of the semantics of services, especially in defining service capabilities. However, existing frameworks of ontology are insufficient to achieve these goals. Therefore, we extend the existing framework to an environment-based approach. This section elaborates this approach to domain modeling in the context of service engineering.

A. Requirements of Domain Modeling in Service Engineering

In order to enable the definition of the semantics of services, we extend the structure of domain ontology. The reason for the extension is due to the following requirements on the domain ontologies.

1) *To specify the effects on the entities in real world that software services exhibit their capabilities.*

The semantics of a software service is its effect on those entities that it interacts with. Hence, the capability of a software service can be best expressed by the changes it brings onto the entities in the real world. The characteristics of these entities in the real world and their interconnections with the service are what we need to know when specifying the capability of the software service. The ontology framework must support the definition of services' capability in terms of their effect on external entities.

2) *To specify the entities in the real world that are shared among services and mediate the interactions among these services.*

Software services are distributed and loosely coupled. Sharing information about the syntax and semantics of services is a precondition for the services to interact with each other. The domain entities in the real world are outside the boundary of software system. They are shared by the services and they mediate the collaborations between services of different vendors. Moreover, they are independent to any specific application and specific implementation. Hence, their specifications can be and should be standardized and published as a part of ontology rather than as a part of services. The ontology framework must support the definition of such common knowledge of the domain.

3) *To specify domain entities that are stable even if the requirements on the services change frequently.*

The main driving force for changes in an open software system is the changing requirements on the processing of the domain entities in the real world. In other words, changes in desired effects on the domain entities in the real world cause the evolution of software. To a particular software service, while its effects on the real world entities evolve with the changing requirements, the real world entities themselves are comparatively stable. The modeling of domain entities should recognize the stable aspects of domain entities so that the model can facilitate the evolution of services on a relatively stable foundation.

The current general ontological structure only supports the declarations of the concepts of a domain and the relations between them [22, 23]. This structure is inadequate to specify the life cycle and operations on domain entities, and thus inadequate in specifying the effects that services can impose on them. Therefore, in the next subsection, we extend the general ontological structure by including entity's dynamic features. In the sequel, a specific ontology in this new framework is called a *domain ontology*.

B. A Meta-Model of Domain Ontology

Our meta-model of domain ontology is an environment-based approach to service capability specification [19]. In this approach, we specify the capabilities of services by defining their effects on the real world entities, such as the tickets, the credit cards, the hotel rooms, etc. These effects are modeled by a state transition system in which the state transitions of the real world entities are the results of service operations applied to the entities. Therefore, in domain modeling, these real world entities are the objects and concepts to be modeled in the ontology. They are domain-specific, but independent of any particular service.

The environment-based view has its origin in the research on requirements engineering [24,25]. In this area, it is widely recognized that the requirements of a software system act as the meeting point between an internal world and an external world. Here, the internal world refers to the "*machine*" (the software's internal construction) and the external world refers to the "*world*" (the environment in which the software will operate).

Along this viewpoint, we conceptualize an application domain by considering the following two aspects.

First, we follow the principles of the Problem Frames method [25] to identify and classify the entities in an application domain. According to the method, a conceptualization of an application domain consists of two parts. The first part is a set of concepts that are captured in the identification of phenomena in reality and can be used to identify and specify explicitly the real world entities. The concept hierarchy is shown in Figure 4(a). It includes the concepts and the hierarchical relationships among them. The meaning of each phenomena-related concept is given in Table 1.

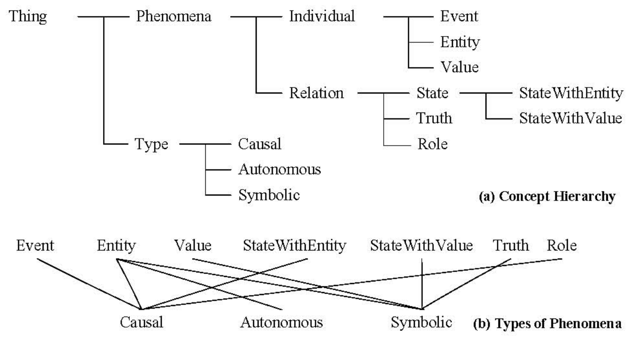


Figure 4 Conceptual Hierarchical Structure of Domain Ontology

The second part of conceptualization is the classification of phenomena into a number of types. Each type characterizes a set of phenomena by certain features (see Figure 4(b)). There are three types, which are causal, autonomous, and symbolic. That is, a phenomenon can be causal, autonomous or symbolic.

- Causal phenomena are phenomena that can be controlled and can cause other phenomena in turn.

- Autonomous phenomena are phenomena that are used to represent the physical autonomous entities.
- Symbolic phenomena are phenomena that are used to symbolize other phenomena and relationships between them.

Table 1 Meanings of the Phenomena Related Concepts

Concept	Meaning
Individual	Something that can be identified, be named and be distinguished from each other
Event	An individual that is an occurrence at some point in time, and regarded as atomic and instantaneous
Entity	An individual that persists over time and can change its properties and states from one point in time to another
Value	An individual that exists outside time and space and is not subject to change, including numbers and characters
Relation	A set of associations among individuals
State	A relation among individual entities and values. It can change over time
State-Of-Entity	A state of an entity
State-Of-Value	A state of a value
Truth	A relation among individual that cannot possibly change over time
Role	A relation between an event and individuals that participant in it in a particular way

For example, event, role and state of entity are causal. And, value, truth and state of value are symbolic.

It is worth noting that an entity can be an interactive or operating individual. It is an instance of its type. It can be seen as a set of related phenomena that are usefully treated as a unit. The classification of phenomena also applies to the classification of entities. This implies that an entity can be causal, autonomous, or symbolic.

- A causal entity has predictable causal relationship among its causal phenomena.
- An autonomous entity can autonomously decide what it wants to do.
- A symbolic entity is a physical representation of data.

Second, we identify and model domain entities in the context of service capability specification and modeling. In particular, a service exhibits its capability through its interaction with its environment and imposing effects on entities in the environment. Therefore, the domain ontology must contain the set of real world entities that interact with the services. Moreover, for each entity, the changes of its states are caused by the operations performed by the services. The ontology must define the state space for each entity as well as its state transitions so that service capabilities can be defined. This led us to the meta-model of service capability specification depicted in Figure 5. It gives the ontological structure of service capability specification. The meanings of the concepts and their relationships in the meta-model are as follows.

- Each service has one or more capabilities.
- Each capability exhibits some effects upon a set of domain entities.
- Each effect is the changes of the states of a set of the domain entities.
- Each occurrence of a scenario causes at least an effect.
- Each scenario is an interaction flow, which is an ordered sequence of interactions.
- Each interaction is a shared phenomenon between a service and a domain entity and represents one individual action that the service performs.
- An interaction has one initiator and one receiver that could be a service or a domain entity.

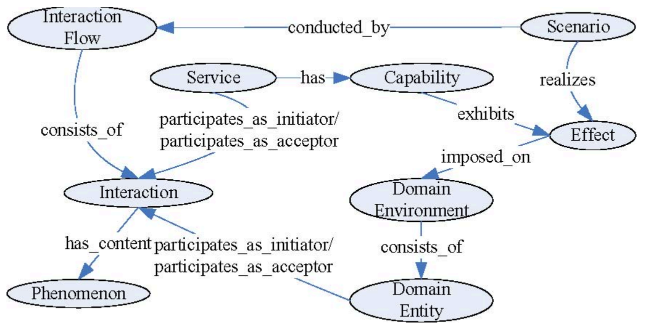


Figure 5 Service Capability Ontology

Note that, a scenario is a sequence of individual interactions into an interaction flow, which realizes a meaningful change to the environment of the service. The effect of an interaction must be understood in the context of such scenarios. Without such a scenario, an interaction or operation of a service may be meaningless.

C. *Example: Online Auction*

We now use online auction as an example to illustrate various domain modeling activities discussed above. The example will also be used throughout the remainder of the paper. The online auction ontology developed in this subsection will be the basis of modeling auction services.

First of all, we identify the domain entities, which are real world entities and could be concrete or abstract. For the auction domain, we have buyer, seller, auction, item, bid and so on. Among them, item, bid and auction are causal, while buyer and seller are autonomous. These domain entity types can be further specified. For example, auction has several symbolic entities, e.g. starting date, ending date and price. A causal entity, e.g. item, may have its attributes.

There are two types of static relationship between entities in domain ontology. The first is the classifications of entities, i.e. the inheritance relation, and the second is the whole-part relationship. Such relationships can be represented in the Caste diagram of CAMLE modeling language [27], in which a double rectangle represents a caste, an arrow indicates an inheritance relation, and a diamond indicates a whole-part relation.

For example, for the auction domain, as shown in Figure 6, the caste nodes in a hierarchical structure within the dashed rectangle represent these domain entities.

The entity-relationship diagram given in Figure 7 shows how these entities are related to each other. Then, for each causal entity in the model, domain ontology further specifies its dynamic features by a state transition diagram to describe its life cycle. Figure 8 shows the state transition diagram for some causal entities in the online auction domain.

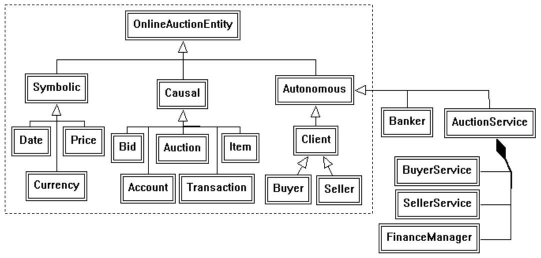


Figure 6 Structure Hierarchy Diagram of Auction Service

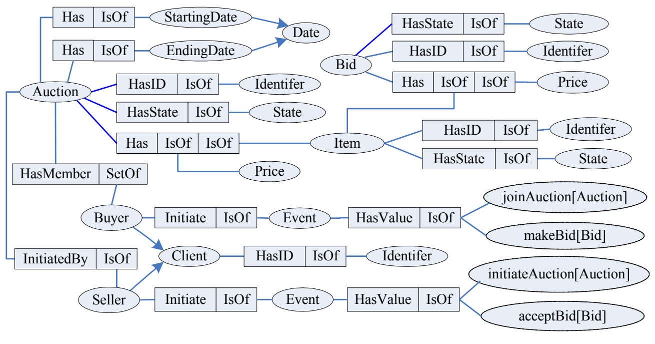


Figure 7 Auction Domain Entity Relation Diagram

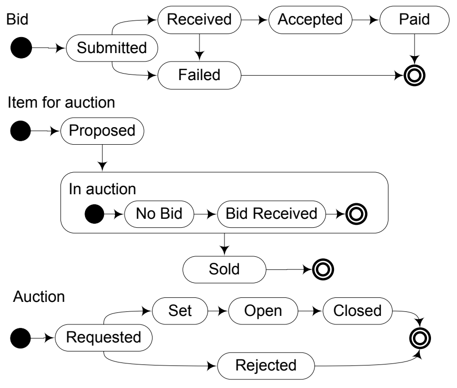


Figure 8 State Transition Diagram for Entities in the Auction Domain

As discussed above, from an environment-based perspective, a service capability can be featured by the entities it operates and the changes it makes upon these entities. With the above domain ontology of online auction, some standard service capabilities can be defined. The following (Figure 9) gives two examples, ApproveJoinAuction and ApproveWithdrawAuction, in the auction domain.

The capability definition of ApproveJoinAuction is for the service MembershipManager. It involves two entity instances, i.e. auction, which is an instance of Auction, and client, where client is an instance of Buyer. The capability of ApproveJoinAuction is the ability that any instance mg of the service role MembershipManager should have. It states that in the scenario when a client (in the role of Buyer) takes an action of Join(auction) and the message is sent to mg, the service instance mg should take the action of acceptTo(auction) and send the message to the client. After completing the action, the client will become a member of auction (changed from being a non-member) and a new association hasMember will be added between the auction and client. The capability definition of ApproveWithdrawAuction is similar.

```
Capability Definition ApproveJoinAuction
{   For Service all mg: MembershipManager
        Participants
            client: Buyer, auction: Auction,
        Scenario
            client: join(auction)→mg
        Action
            mg: acceptTo(auction→client
        Effects
            client:StateChange
                pre:  beInState(non-member(auction))
                post: beInState(member(auction))
            auction:AssociationCreation
                pre:  not(hasMember(auction,client))
                post: hasMember(auction,client)
}
Capability Definition ApproveWithdrawAuction
{   For service all mg: MembershipManager
        Participants
            client: Buyer, auction: Auction,
        Scenario
            client: withdraw(auction→mg
        Action
            mg: withdrawOK(auction)→client
        Effects
            client:StateChange
                pre:  beInState(member(auction))
                post: beInState(non-member(auction))
            auction:AssociationElimination
                pre:  hasMember(auction,client)
                post: not(hasMember(auction,client))
}
```

Figure 9 Examples of Capability Definitions

Figure 10 is an example of a service's capability definition. It defines the capability of service MembershipManager by referring to the capability definitions given in Figure 9. An alternative format for service capability definition is to replace the references by the body of capability definition.

```
Capability Profile MembershipManager {
    Capability:
        ApproveJoinAuction;
        ApproveWithdrawAuction;
}
```

Figure 10 Example of Service Capability Definition

A capability model sets the scope of the service by naming the engaged domain entities and the requests that the service can process. It defines the functionality provided by the service.

IV. Integration of Domain Ontology with Agent-Oriented Modeling

In this section, we revise and extend the agent-oriented modeling of services [16] to incorporate domain analysis. We will demonstrate how to use the domain ontology to support structure modeling by providing sharable terminology and common knowledge of the application domain. We also show how capability modeling of services based on domain ontology supports the definition of the semantics of services. This supports the explicit representation of services and improves the model's readability and understandability.

A. *Structure Modeling*

Structure modeling aims at identifying various types domain entities involved in the modeled service and clarifying the relationships among them. Normally, a collection of domain entities involved in the services should have been identified, defined and classified in domain analysis. However, the domain ontology usually only contains the common entities that all services of the domain will recognize. Therefore, structure modeling needs to further analyze as the constituents of the specific services under development. Particularly, at the highest level of abstraction, we inherit the domain entity hierarchy and extend it by including the software services, such as the service manages auction processes, as a kind of autonomous entities.

The inheritance and whole-part relationships between entities in structure modeling must observe the following points.

First, the entities and the relationships in the structure model must be consistent with the entities and the relationships specified in the domain ontologies. In particular, the inheritance and whole-part relations specified in the ontologies must be preserved in the structure model.

Second, a structure model may contain entities from multiple domain ontologies. For example, Figure 6 contains entities from the auction domain ontology and the banking domain ontology. Consistency between the ontologies must be maintained and conflicts resolved.

Third, a structural model may also have additional entities that are not in the domain ontologies. In particular, it may contain entities to represent the services to be implemented as well as new entities to be created and processed by the service. For example, in the structural model of auction services shown in Figure 6, we add a new abstract autonomous entity, i.e. AuctionService, and attach to it the concrete auction services Service to Buyer, Service to Seller and Finance Manager as it components.

Finally, entities in a domain ontology may be excluded from the structure model if the entity is not involved in the service under development. For example, in the banking domain ontology, we probably will have entities like saving account, current account, etc. as subclass of account. In the context of online auction services, we will not deal with saving account, thus it is excluded from the structure model.

The second type of static relationships between entities that structure modeling deals with is between autonomous entities and causal/symbolic entities. This type is similar to the association relations between objects in object-oriented modeling. We model such relations in an entity relationship diagram; see Figure 11 for the example of online auction service.

Note that, in object-oriented modeling, this type of static relationships together with inheritance and whole-part relations are usually represented in one class diagram. However, in our approach they are split into two diagrams, one only contain information about inheritance and whole-part relationships and the other only represent associations. One reason for this is to avoid over crowded diagrams as service oriented applications are often too complicated to be represented on one diagram. Another reason is that the associations between services and domain entities can be dynamically established. Thus, the associations need to be modeled separately. Normally, such entities relationship diagrams can be derived from domain ontology. For the auction service example, the entity relationship diagram given in Figure 11 can be derived from the domain ontology given in Figure 7.

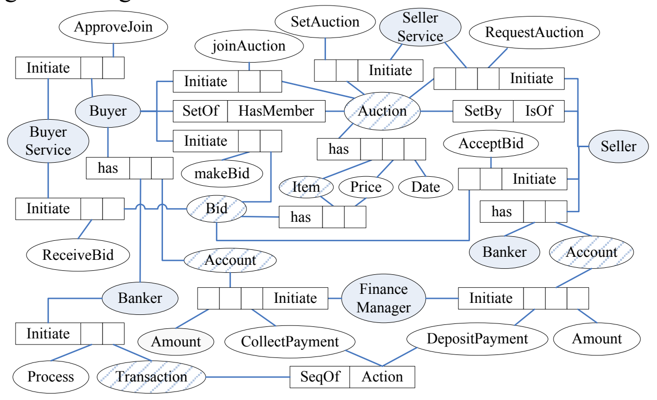


Figure 11 Auction Entity Relationship Diagram

B. *Collaboration modeling*

A collaboration model specifies the interactions between the participants of a service. It must be consistent with the structural model and the domain ontology in the following two senses.

First, the participants of the collaboration must be entities identified in the caste diagram.

Second, the actions must be those that have been explicitly defined in the domain ontology. For example, in the online auction example, a Buyer can initiate events, e.g. join[Auction] and make[Bid]. And, the buyer who made the successful bid will obtain the item after making the payment.

As discussed in Section 3, scenarios provide the context in which capabilities and interactions can be understood. We use scenario analysis to develop collaboration models. Informally, a scenario is typical situation of service usage in which a sequence of interactions between participants takes place.

The first step in scenario analysis is to identify a set of scenarios in the operation of the service under development. For example, we can identify the following list of the scenarios in the operation of the online auction service.

1. Join an auction
    a. successful;
    b. failed due to wrong manager;
    c. failed due to state is not open;
    d. failed due to the membership already exists.
2. Set up an auction
    a. successful;
    b. failed due to manager busy;
    c. failed due to item not valid.
3. Make a bid
    a. successful;
    b. failed due to low bidding price;
    c. failed due to auction closed.
4. Complete an auction
    a. successful;
    b. failed auction with no bid.

A scenario can be specified as a sequence of interactions between participants, where each interaction is an action taken by the participants, and the assertions on the states of the entities before and/or after each action. It can be represented in the CAMLE modeling language using scenario diagrams. Figure 12 is an example of scenario diagram that defines the scenario Join Auction Successfully.

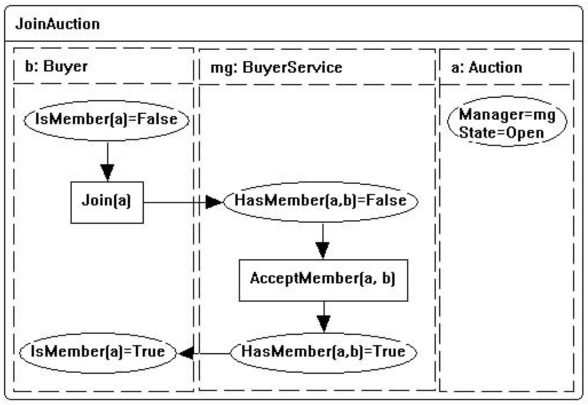


Figure 12 Scenario Diagram for Join Auction

As shown in above example, in a scenario diagram, each participant has a swim lane with the name and its caste in the top compartment. Its actions and state assertions are placed in the swim pool, i.e. the lower compartment. The assertions about the states of the entities are represented by predicate nodes, which are depicted as oval circles. These assertions reflect the effects of the actions in the scenario in terms of the state changes. The actions taken by a participant are represented in action nodes, which are depicted as boxes, and placed in the swim lane of the participant. The temporal ordering of the state changes and actions are indicated by arrows between them.

From a scenario diagram, we can derive a scenario specific collaboration diagram to describe the communications between the participants in a scenario. For example, Figure 13 shows the collaboration diagram for the Join Auction scenario.

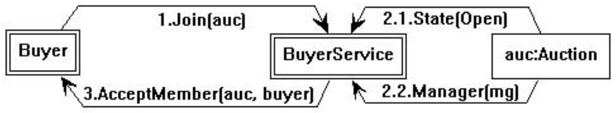


Figure 13 Collaboration Diagram for Join Auction Scenario

The collaboration model of a service contains a set of scenario specific collaboration diagrams and a general communication diagram, which is also in the notation of collaboration diagram, but specifies all the communications between castes and other entities. The general communication diagram can be derived from the scenario specific collaboration diagrams fairly straightforwardly. Figure 14 shows the general communication diagram for the online auction service.

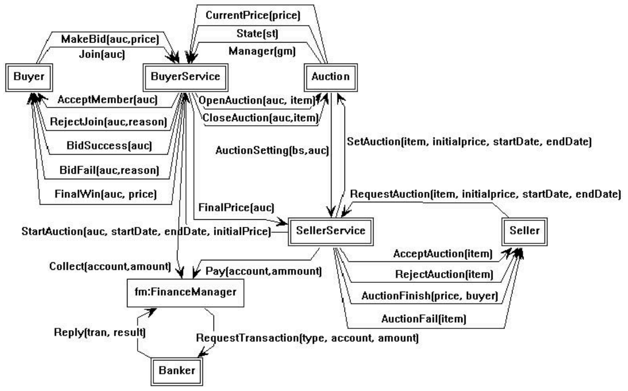


Figure 14 General Collaboration Diagrams of Online Auction Service

During the development of a collaboration model, new entities may be discovered. In such cases, the process will be backtracked iteratively to the domain analysis and capability modeling for the new entities and updating the structural model by including the new entities.

C. Behavior Modeling

The next phase of the agent-oriented modeling of service application is behavior modeling. It aims at defining a behavior model for the service and a model for each participant of the interactions with the service. A behavior model consists of a set of behavior rules that determine the actions to be taken in certain scenarios.

In the agent-oriented modeling language CAMLE, a behavior model is a behavior diagram that depicts behavior rules. Each behavior rule consists of the following elements:

- the scenario in the environment that will trigger the agent to take an action;
- the precondition on the agent's internal state for the action to be performed;
- the action to be taken by the agent together with the parameters of the action.

Figure 15 shows a segment of the behavior diagram for the Buyer Service of the online auction system. It defines how the service processes a membership request made by a buyer.

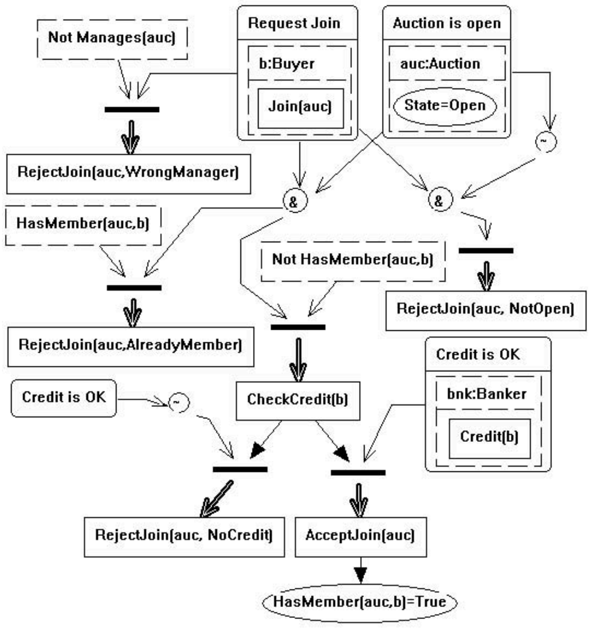


Figure 15 Behavior Rules to Process Membership Requests

V. Consistency and Completeness

As discussed above, consistency and completeness between service models and domain ontologies plays a vital role in our approach. This section discusses how to check the consistency and completeness between service models and domain ontologies.

A. *The Framework*

The automated modeling tool CAMLE is capable of checking the consistency and completeness of agent-oriented models against a set of rules [26,27]. Zhu and Shan [16] employed this facility to check service oriented models developed by different vendors. Here, we extend the framework proposed by Zhu and Shan so that domain ontologies can be taken into consideration.

As illustrated in Figure 16, the original approach to checking of service-oriented models requires the models of services to be divided into the following three parts.

a. The specification of provided services. This part is published;
b. The specification of expected behavior of service requesters. This part is also published;
c. The internal design of the services provided by the vendor. This part is hidden from the public.

When checking a service model's consistency with models of the services that it requests, all parts of its model is first merged together with the second part of the models of other services. Then, the consistency and completeness constraints are applied. This approach solves the problem of model consistency and completeness checking, but it has the following drawbacks.

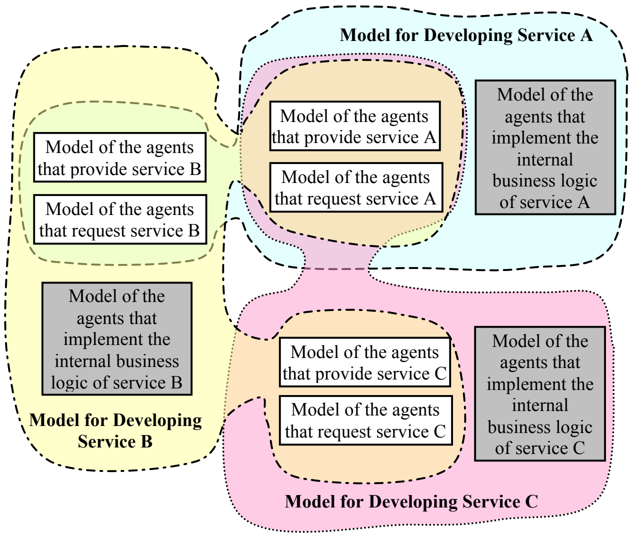

Model for Developing Service A
Model of the agents that provide service A
Model of the agents that request service A
Model of the agents that implement the internal business logic of service A
Model of the agents that provide service B
Model of the agents that request service B
Model of the agents that implement the internal business logic of service B
Model for Developing Service B
Model of the agents that provide service C
Model of the agents that request service C
Model of the agents that implement the internal business logic of service C
Model for Developing Service C

Figure 16 Zhu and Shan's Approach to Consistency Checking

First, the models of the invoked services may be not available. For many reasons, a vendor may be reluctant to publish its models such as for protecting intellectual properties. Second, when a service is discovered and linked dynamically, checking consistency at run-time may be too late. Without a standard of the design of services, it is unlikely to develop services that are consistent with each other.

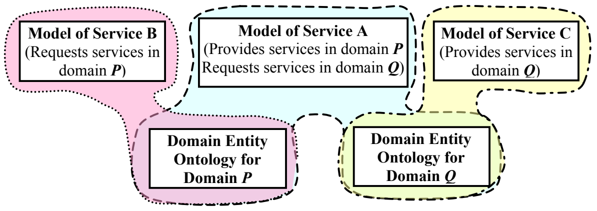

Model of Service B (Requests services in domain P)
Model of Service A (Provides services in domain P Requests services in domain Q)
Model of Service C (Provides services in domain Q)
Domain Entity Ontology for Domain P
Domain Entity Ontology for Domain Q

Figure 17 Proposed Approach to Consistency Checking

As illustrated in Figure 17, the approach proposed in this paper solves these problems by requiring all services of a particular application domain to follow a standard of domain ontology. When a service is developed, its consistency is checked against the standard and thus the applications are consistent with each other. Once the consistency is checked, there is no need to check at run-time dynamically. Moreover, it does not require the other vendors to publish their models or even a part of their models. Such consistency and completeness checking can also be supported by automated modeling tools like CAMLE.

B. *Consistency constraints*

However, we do need new consistency constraints since the existing constraints are only on CAMLE models rather than between CAMLE models and domain ontologies [26]. Here, we propose the following consistency constraints.

- *Rule SE*: The values of a symbolic entity that are passed between services must be within the valid range of values of the entity.
- *Rule SV*: Every state of a causal entity in the domain ontology that is referred to in the communications between a service provider and a service requester must be defined in the domain ontology.
- *Rule ST*: Every state transition make by a service provider on a causal domain entity must be consistent with the definition of the entity in the domain ontology.
- *Rule RB*: Every action taken by a service requester that plays a role of autonomous entity in the domain ontology must be consistent with the expected behavior as defined by the domain ontology.
- *Rule PB*: Every service provider that plays the role of an autonomous entity in the domain ontology must behave consistently with respect to the expected behavior as defined by the domain ontology.

It is worth noting that, first, in the above consistency constraints, *Rule SV*, where *SV* stands for *State Values*, does not require every state that a service provider set to a causal entity is defined by the ontology. An exception is when the state is only for the internal uses by the service. For example, a temporal state `pending for approval' could be set to an auction after it is requested by a seller but before it is formally approved by the service. Similarly, *Rule ST*, which stands for *State Transition*, does not apply to transitions only for internal uses, such as transitions to and from an internal state. However, it does require the external observation of the state transitions to be consistent to the domain ontology.

Second, *Rule RB*, where *RB* stands for *Requester's Behavior*, and *Rule PB*, where *PB* stands for *Provider's Behavior*, actually refer to a large set of consistency constraints that have been proposed and implemented by CAMLE [26,27]. Because the models of these expected behaviors of autonomous entities are modeled in CAMLE, the consistency between the domain ontology is a special case of inter-diagram consistency between CAMLE diagrams. Readers are referred to [26] for details.

Finally, *Rule SE* stands for rule for *Symbolic Values*.

C. *Completeness constraints*

In addition to the above consistency constraints, one may also want models to be complete if a service has the full capability to process certain domain entities. The following are some examples of completeness constraints.

- *Rule SE-C*: For every value of a symbolic entity that are passed between a service requester and a service provider, the receiver side must be able to process the value.
- *Rule SV-C*: Every possible state value of a causal entity defined by a domain ontology must be processed by the service.
- *Rule ST-C*: Every possible state transitions of a causal entity defined by a domain ontology must be processed by the service.
- *Rule RB-C*: A service requester must be able to process every actions taken by a service provider if the action is consistent with the expected behavior of the role that service provider is playing.
- *Rule PB-C*: A service provider must be able to process every action taken by a service requester that is consistent with the expected behavior of the role of the autonomous entity that the requester is playing.

Note that completeness constraints are not compulsory in the sense that a service does not satisfy the completeness constraints may still be regarded as a valid system although completeness are very desirable.

VI. Conclusions

Along the line of the work in [16], this paper goes deeper in the agent-oriented approach to the service-oriented modeling. The main contributions of this effort are summarized as follows.

This approach extends the agent-oriented flavor of our previous work by explicitly capturing the capability and knowledge of service agents. This is achieved by employing the domain ontologies, which facilitates modeling the changes that the service agents impose on its environment, i.e. a set of real world entities. This also helps to address a main concern of service developers on how to identify services and how to realize them. By employing domain ontology, it is natural to recognize services by considering the required changes to the domain entities in various scenarios, and to realize services by considering the implementations of the required changes of domain entities through collaborations with other autonomous entities embodied by other services.

Domain ontology is independent of the specific services, but shared by all applications of a specific domain. Employing domain ontology means that the knowledge of service agents can be based on standards of domain ontologies. This is another advantage of this approach because it addresses yet another key issue in the development of services, i.e. services owned by different vendors are often developed without sharing documents but expected to interact at run time. By including domain ontology, a service agent can have the ability to understand the meaning of the terminology used by other services. The consistency between a service and the domain ontology can be checked automatically, which implies that services based on the same ontology should be consistent with each other. Moreover, the formal reasoning of the models is possible and the properties of services can be proved.

An iterative process of service oriented modeling has been advanced, which is obviously urgently needed for developing service-oriented applications. This process is set in the context of model-driven development of services. It covers many important aspects of service-oriented development and supported by the modeling language CAMLE and its automated modeling environment. The example models used in this paper were constructed by the tool.

There are a number of issues worthy further research. We are investigating more facilities for conducting the model analysis and the verification and validation of modeled services. We are also developing facilities for model-based testing of services to incorporate domain ontologies and formal specifications. More efforts should also be made on the development of more domain ontology for capturing the precise meanings of the service capability and other models.

REFERENCES

[1] C. M. MacKenzie, K. Laskey, F. McCabe, and et al., OASIS reference model for service oriented architecture 1.0, Available online at URL: http://www.oasis-open.org/committees/tc-home.php?wg-abbrev=soa-rm.

[2] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley, SOMA: A method for developing service-oriented solutions, IBM Systems Journal, vol. 47, no. 3, pp. 377--396, 2008.

[3] L.-J. Zhang, N. Zhou, Y.-M. Chee, A. Jalaldeen, K. Ponnalagu, R. R. Sindhgatta, A. Arsanjani, and F. Bernardini, SOMA-ME: A platform for the model-driven design of soa solutions, IBM SYSTEMS JOURNAL, vol. 47, no. 3, pp. 397--413, 2008.

[4] B. Michael, Introduction to Service-Oriented Modeling: Service Analysis, Design, and Architecture. John Wiley and Sons, 2008.

[5] M. P. Papazoglou and W. J. van den Heuvel, Business process development lifecycle methodology: Bridging together the world of business processes and web services, University of Tilburg, Tech. Rep., 2006.

[6] K. Levi and A. Arsanjani, A goal-driven approach to enterprise component identification and specification, Communications of The ACM, vol. 45, no. 10, pp. 45--52, 2002.

[7] M. Endrei, J. Ang, and A. Arsanjani, Patterns: Service oriented architecture and web services, IBM, Tech. Rep., 2004.

[8] O. Zimmermann, P. Krogdahl, and C. Gee, Elements of service-oriented analysis and design, IBM, Tech. Rep., 2004.

[9] A. Arsanjani, Service-oriented modeling and architecture, 2004, Available online at URL: http://www-128.ibm.com/ developerworks/ webservices/library/ws-soa-design1/.

[10] OASIS, OASIS standard WS BPEL 2.0, Available online at URL:http://download.boulder.ibm.com/ibmdl/pub/software/ dw/specs/ws-bpel/ws-bpel.pdf.

[11] W3C, Web services choreography description language 1.0, Available online at URL: http://www.w3.org/TR/ws-cdl-10.

[12] M. Stal, Web services: beyond component-based computing, Communications of ACM, vol. 45, no. 10, pp. 71--76, 2002.

[13] H. Zhu and L. Shan, Agent-oriented modelling and specification of Web services, in Proceedings of 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005), Sedona, Arizona, USA, February 2005, pp. 152--159.

[14] M. P. Singh and M. N. Huhns, Service-Oriented Computing: Semantics, Process, Agents. Wiley, 2005.

[15] S. A. McIlraith, T. C. Son, and H. Zeng, Semantic web services, IEEE Intelligent Systems, vol. 16, no. 2, pp. 46--53, 2001.

[16] H. Zhu and L. Shan, Modelling Web Services in the Agent-Oriented Modelling Language and Environment CAMLE, International Journal of Simulation and Process Modeling, vol. 3, no. 1&2, pp. 26--77, 2007.

[17] H. Zhu, Towards an Agent-oriented paradigm of Information systems. Idea Group Inc., 2006, in Handbook of Research on Nature Inspired Computing for Economy and Management, pp. 679--691.

[18] P. Wang, Z. Jin, and L. Liu, An approach for specifying capability of web services based on environment ontology, in Proceedings of 2006 IEEE International Conference on Web Services (ICWS'06), 2006, pp. 365--372.

[19] P. Wang, Z. Jin, L. Liu, and G. Cai, Building towards capability specifications of web services based on an environment ontology, IEEE Transactions on Knowledge and Data Engineering, vol. 20, no. 4, pp.547--561, 2008.

[20] B. Wu, Z. Jin, and B. Zhao, A modeling approach for service-oriented application based on extensive reuse, in Proceedings of 2008 IEEE International Conference on Web Services (ICWS'08), 2008, pp. 754--757.

[21] L. Shan and H. Zhu, CAMLE: a caste-centric agent-oriented modelling language and environment, in Proc. of SELMAS'04 at ICSE'04, Edinburgh, UK: IEE, 2004, pp. 66--73.

[22] T. R. Gruber, A translation approach to portable ontology specifications, Knowledge Acquisition, vol. 5, no. 2, pp. 199--220, 1993.

[23] A. Maedche, Ontology Learning for the Semantic Web. Kluwer Adademic Publisher, 2002.

[24] D. L. Parnas and J. Madey, Functional documents for computer systems, Science of Computer Programming, vol. 25, no. 1, pp. 41--61, 1995.

[25] M. Jackson, Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, 2001.

[26] L. Shan and H. Zhu, Specifying consistency constraints for modelling languages, in Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'06), July 5-7, 2006, pp. 578--583.

[27] H. Zhu and L. Shan, Caste-centric modelling of multi-agent systems: The CAMLE modelling language and automated tools, in Model-driven Software Development, Research and Practice in Software Engineering, Vol. II, S. Beydeda, M. Book, and V. Gruhn, Eds. Springer, 2005, pp. 57--89.