Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 521094, 11 pages
http://dx.doi.org/10.1155/2013/521094

Research Article

An Approach for Composing Services Based on Environment Ontology

Guangjun Cai[1,2,3] and Bin Zhao[1,3]

[1] *The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *Electronics Information Engineering College, Henan University of Science and Technology, Luoyang 471023, China*
[3] *Graduate University of Chinese Academy of Sciences, Beijing 100049, China*

Correspondence should be addressed to Guangjun Cai; guangjuncai.cn@gmail.com

Received 22 May 2013; Accepted 11 July 2013

Academic Editor: Yuxin Mao

Service-oriented computing is revolutionizing the modern computing paradigms with its aim to boost software reuse and enable business agility. Under this paradigm, new services are fabricated by composing available services. The problem arises as how to effectively and efficiently compose heterogeneous services facing the high complexity of service composition. Based on environment ontology, this paper introduces a requirement-driven service composition approach. We propose the algorithms to decompose the requirement, the rules to deduct the relation between services, and the algorithm for composing service. The empirical results and the comparison with other services' composition methodologies show that this approach is feasible and efficient.

1. Introduction

Service-oriented computing (SOC) [1], with the standardized, cross-platform and transparency characteristics, can effectively reuse and compose the existing software and system resources. It comes to become a new milestone in distributed system, to be a mainstream computing model in heterogeneous environments, and to improve the competitive advantage in the field of e-commerce [2]. Service composition, as a method to use the existing services and generate new services, is one of the key factors of the success of the service-oriented computing [3].

Many institutions and enterprises have been currently engaged in the research of this area, and a series of standard language and method [3, 4] are published. The separation between the service description and the service implement, and the standardization of the service description and the service process provide the foundation to interoperate among services in various platforms. However, there are still many open problems need to be addressed for the reasons that the service is open, dynamic and changeable and that the composition problem with high complexity [4]. We believe that by solving the following problems the efficiency of the service composition can be improved.

(i) Lacking of effective model's support. Some methods (such as [5]) do not use the model, while others (such as [6]) are based on a service-oriented process model changed with the change of services.

(ii) Few requirements and domain knowledge. Many methods (e.g., [2]) need no constraints on the behaviour of the composition services, and the domain knowledge is generally only used in the detection of match between services (such as [7]).

(iii) Lacking support to the composition of the large-grained or precise services. For example, [8] can only composite the atomic services as a composite object.

The environment model completely independent of the service implement provides a basis to acquire a precise and on-demand requirement. Based on it, a requirement driven approach is proposed which with the requirement descriptions and the service descriptions based on environment ontology as inputs. This approach firstly reduces

Table 1: Meanings of the concepts.

Concept	Meaning
Environment entity	Entities which a system will interact with
Causal entity	Entities whose properties include predictable causal relationships among its shared phenomena
Autonomous entity	Entities which can receive or send message autonomously usually consist of people
Symbolic entity	Entities which are a physical representation of data
Attribute	A named variable for characterizing a static property of an entity
Static attribute	Characterize the static properties of the environment entity
Dynamic attribute	Characterize the dynamic properties of the environment entity
Value	Intangible individual entity that exists outside time and space, and is not subject to change
Tree-like hierarchical state machine	A hierarchical state graph for characterizing dynamic properties of a causal entity
Finite state machine	A directed state graph for characterizing a dynamic property
Divide	Relation between the basic state machine and a state
State	Value of an entity at a given time
Transition	A state change in a state machine represent the relation between two states
Message	The content that is sent or received from one environment entity to another
Command message	Contain the action command
Data message	Message that has no command contains only the parameters
Interaction	Observable shared phenomenon between a system and its environment entity
Message interaction	Interaction that occurs through the messaging
Value interaction	Interaction that occurs through reading or assigning the value of the static attribute

the complexity of the problem through the decomposition of the requirement in order to improve the degree of parallelism of the service composition. Then, the rules to determine the relations between the services are proposed to improve the granularity of the service. Finally, it can present the composition result in the WS-CDL [9].

The rest of this paper is structured as follows. In Section 2, we introduce environment ontology and a usage scenario and present the environment ontology-based descriptions of some services and requests. Section 3 proposes some algorithms decomposing the requirement, some rules to determine the relations between services, and the method to compose the services. Section 4 presents some implement results and analyse it. After discussing some related work in Section 5, we conclude in Section 6.

2. Environment Ontology and the Description Based on Environment Ontology

The idea of environmental modelling originates from requirements engineering. Reference [10] pointed out that the semantics of software systems is in the environment of the software rather than in the software itself. Then, [11] introduces it in service-oriented computing and uses it as the basis to characterize the services and the user requirement.

2.1. Environment Ontology. The environment ontology describes the nature of the entity of the service problem, as well as its possible changes, the process of the change, the conditions, and the result of the change. Environment the service entities interact with consists of various environment entities. Environment entities, either concrete or abstract, shared the same semantic with all the relevant service and the user requirements rather than relying on a service or requirement. The top-level concepts are shown in Table 1.

The various types of environmental entities require a different form of the description except for static properties and the interaction. Causal entities have a description of the dynamic properties in the form of the tree-like hierarchical state machines; symbolic entities and autonomous entities are with a data set and a set of events, respectively. Part of the concept is defined as follows.

Define 1. Finite state machine (Fsm) describing the changes in one causal entity is a four-tuple ⟨States, Trans, init, fin⟩, where *States* is a set of the state; *Trans*, a set of transitions, is defined as $\{\langle ss, mess, ts\rangle \mid ss, ts \in$ States, *mess* denote the message set$\}$; *Mess*, a set of messages, is defined as $\{\langle dir, mes\rangle \mid dir = \downarrow|\uparrow$, *mes* is a data message *data* or a command message *event*$\}$, Finally, $init, fin \in States$, is the initial state and the target state, respectively.

Define 2. Tree-like hierarchical state machine (Hsm), to describe the state change of an individual controllable entity, is a triple ⟨Fsms, Divs, rootfsm⟩, where *Fsms* is a set of *fsm*; $Divs = \{\langle s, fsm\rangle \mid s$ is a state, $s \notin fsm \wedge s \in hsm\}$, where *s* is called the parent state of *fsm*; $rootfsm \in Fsms$.

Define 3. The effect of the environment entity denoted *Eff*, describe the changes of the environmental entity, is defined as $Events \mid Datas \mid Hsm$, where $Eff := Events := \{\langle dir, event\rangle \mid dir = \downarrow|\uparrow$, *event* is a command message$\}$ when the entity is an autonomous entity; $Eff := Datas := \{\langle dir, data\rangle \mid dir = \downarrow|\uparrow$, *data* is a data message$\}$ when the entity is a symbolic entity; $Eff := Hsm$ when the entity is a causal entity.

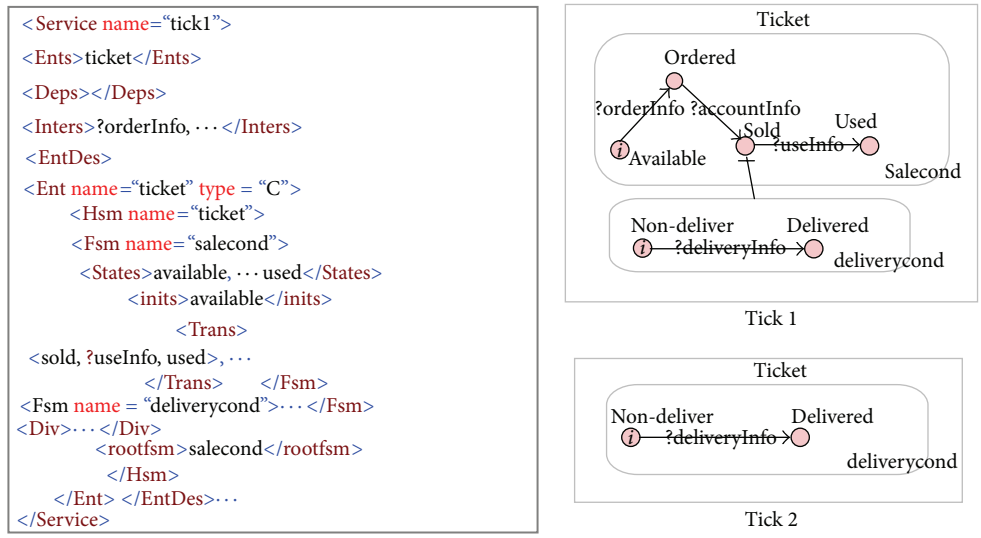


Figure 1: The description of the available service based on environment ontology.

Define 4. Environmental entity (EnvEnt) defined as a four-tuple ⟨Name, Type, AttrSet, Eff⟩, where; *Name* is the name of the environment entity; Type := "S" | "C" | "A" represents the environment entity, respectively, representing a symbolic entity, the causal entity, and autonomous entity; *AttrSet* := {⟨attrn, attrv⟩ | *attrn* is an attribute name and *attrv* is an attribute value}; *Eff* describes the change of the environment entity.

2.2. Describing the Service Based on Environment Ontology

Define 5. The service description based on environment ontology characterizes the semantics of the services through the physical changes the service causes. It can be defined as ⟨name, Ents, Inters, Deps, EntDes, Sers, and Funcs⟩, where *name* is a service name; *Ents* is a set of the environment entities; *Inters* is a set of interactions and it is a subset of the messages; $Deps = \{\langle se, inter, te\rangle \mid se, te \in Ents$ is, respectively, the source entity and the target entity; $inter \in Inters\}$ is generated by the effect of the service between the environment entities; $EntDes := \{e' \mid e'$ describes part of one environment entity $e \in Ents\}$ is the description of the environmental entity under the effect of the service; $Sers = \{ser \mid ser$ is a service$\}$ is a set of services; $Funcs := \{\langle func, ser\rangle \mid func \in Ents \cup Hsms \cup Fsms \cup Trans \cup Datas \cup Events, ser \in Sers\}$ is a set of functions describing the relation between each functionality and the services provide it.

The process of the service description is a process projecting service functionality to the environment entities. The input of the process consists of the Ents, Inters, and the environment ontology; the output of the process consists of Deps and EntDes. The specific algorithm can see [11, 17]. Figure 1 lists the text description and the graphics description of the two ticketing services.

2.3. Describing the Requirement Based on Environment Ontology

Define 6. The requirement description based on environment ontology can be seen as a service description without the service's implementation. It is defined as a quintuple ⟨name, Ents, Inters, Deps, EntDes⟩, where the meaning of the elements is the same with that in Define 5.

The process to describe the requirement is similar to that which depicts services. There are two results when the requirement is described: the requirement is described with only one option, or description with more than one option. The latter case needs more user constrain. Figure 2 presents a specific illustration of the requirement, wherein the text description is presented in (a) and the effect of changes of the ticket in (b).

3. Composing the Services Based on Environment Ontology

The task of web service composition is to search appropriate service and to arrange them in an appropriate manner. For the requirements which are depicted in the same level with the composition result, we present a method through decomposing the requirement to search the available service based on the service discovery method in [11]. The task is divided into three parts: decomposing the requirement, determining the relation between the services, and generating the composition service.

3.1. Decomposing the Requirement. The nature of the composition problem is that one problem cannot be addressed by a single service. Furthermore, our method is based on a requirement described in a formal detailed determine way.

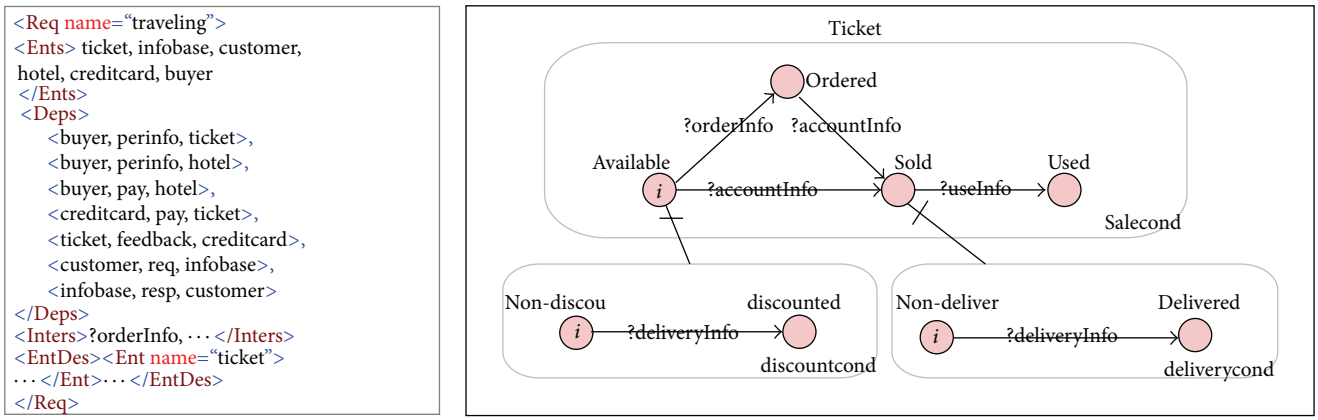


Figure 2: The requirement description based on environment ontology.

Thus we can acquire a composite service through a way which firstly decomposes the request into various parts and then composes services according to the requirement.

3.1.1. Decomposing the Requirement Involving Multienvironment Entities. In this section, we decompose the requirement among environment entities and do not consider the internal description of the environment entity. The type of dependency among the environment entities can be classified into four classes: no dependence, dependent on the environment entities in one direction, and dependent on a symbol entity or autonomous entity, and dependent on each other. The corresponding decomposition algorithms are shown in Algorithms 1, 2, 3, and 4.

The elements in *ReqSet* produced by Algorithm 1 have no dependence on other entities. Algorithm 1 satisfies consistency and has order of n^2 time complexity where n is the number of entities in the requirement. Algorithm 2 decompose the requirement according to whether the interaction between the entities is in one direction. When multiple environment entities meet the conditions of the decomposition, the consistency can only satisfy all the decompositions with the same order. Its time complexity is the same with Algorithm 1. Algorithm 3 decomposes the requirement according to the type of the entity and has the same consistency and time complexity with Algorithm 2. Algorithm 4 is used when there is only message dependence among different entities. It can meet the consistency and its time complexity is equal to $O(n)$ where n is the number of entities. Taking the requirement in Figure 2(a) as the input to present how to use these algorithms. The decomposition results are listed in Table 2.

3.1.2. Decomposing the Requirement Involving One Environment Entity. This section decomposes the requirement in one environment entity. Autonomous entity or symbolic entity having no constrains on its interaction can be directly divided into individual event or data by Algorithm 7, and no further discussion is required. But in a causal entity, which is specified as a tree-like hierarchal state machine, there exists the dependence between a state and a tree-like hierarchal state machine, the dependence between a state and a finite state machine, and the dependence between states. The decomposition algorithms are shown in Algorithms 5, 6, and 7.

Algorithm 5 decomposes the requirement containing a single *hsm* based on the couple between the finite state machines with the same state as an ancestor greater than the couple between the finite state machines with the different state as ancestor. The result of it is a set of the requirement each involving one *hsm*, and it can be called repeatedly several times until each includes only one *fsm*. Algorithm 5 satisfies consistency and has the time complexity $O(f)$, where f, less than or equal to the number of *fsm* in the *hsm*, is the number of the requirements the decomposition produces.

Algorithm 6 decomposes the requirement containing a *fsm* based on the key state into more than one *fsm*. The key state can be specified by the user, the initial state of *fsm*, and the termination state of *fsm* or the parent state of the finite state machine contain one key state. It satisfies consistency and has the time complexity $O(m^2)$, where m is the number of states in the *fsm*.

Algorithm 7 can be used to decompose the requirement containing a single autonomous entity, a single symbolic entity, or an individual *fsm* based on the basic functionality unit of the requirement. It satisfies consistency and has the time complexity $O(k)$, where k is the number of the basic functionalities.

Taking the requirement in Figure 2(b) as the input to present how to use Algorithms 5–7. The decomposition results are listed in Table 3.

3.2. Determining the Relations of the Services. The second task of web services composition is to determine the relations among services so that they can collaborate with each other for satisfying the request. To determine the relations among services is to derive the relationship between the various services based on the relations of the functionalities in the

```
DecByDep(Req, ReqSet)    //input Req, output ReqSet
(1)  while(Req.Ents ≠ ∅)
(2)     e:= get (Req.Ents);
(3)     if(Relate(e) ≠ Req.Ents)//Relate(e) refer to a set include e and the entities depend on e
(4)        Set_e:= create (Relate(e)); //create a set include the elements in Relate(e)
(5)        ReqSet:= ReqSet ∪ create(Set_e, Req); //create(Set_e, Req) is to acquire a requirement corresponding to the Set
(6)        Req:= Req.Remove(Set_e); //Removing a requirement corresponding to the Set from Req
(7)     end if
(8)  end while
(9)  ReqSet:= ReqSet ∪ Req;
```

Algorithm 1: Decomposition algorithm among unrelated environmental entities.

```
DecByDepDir(Req, ReqSet)
(1)  if(∃e((e ∈ Req.Ents)∧∀dep(des ∈ Req.Deps∧(dep.se ≠ e∨dep.te ≠ e))))
(2)     ReqSet:= create (e, Req);
(3)     ReqSet:= ReqSet ∪ DecByDep (Req.Ents-{e});
(4)  end if
```

Algorithm 2: Decomposition algorithm according to the dependent type between environmental entities.

```
DecByEntTyp(Req, ReqSet)
(1)  if(∃e(e ∈ Req.Ents∧(type(e) = A∨type(e) = S)))
(2)     ReqSet:= create(e, Req);
(3)     ReqSet:= ReqSet∪DecByDep(Req.Ents-{e});
(4)  end if
```

Algorithm 3: Decomposition algorithm according to the type of environmental entities.

requirement description as well as the relations between the functionalities and services. The rules are given in this section in two layers: to determine the relations between the services among multienvironment entities and to determine the relations between services in one environment entity. They will help to use the service in more appropriate granularity, to reduce complexity of the external control flow, and to reduce the number of services in composition services and the communications overhead between the services.

3.2.1. Determine the Service Relations among Multiple Environment Entities. If the requirement involving multienvironment entities needs to be accomplished by several services instead of one available service, we need to identify the relations between the services.

Let *inter* and *inters* be, respectively, an interaction and a set of the interactions between environment entities and *type(inter)* the type of *inter*, whose value could be *msg* or *val*. *R(inter)* and *S(inter)*, respectively, represent the environment entity receiving *inter* and that sending *inter*. |*inters*| and |*con(inters)*| represent the number of interactions in *inters* and the number of the messages in *inters*, respectively. *Type(ent)* denotes the type of the environment entity *ent*. Determining the relations between the services involving multienvironment entities are shown in Box 1.

Rule 1 shows that when the multiple value interactions act on the same property of the same environment entity and wherein the received value interaction, the order of the value interactions is subject to the constraints of the received value interaction. Rule 2 shows that when the multiple message interactions act on one causal entity, the order of the message interactions is subject to the constraints of the causal entity. Rule 3 shows that when the multiple message interactions act on one autonomous entity, the order of the message interactions is subject to the constraints of the autonomous entity. Rule 4 shows that when the multiple message interactions act on one symbol entity and wherein one received interaction, the order of the message interactions is subject to the constraints of the received interaction. The time complexity of these rules is up to $O(h^2)$, where h is the number of interactions.

Back to the example in Figure 2(a), there is no interaction set to meet Rules 1 and 4. The result produced by Rule 2 is {⟨{⟨creditcard, pay, ticket⟩, ⟨ticket, feedback, creditcard⟩}, ticket⟩, ⟨{⟨creditcard, pay, ticket⟩, ⟨ticket, feedback, creditard⟩}, creditcard⟩}. The result produced by Rule 3 is {⟨{⟨customer, req, infobase⟩, ⟨infobase, resp, customer⟩}, customer⟩}.

3.2.2. Determine the Service Relations in One Environment Entity. The task of this section focused on the causal entity described by the tree-like hierarchical state machine. We need to determine the relations among the functionalities which associate with the services acting on one causal entity. If the relation is between the services, we need to divide the functionality in some time. If the relation is in a service, we need to merge the functionalities into one service. According to the structure of the causal entity, this task can be divided

```
DecByEnt(Req, ReqSet)
(1)  for(each e in Req.Ents){ // the number of environmental entities in Req.Ent greater than 1
(2)      ReqSet:= ReqSet ∪ create(e, Req);
(3)  end for
```

Algorithm 4: Decomposition algorithm among the various environmental entities.

Table 2: Decomposition result of each step by Algorithms 1–4.

Algorithm	Decomposition result
Algorithm 1	{infobase, customer}, {hotel, creditcard, ticket, buyer}
Algorithm 2	{infobase, customer}, {buyer}, {hotel}, {creditcard, ticket}
Algorithm 3	{infobase}, {customer}, {buyer}, {hotel}, {creditcard, ticket}
Algorithm 4	{infobase}, {customer}, {buyer}, {hotel}, {creditcard}, {ticket}

```
DecByHsmStr(Req, ReqSet) // input Req, which can be represented by a Tree-like hierarchal state machine(thsm), output ReqSet
(1)  rootfsm:= GetRootfsm(Req);
(2)  for(each subhsm of rootfsm)//subhsm.rootfsm.superstate ∈ rootfsm
(3)      ReqSet:= ReqSet ∪ create(subhsm, Req);
(4)  end for
(5)  ReqSet:= ReqSet ∪ create(rootfsm, Req);
```

Algorithm 5: Decomposition algorithm based on the structure of the *hsm*.

```
DecByState(Req, ks, ReqSet)// input Req which can be represented by a finite state machine and ks, output ReqSet
(1)  if((Req.State-(From(ks) ∪ To(ks))) = ∅ ) //Req.State is a set of the states in Req
(2)      From(ks):= From(ks)-{ks}; //From(ks) is a set of the states which ks can reach
(3)      To(ks):= To(ks)-{ks}; //To(ks) is a set of the states which can reach ks
(4)  end if
(5)  Req1 = create(Req.State-(From(ks) ∪ To(ks)), Req);
(6)  Req2:= create(From(ks)-To(ks), Req);
(7)  Req3:= create(To(ks)-From(ks), Req);
(8)  Req4:= create(From(ks) ∩ To(ks), Req);
(9)  ReqSet:= Req1 ∪ Req2 ∪ Req3 ∪ Req4;
```

Algorithm 6: Decomposition algorithm based on the key state.

```
DivByBeha(Req, ReqSet)// input Req which can a autonomous entity, a symbolic entity or a fsm, output ReqSet
(1)  for(each b in Req)// b is the basic unit of the functionality, which can be a data, an event or transition
(2)      ReqSet:= create({b}, Req);
(3)  end for
```

Algorithm 7: Decomposition algorithm based on the basic unit of the functionality.

Table 3: Decomposition result of each step by the decomposition Algorithms 5–7.

Algorithm	Inputs	Decomposition result
Algorithm 5	Figure 2(b)	{salecond}, {discountcond}, {deliverycond}
Algorithm 6	Salecond in Figure 2(b), ordered	{<available,. . .,ordered>, <ordered,. . .,sold>, <available,. . .,sold>}, {<sold,. . .,used>}
Algorithm 7	Salecond in Figure 2(b),	{<available,. . .,ordered>}, {<ordered,. . .,sold>}, {<available,. . .,sold>}, {<sold,. . .,used>}

Rule 1: (|inters|>1) ∧ (type(inters)=val) ∧ ∀inter((inter∈inters) → (R(inter)=ent ∨ S(inter)=ent)) ∧ (|con(inters)|=1) ∧ ∃inter ((inter∈inters) ∧ R(inter)=ent) => constrain(inters,inter) ∧ (R(inter)=ent).
Rule 2: (|inters|>1) ∧ (type(inters)=msg) ∧ ∀inter((inter∈inters) → (R(inter)=ent ∨ S(inter)=ent) ∧ (type(ent)=C)) => constrain (inters,ent).
Rule 3: (|inters|>1) ∧ (type(inters)=msg) ∧ ∀inter((inter∈inters) → (R(inter)=ent ∨ S(inter)=ent) ∧ (type(Ent)=A)) => constrain (inters, Ent).
Rule 4: (|inters|>1) ∧ (type(inters)=msg) ∧ ∀inter((inter∈inters) → (R(inter)=ent ∨ S(inter)=ent)) ∧ (type(ent)=S)) ∧ (|con(inters)| =1) ∧ ∃inter((inter∈inters) ∧ R(inter)=ent) => constrain(inters, inter) ∧ (R(inter)=ent).

Box 1

Rule 5: <s,fsm>∈Divs ∧ Sers(Pres(s))=(Sers(Sucs(s))) ∧ |Sers(Sucs(s))|=1 ∧ (Sers(Sucs(init(fsm))) ≠ Sers(Pres(s)) ∨ Sers (Pres(fin (fsm))) ≠ Sers(Sucs(s))) => Divide(Sers(Sucs(s)), s).
Rule 6: <s, fsm>∈Divs ∧ Sers(Sucs(init(fsm))) ≠ Sers(Pres(s)) => Divide(Sers(Sucs(init(fsm)))∩Sers(Pres(s)),fsm); <s,fsm>∈Divs ∧ Sers(Pres(fin(fsm))) ≠ Sers(Sucs(s)) => Divide(Sers(Pres (fin (fsm)))∩Sers(Sucs(s)), fsm).
Rule 7: <s,fsm>∈Divs ∧ Sers(Pres(s))=Sers(Sucs(init(fsm))) ∧ |Sers(Pres (s))|=1 => Merge(Pres(s), Sucs(init(fsm))); <s,fsm>∈Divs ∧ Sers (Sucs(s))=Sers(Pres(fin(fsm))) ∧ Sers(Sucs(s))|=1 => Merge (Sucs(s), Pres (fin(fsm))).
Rule 8: Sers(Pres(s))=Sers(fsm) ∧ |Sers(Pres(s))|=1 ∧ ∀fsm(<s, fsm>∈Divs → (Sers(fsm)=Sers(Pres(s)))) => Merge(Pres(s),fsm); (Sers (Sucs(s))=Sers(fsm)) ∧ |Sers(Sucs(s))|=1 ∧ ∀fsm(<s,fsm>∈Divs → (Sers(fsm) =Sers (Sucs(s)))) => Merge (Sucs(s), fsm).

Box 2

Rule 9: (|Pres(s)|=|Sucs(s)|=1) ∧ (Ser(Pres(s))=Ser(Sucs(s)))) => Merge(Pres(s),Sucs(s)).
Rule 10: BranchEntry(b) ∧ BranchExit(e) ∧ (source(t) =source(t′)=b) ∧ (target(t) =target(t′)=e) ∧ (Ser(t)=Ser(t′)) => Merge(t, t′).
Rule 11: BranchEntry(b) ∧ |Sers(Sucs(b))|=1 => Move(Sers (Sucs(b)), b).
Rule 12: (|Pres(s)| >1 ∨ (|Sucs(s)| >1)) ∧ (Ser(Sucs(s))∩Ser(Pres(s)) ≠ ∅) ∧ (|Sers(Sucs(s))| >1 ∨ (|Ser(Pres(s))| >1)) => Divide (Ser(Sucs(s))∩Ser(Pres(s)),s).
Rule 13: s∈From(s) ∧ t∈Pres(s) ∧ source(t)∉From(source(t)) ∧ t′∈Pres(s) ∧ source(t′)∈From(source(t′)) ∧ Ser(t)=Ser(t′) => Divide (Ser (t), s).
Rule 14: s∈From(s) ∧ ((t∈Sucs(s)∧target(t)∈From(target (t))) ∧ (t′∈Sucs(s) ∧ target(t′)∉From(target(t))) ∧ (Ser(t) =Ser(t′)) => Divide (Ser(t),s).

Box 3

into the transformation between the finite state machine and the transformation in a finite state machine.

Let s a state, t and t' a transition, *Pres(*s*)* a set of transitions which ending with s, and *Sucs(*s*)* a set of transitions which starting from s. *Ser(*t*)* and *Sers(fsm),* respectively, represent $\langle t, ser\rangle \in funcs$ and $\langle fsm, ser\rangle \in funcs$. $|set|$ indicates the number of elements in the *set*, such as $|Pres(s)|$ which is the number of the transitions in *Pres(*s*)*. *init(fsm)* and *fin(fsm),* respectively, denote the initial state and the final state of the *fsm,* and *source(t)* and *target(t)* represent the source state and the target state of the transition t. $Merge(t, t')$ denotes merging the two transitions, *merge(t, fsm)* denotes merging t and *fsm*, *divide(ser,s)* represents dividing the *ser* by s, *divide(ser, fsm)* represents dividing the *ser* on the boundary of *fsm*, *move(ser, b)* represents move of the b to the end of *ser*.

(1) The Transformation between the Finite State Machines. The transformation rules determining the relations of the services acting on a tree-like hierarchical state machine are shown in Box 2.

Rules 5 and 6 are used to determine, respectively, whether the service needs to be divided by the parent state of a finite state machine and whether the service acting on several finite state machines needs to be divided by the boundary of the finite state machine. Rules 7 and 8 can be used to merge the services acting on several finite state machines. All the rules of this section can be completed in polynomial time n which equals the number of *states* plus the number of *transitions* contained in the relevant *fsm*.

(2) The Transformation in a Finite State Machine. When a finite state machine associates with several services, we determine the relations between these services. Thus, our task in this section is to conclude the relations between the services according to the relation between transitions and the relation between the transition and the services. Before introducing the transform rules, we give some definitions firstly.

Define 7. The branch entry point is denoted as $BranchEntry(s)$iff $(|Sucs(s)| > 1) \wedge (|Pres(s)| = 1) \wedge (s \notin To(s))$.

```
Input: Req //the requirement description based on environment ontology
Output: Ser //the composition service
(1)  ReqSet:= {Req}; //ReqSet is a set of the requirement needing to be implemented by the services;
(2)  while(ReqSet ≠ ∅)
(3)    for(each req in ReqSet)
(4)      ser:= discovery(req, SerSet);
(5)      if(ser ≠ ∅){//discovery(req, SerSet) successes
(6)        LabReq:= LabReq ∪ label(req, ser); //label the req using the ser
(7)        continue;
(8)      end if
(9)      ReqSet:= ReqSet ∪ dec(req, rule)-{req}; // dec() is used to decomposition the requirement
(10)   end for
(11) end while
(12) Ser.{name, Ents, Deps, EntDes}:= req{name, Ents, Deps, EntDes}
(13) determin the Sers and Funcs of the Ser according to the req and LabReq;
(14) for(each ent in LabReq.Ent)
(15)   if(ent.type = C&&|Sers(ent)| > 1)
(16)     for(each fsm in rhsm)
(17)       if(|Sers(fsm)| > 1&&hasproced(subFsm(fsm)))
(18)         adjust the Sers and Funcs of the Ser using the rule 9–14;
(19)         adjust the Sers and Funcs of the Ser using the rule 5–8;
(20)       end if
(21)     end for
(22)   end if
(23)   adjust the Sers and Funcs of the Ser using the rule 1–4;
(24) end for
(25) generateCDL(Ser);//generate the description in WS-CDL of the ser
```

Algorithm 8: Composition algorithm based on environment ontology.

Define 8. The branch exit point is denoted as $BranchEixt(e)$ if and only if the state e satisfies $BranchEntry(b) \wedge ((From(b)\text{-}(To(e) \cup From(e))) = \varnothing) \wedge (\forall s(s \in (To(e) \cap From(b)) \rightarrow ((From(b)\text{-}(To(s) \cup From(s))) \neq \varnothing)) \wedge (e \notin From(e))$.

The rules are shown in Box 3. Rule 9 orients the sequence structure mergers the services in a state which has a single pervious transition and a single successor transition. Rules 10, 11, and 12 orient the choice structure, in which Rules 10 and 11 are used to merge the branches, and the Rule 12 is used to divide the service. Finally, Rules 13 and 14 orient the loop structure, in which Rule 13 is used at the entrance of the loop and Rule 14 is used at the exit of loop when one service acts on both in the loop and outside the loop. All of them belong to the polynomial time complexity class.

3.3. Generating the Composition Service. The algorithm generating the composition service is shown in Algorithm 8. It consists of two main phases. The first phase decomposes the requirement and discovery of the corresponding services in lines 1–11. The discovery process can be accomplished by the method proposed in [11] or some other service discovery or composition method. The second constructs the composition service and adjusts the relation between the services in lines 12–25.

```
<Service name="traveling">
  <Ents>ticket, hotel</Ents>
  <Deps><buyers, perinfo, ticket>,⋯ </Deps>
  <Inters>?orderInfo, ⋯ </Inters>
  <EntDes>
    <Ent name="ticket" type="C">⋯ </Ent>⋯
  </EntDes>
  <Sers>tick41, tick42, tick2, tick3,⋯ </Sers>
  <Funcs><Func> <Ent name="ticket"> <Hsm name="ticket">
          <Fsm name="salecond">
        <Tran> <available, {?orderInfo, ⋯ }, sold> </Tran>
    </Fsm> </Hsm> </Ent>
  <Sers>tick41</Sers>
  </Func></Funcs>⋯
</Service>
```

Figure 3: Part of the description of the composition service.

As that analyzed in previous part, the time complexity of Algorithms 1–7 and the Rules 1–14 belongs a polynomial complexity. In Algorithm 8, the execution number of Algorithms 1–4 does not exceed the number of the environmental entities in the requirement, the execution number of Algorithms 5–7 does not exceed the total number of the basic behaviors, the execution number of the Rules 1–4 does not exceed the total number of the interaction, and the execution number of Rules 5–14 does not exceed the number of states. Thus, Algorithm 8 is a polynomial complexity algorithm. Part of the description of the composition service generated from the requirement in Figure 2 is shown in Figure 3.

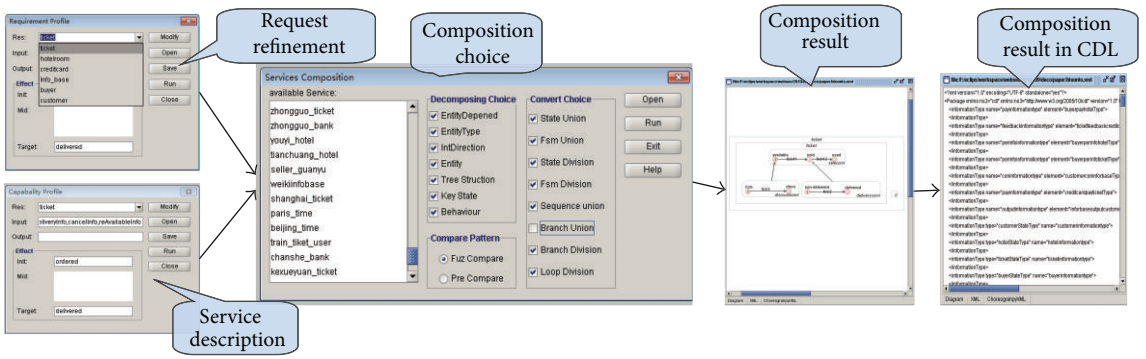
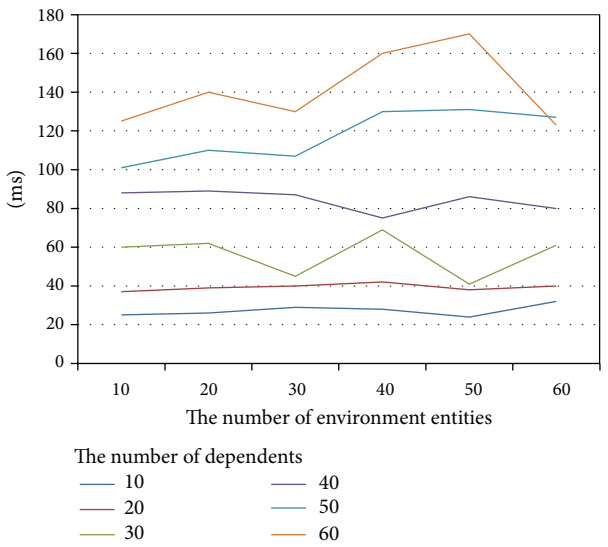
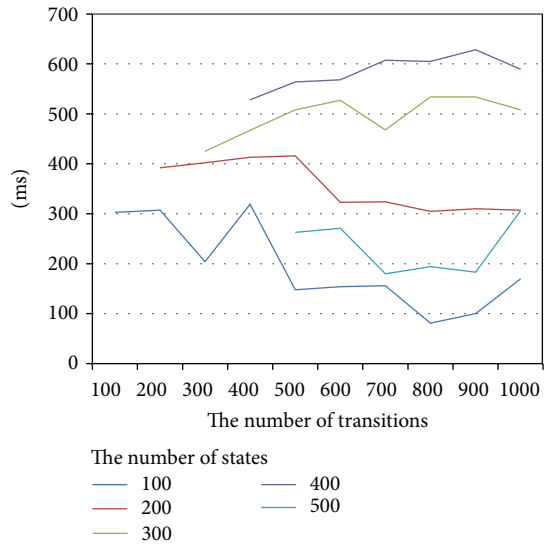


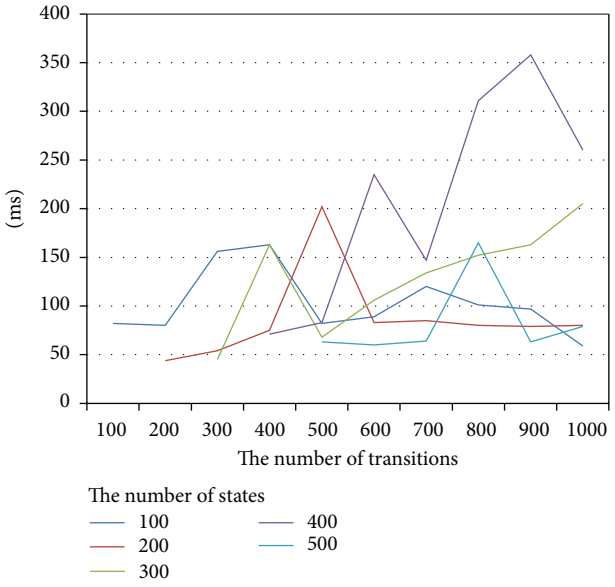
FIGURE 4: The main interface and the composition result of a prototype.



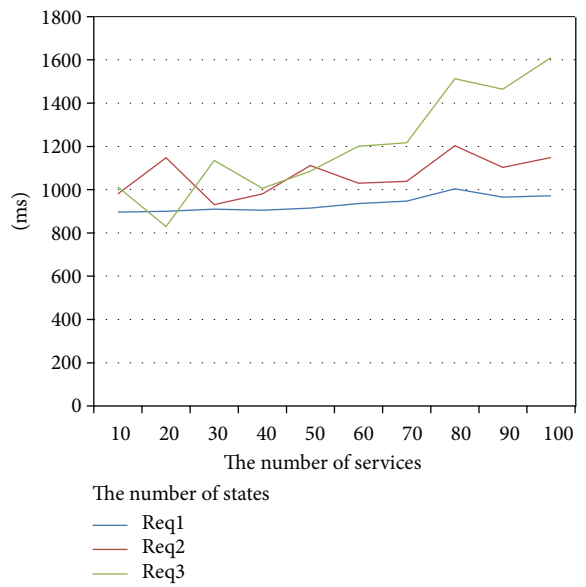
(a) The time to decompose the requirement among environment entities



(b) The time to decompose the requirement in an environment entity



(c) The time to transform the service model



(d) The time to compose services

FIGURE 5: The figure of composition time.

Table 4: Comparison of various approaches to service composition.

Approach	Service description	Requirement description	Description object	Service type	Approaches	Others	Complexity
Berardi et al. [12]	Behavior	B	Service	B, A	Behavior equivalence	Single community	Exponential
McIlraith and Son [2]	IOPEB	OE	—	A	Agent	User constrain	—
Bultan et al. [13]	B	B	Conversation	B, A	Behavior equivalence	—	—
Roman et al. [14]	IOPEB	IOPEB	Service, goal	—	Mediator	Domain ontology	Exponential
Maamar et al. [5]	IOB	O	Context	A	Agent	Multilevel context	—
Sirin et al. [8]	IOPE	OE	—	A	Complex service-based decomposition	Complex and simple service	—
Brogi et al. [6]	IOB	O	—	C, A	Graph constructing and coloring	—	Exponential
Arpinar et al. [7]	IO	IO	Domain, service and process	A	Matching and searching	—	—
Liang et al. [15]	Qos	Qos	Service	A	Matching	Constrain	Polynomial
Tao et al. [16]	IO, Qos	IO, Qos	Service, task	A	Decomposition and discovery	Qos	Polynomial
Cai [17, 18]	IOPEB	IOPEB	Environment	A, B, C	Decomposition and transformation	Environment ontology	Polynomial

4. The Implementation and Analysis

To achieve the service composition described in this paper, the six modules need to be implemented. The first one is used to refine the request into the requirement based on the environment ontology. The second is used to describe a service based on the environment ontology. The third is to decompose the requirement. The fourth is used to discovery and matching of the service. The fifth is to determine the relationship between the services, and the final is to generate the composition result described in the appropriate manner. The main interface and the composition result of a prototype we implemented in java are shown in Figure 4.

The required times the algorithms run on our prototype platform are shown in Figure 5. It is consistent with our previous analysis. The reasons the proposed method could improve the efficiency of the composition can be concluded as follows.

(i) The first one is the enrichment of the composition knowledge introduced by the description of the requirement and the service based on environment ontology. The former limits the position of the service to be used, while the latter can help to eliminate some paths that are just possible in theory but not allowed by the domain knowledge or the user.

(ii) The second lies in the hierarchal structure. The position and the order to decompose the requirement to some extent determine the location and the priority to composite the services. The decomposition also helps to improve degree of parallelism.

(iii) Thirdly, this approach can support the composition of the composite service and behavior services.

Besides these, this work contributes to expand the range of the service description into the problem supporting to introduce the more precise Qos constrain in the future work, and wherein some work can be used to other domains such as the rules determining the relationship between services which can be used for processing control flow.

5. Related Work

This section briefly discusses the relationships between our works with the existing service composition approaches. The differences between ours with others are illustrated in Table 4, where I, O, P, E, B and "—" denote input, output, precondition, effect, behavior, and unspecified content explicitly, respectively. Instead of focusing on the description of Web services of their own, we give attention to the effects imposed by the services on their environment entities and state that all the capabilities are based on the environment entities, whose characteristics and interconnections are observable and applicable during service discovery and composition. And for the character of requirement decomposition and discovery, the approach can adapt for composing different type services by changing the discovery constrains.

6. Conclusion and Future Work

In order to solve the problems of the service composition efficient in the single problem domain, this paper proposes the methods to generate requirement, to describe the service, and to compose the service based on environment ontology. Compared with the existing efforts in this field, this work advances the state of art in the following aspects.

(i) More efficient composition: the domain knowledge described in environment ontology and the method to decompose the requirement not only can reduce the size of the problem, but also help to improve the degree of parallelism.

(ii) More types of services composition: the method to determine the service relationship according to the problem model not only can support a composition of atomic services but also support the composition of the composite services and the behavior services.

(iii) Optimized composition model: the method adjusting the relation between the services based on the composition structure provides the foundation to schedule localization and optimizes the execution model.

In addition, the prototype implementation can generate a composition result described by the WS-CDL. This paper presents an ongoing work for tackling the issue of automatic service composition. Subsequent work will extend the ontology for supporting the service composition in various domains and the multilevel problem. And then we will enhance the service composition procedure for considering the nonfunctional concerns and the correctness of the composite services. Moreover, as stated in [1], the transaction and security must be considered if we would use it in reality. Finally, some other application domains will be researched to use this method.

Acknowledgments

This work is partially supported by the National Natural Science Fund for Distinguished Young Scholars of China under Grant no. 60625204, the Key Project of National Natural Science Foundation of China under Grant nos. 60736015 and 90818026, and the National 973 Fundamental Research and Development Program of China under Grant no. 2009CB320701.

References

[1] M. P. Papazoglou and D. Georgakopoulos, "Service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 25–29, 2003.

[2] S. McIlraith and T. C. Son, "Adapting Golog for composition of semantic Web services," in *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning*, pp. 482–496, Toulouse, France, 2002.

[3] N. Milanovic and M. Malek, "Current solutions for Web service composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.

[4] R. Jinghai and S. Xiaomeng, "A survey of automated Web service composition methods," in *Semantic Web Services and Web Process Composition*, pp. 43–54, Springer, Heidelberg, Germany, 2005.

[5] Z. Maamar, S. K. Mostéfaoui, and H. Yahyaoui, "Toward an agent-based and context-oriented approach for Web services composition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 5, pp. 686–697, 2005.

[6] A. Brogi, S. Corfini, and R. Popescu, "Semantics-based composition-oriented discovery of Web services," *ACM Transactions on Internet Technology*, vol. 8, no. 4, pp. 1–39, 2008.

[7] I. B. Arpinar, R. Zhang, B. Aleman-Meza, and A. Maduko, "Ontology-driven Web services composition platform," *Information Systems and e-Business Management*, vol. 3, no. 2, pp. 175–199, 2005.

[8] E. Sirin, B. Parsia, D. Wu, J. Handler, and D. Nau, "HTN planning for Web service composition using SHOP2," *Web Semantics*, vol. 1, no. 4, pp. 377–396, 2004.

[9] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, Web Services Choreography Description Language Version 1.0, 2005, http://www.w3.org/TR/ws-cdl-10/.

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

[11] P. Wang, Z. Jin, L. Liu, and G. Cai, "Building toward 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.

[12] D. Berardi, D. Calvanese, G. D. Giacomo et al., "Automatic composition of eservices that export their behavior," in *Proceedings of the 1st International Conference on Service-Oriented Computing (ICSOC '03)*, pp. 43–58, Springer, Trento, Italy, 2003.

[13] T. Bultan, X. Fu, R. Hull, and S. Jianwen, "Conversation specification: a new approach to design and analysis of e-service composition," in *Proceedings of the 12th International Conference on World Wide Web*, pp. 403–410, 2003.

[14] D. Roman, J. Scicluna, C. Feier, D. Fensel, A. Polleres, and J. de Bruijn, "Ontology-based Choreography and Orchestration of WSMO Services," WSMO Final Draft, 2005, http://www.wsmo.org/TR/d14/v0.4/.

[15] Z. Liang, H. Zou, F. Yang, and R. Lin, "A hybrid approach for the multi-constraint Web service selection problem in Web service composition," *Journal of Information & Computational Science*, vol. 9, no. 13, pp. 3771–3781, 2012.

[16] F. Tao, L. Zhang, K. Lu, and D. Zhao, "Research on manufacturing grid resource service optimal-selection and composition framework," *Enterprise Information Systems*, vol. 6, no. 2, pp. 237–264, 2012.

[17] G. Cai, "Web service composition on the environment level," in *Proceedings of the 6th International Conference on Semantics, Knowledge and Grid*, pp. 243–250, 2010.

[18] G. Cai, "Requirement driven service composition: an ontology-based approach," in *Proceedings of the 6th International Conference on Intelligent Information Processing*, pp. 16–25, 2010.