J Intell Manuf
DOI 10.1007/s10845-010-0410-7

A dynamic service composition schema for pervasive computing

Zhuzhong Qian · Zhenghui Wang · Tianyin Xu · Sanglu Lu

Received: 30 June 2009 / Accepted: 22 April 2010

Abstract Pervasive computing, the new computing paradigm aiming at providing services anywhere at anytime, poses great challenges on dynamic service composition. Existing service composition methods can hardly meet the requirements of dynamism and performance for pervasive computing. This paper proposes a Petri net based service model to formally describe the function of services and employs a parameter based service description to represent both semantic and syntactic of services. And services are pre-aggregated in a two-layered graph according to the input and output parameters of the service description. Furthermore, we design a novel service composition scheme to achieve the user requirement through a tree search algorithm. The theoretical analysis and comprehensive simulation experiments show that both service model and composition scheme are correct and efficient.

Keywords Pervasive computing · Service composition · Petri net

Introduction

Pervasive computing is a promising computing paradigm by which users can access resources they need anywhere at anytime. Pervasive computing also emphasizes on the technique invisible, i.e., users submit their requirements and get the results through access points, without knowing how to achieve those requirements. Usually, these requests are complicate and require several devices working together. To implement inter-operation among heterogenous devices, these resources are encapsulated as services, which can communicate with each other through standard protocols. This kind of software architecture is called service-oriented architectures (SOA). SOA proposes a promising way to dramatically increase services cooperation by utilizes utilizing available services to produce more complicated software using a loosely coupled architecture. SOA is suitable for designing and deploying applications in a pervasive computing environment (Kalasapur et al. 2007).

Generally, a single service is based on one single resource which is relatively simple, while the complicated requests are usually complex and can be achieved by composing the single services. The mechanism for combining two or more services together to form a "new" service is known as the *service composition*. It can decrease the cost of developing pervasive software components in terms of money, time and human resources by accomplishing new functions based on the existing services (Raman and Katz 2003a). Service composition is a good method to satisfy dynamic and complicated requests. But there are still two problems that need to be resolved, *i*) how to synthesize the composite services according to the requests and *ii*) how the composite service should be run in realtime.

WSDL (2006) is widely used to describe the static interface of service and the composite services, and can be defined with service composition languages (eg. BPEL, WSCI). However, WSDL cannot describe the semantics of a service, meaning the service composition process cannot be achieved automatically. The programmers have to select available services and write the specification of the composite services manually. Thus, this kind of composition mechanism could not be a realtime synthesis method. To automatically generate composite services that meet user requirements, several service models and related composition algorithms are proposed. OWL-S (Martin 2003), the session model

Z. Qian (✉) · Z. Wang · T. Xu · S. Lu
State Key Laboratory for Novel Software Technology, Nanjing University, 210093 Nanjing, People's Republic of China
e-mail: qzz@nju.edu.cn

Published online: 12 May 2010

(Bultan et al. 2003), and the state transition model (Fujii and Suda 2004) describe the service from different perspectives. The automatic composition approach, like the above models, focuses on the semantics of services and the verification of the composite service. However, these composition mechanisms seldom consider the performance of the algorithms, and response time is one of the most important Quality of Service (QoS) metrics in a pervasive environment, which deeply impacts the user experience. Furthermore, these service models define strict input and output conditions, which may also decrease the success ratio of service composition.

Different from the above mechanisms, this paper presents a deep study of the service composition problem for pervasive computing and proposes an effective mechanism for automatic service composition based on a two-layered graph. Our main contributions can be summarized as follows:

(1) We extend the Petri net based service model and define a novel parameter-based service description to represent a service in both semantic and syntactic layers. This service description framework decomposes the messages into parameters, by which complicated service processes can be easily represented. And the two-layered graph makes the composition more flexible.
(2) We design a graph-based service aggregation approach to organize service information in a pervasive environment, by which single services are "pre-organized" and their dependencies are "pre-explored". Furthermore, based on this approach, we introduce a dynamic service composition scheme for quick service composition at running time.
(3) We demonstrate the effectiveness of our scheme through both theoretical analysis and comprehensive simulations. The results show the efficiency of our scheme.

The rest of this paper is organized as follows. Section "Related work" overviews related work. Section "System overview" introduces the general system architecture of pervasive computing environments. Section "Service model" presents the parameter-based service model. The service aggregation mechanism and task resolution scheme are described in Sections "Service aggregation" and "Task resolution", respectively. We evaluate the system performance using simulations in Section "Performance evaluation". Sections "Discussion" and "Conclusion" discuss and conclude the paper.

Related work

Several works of research have addressed the service composition problem. Automatic service synthesis considers how to coordinate the service components to meet the functional requirement and generate the specification of the composite service. Based on the specification, QoS-driven service composition (selection) focuses on how to choose a set of service instances to effectively achieve the composite services.

Basically, there are 3 kinds of models to describe services (Hull and Su 2005): OWL-S (Martin 2003), the session model (Bultan et al. 2003) and the state transition model. OWL-S is a semantic description of services including 3 ontologies: *i*) the service profile represents a service, *ii*) the service model defines a process of the service, and *iii*) service grounding describes how to communicate with the service. Service composition mechanisms based on OWL-S are essentially dependent on IOPE (input, output, precondition, and effect) of services and utilizes workflow (Aalst and Hee 2002), Pi calculation (Parrow 2001) and intelligent planning (Russell and Norvig 1995) to achieve service composition. In the session model, a service is considered as a peer which is presented by a mealy model, where every service has a FIFO stack to receive messages. Additionally, there exists a sequence of the messages called the global status records all the messages among the services. In a service composition system's based on the session model, the sequence of messages is predetermined and can be defined by automata. Thus, the task of the service composition is to find a set of peers whose sequence of messages satisfies the desired automata. A typical state transition model the Roman model (Berardi et al. 2003), which uses an action based transition model to describe services. It employs proposal dynamic logic (PDL) (Kozen and Tiuryn 1990) to achieve service composition, where users provide a detailed composite process defined as an automaton. However, the Roman model, does not consider messages. It is worth noting that the Colombo model for service composition (Hull 2005), which is based on both the session model and the Roman model. Berardi et al. (2005) presents the details of the model and give PDL rules to automate service composition. Colombo uses an automata to describe service processes. However, this makes the process too complex and unreadable.

The service models described above are used in different cases. IOPE of OWL-S describes a service as a "black box", and the observer does not know the internal process of a service. The session model and state transition model show the detailed process of a service and define the interactions between service and service requester. The latter usually employs a graph structure to describe services and their requirements.

To effectively compose the services, some scholars propose graph based composition mechanisms. The PICO (Kalasapur et al. 2007) and dependency graph (Hashemain and Mavaddat 2005) approaches address the service composition problem by storing service behaviors in a graph structure. The nodes in the graph capture the inputs and outputs

of services, while its edges represent the input-output dependencies imposed by each service. Although these approaches convert the service composition problem into the shortest path problem, the input and output of the services are considered as one parameter. This approach is not practical and flexible because most of the services have several parameters and it cannot represent cooperation between more than 2 services. The method in Hashemain and Mavaddat (2006) expands the graph-based approach to handle services with multiple parameters. Each node in it captures the input and output parameters while four different kinds of edges represent the different relations. However, the service composition problem based on this irregular graph structure is much more complicated than the shortest path problem, whose complexity is exponential in the worst case.

Because several service instances in different nodes can offer the same function, QoS-driven service selection becomes critical in order to generate an optimal composite service in runtime. Zeng et al. (2003) considered multiple criteria and global constraints and then formulated the service selection as an optimization problem that can be solved by linear programming. Raman and Katz (2003b) proposed the LIAC algorithm to construct a service path. The basic idea is to construct a composite service overlay using k+1 copies of the original service overlay (for a k-tier composite service). The service selection is then equivalently converted to the shortest path problem in this composite overlay network, which can then be solved using the well-known Dijkstra's algorithm. With QoS-assured multimedia service provision in mind, (Gu et al. 2004) proposed SpiderNet, a quality-aware service composition middleware, that uses a heuristic algorithm to select the service instances in hop-by-hop manner, finding a service path ensuring multiple end-to-end QoS constraints.

To provide an effective automatic service composition mechanism for pervasive computing. This paper first presents a service model based on the Petri net, a widely used formal method (Kozen and Tiuryn 1990). A Petri net is suitable to describe a concurrent and asynchronous process. Compared with the state transition model, our model has much fewer states, and the service aggregation algorithm pre-defines the relationship among existing services, leading to much better performance at running time than any of the previous approaches. Thus, it is more suitable to apply this method in a pervasive environment to achieve the dynamic requests with high QoS assurance.

System overview

Generally, the system architecture of service-oriented pervasive computing environments is composed of 5 parts, as depicted in Fig. 1.

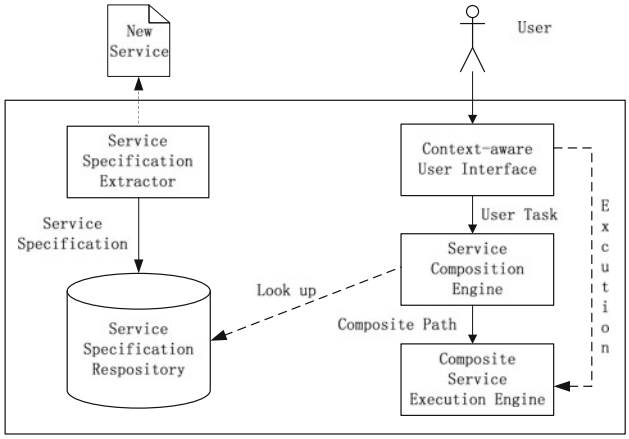

Fig. 1 The system architecture of pervasive computing environment

We briefly describe the key modules. (1) *Context-aware User Interface* (CUI) perceives user behavior based on voice, expression or action, and extracts the user requirements as the tasks. (2) *Service Specification Repository* (SSR) aggregates the necessary information of all available services in a specific data structure. (3) *Service Specification Extractor* automatically extracts and stores services information in the SSR. (4) *Service Composition Engine* (SCE) receives user tasks from CUI, extracts the expected behavior of the composite service and tries to find all possible composite paths in the SSR. (5) *Service Execution Engine* (SEE) selects one composite path from all the candidates considering the QoS requirements and network conditions. It also manages the order, correctness and efficiency of the execution.

This paper focus on the SSR and the SCE. We extend the Petri Net based service model to describe the atomic service and design an aggregation algorithm to represent the atomic services and their relations in the SSR. As the core of the SCE, the task resolution algorithm finds all possible composite paths according to user requests and information in the SSR.

Service model

In service oriented architecture, requesters do not need to know the internal behaviors of the service. What they require is how to invoke these services with correct messages. As the service provider, they also rely on these messages (e.g. internal status messages, incoming request messages) to trigger service behaviors in the appropriate order. Consequently, the composite service can be considered an interleaving sequence of messages and behaviors.

According to the above facts, we propose a Petri net based model to describe the internal process of services, which is essentially an implementation of the OWL-S process model.

In this section, we provide a detailed description of the service model and the parameter-based service description as an implementation of the formal model.

Petri net based service model

A message is a unique link between a service provider and a service requester. Generally, service grounding describes how to invoke a service behavior and the effect of the behavior, including the name, type and description of all the input and output parameters. In our framework, a service is invoked by a message sent by the requester. This message should be consistent with the WSDL description of the service, including all the parameters needed. That is, several parameters compose one message to trigger the service. And one service is considered as a process composed of a set of atomic actions. This kind of service is also called a *stateful* service, i.e. one service process includes several states and may interact with the requester during execution.

The following is a formal definition of the Petri net based process to describe the function of a service. $\mathcal{W}(P, T, F, M, I, S^0, G_f)$, where

- (P, T, F) is a directed net, called basic process of $\mathcal{W}$, satisfies $P \cap T = \phi$, $P \cup T \neq \phi$, $F \subseteq (P \times T) \cup (T \times P)$ and dom $(F) \cup cod(F) = P \cup T$;
- M is the set of parameters;
- $I : (P \rightarrow T) \cup (T \rightarrow P), anyI(p,t) \in [C(t)_M S \rightarrow C(p)_M S]_L$ and $I(p,t) = 0, iffI(p,t)$ is not in F;
- S^0: $P \rightarrow D_M S$ is the beginning label of $\mathcal{W}$ satisfies $\forall p \in P : S_0(p) \in C(p)_M S$, that is to say $S^0(p)$ is the multi-set of p;
- $G_f \subseteq R(M_0)$ is the set of terminal states.

P (place) denotes the state of the services; T (transition) denotes an action of the services. Relation F connects P and T, which shows the track of tokens in P flowing in the net. The basic process $\mathcal{W}$ represents the logic sequence of one service. Parameter M corresponds to the token flowing in the net. During the processing of the services, a set of tokens trigger one certain action (i.e. several parameters compose one request message and invoke the related service); and then, the actions consume messages and create messages as well (i.e. the input and the output of the services). Function I is the bridge of messages and actions, which shows how the actions affect the messages. Furthermore, we define $I-$ to denote that certain messages are consumed to trigger a certain action, and $I+$ to denote that messages are created by an action.

$*t = I - (p,t)$ are used to denote the set of parameters which is required to trigger action t. Messages in the process is the resource, and function $I-$ and $I+$ describe how the messages control the actions through consuming and creating messages. S^0 represents all the possible messages during the execution of the whole service process. During the execution of a composite service, the requester intercommunicates with the services according to the intermediate results. S^0 describes the potential interactions between the requester and the provider. According to the S^0, we also can use $S_i[\sigma\rangle S_j$ to represent that at label S_i following the action sequence σ, it reaches label S_j. G_f is the terminal set of W, and R(S^0) defines the set of all labels it can reach after an arbitrary action sequence. Different sets of parameters trigger different service actions. However, because of the asynchronization of messages, the order of the messages is non-deterministic. This leads to the non-determinacy of the action sequence. But based on the theory of Petri net (**?**), if the process reaches a certain status with a set of actions, then the process could reach the status through these actions in any order. That is, the order of the actions is not important for the analysis of the process of services.

This Petri net based service model decomposes messages between the service requester and the service provider into several tokens, where one token corresponds to one parameter in the messages. This model can easily describe the cooperation of services and all the Petri net tools can be used to guarantee the correctness of any of the processes.

Parameter-based service description

To implement the Petri net based service model and describe more context information such as price, QoS and confidence, we define a two-layered graph structure to represent a single service. $G_s = \{V_s, V_i, V_o, E, A_s, A_i, A_o\}$, where:

- V_s: A single element set representing the service itself.
- V_i: Each element in set V_i represents one input parameter.
- V_o: Single element set representing the output of a service.
- E: The directed edge set including all the edges from elements in V_i to V_s as well as V_s to V_o.
- A_s: Attributes of V_s including the semantic description (function) and the QoS assurance (e.g. reliability, delay, etc.).
- A_i: Each element of set A_i represents the attributes of the relevant element of V_i including the semantic description (Stype) and syntactic description (Type).
- A_o: Similar to A_i, it includes semantic (Stype) and syntactic description (Type) of V_o.

The key point of semantic description is how to represent the semantic information of a service. Since there are several candidate languages for semantic markup and each has its own advantages, we do not appoint any specific language. The requirement of the semantic description tool is that it

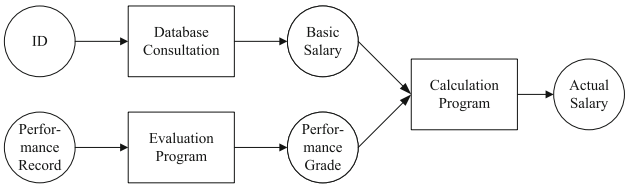

Fig. 2 The process of the salary system

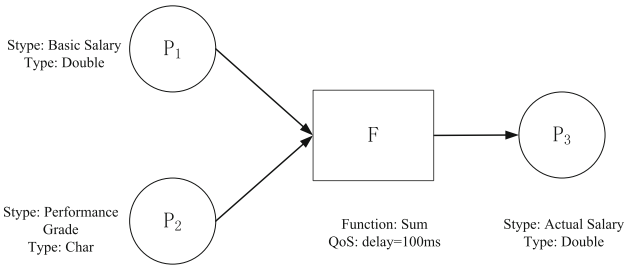

Fig. 3 The service model of the calculation program service

need describe a set of classes, properties and service functions. Thus, not only is the OWL-S (Martin 2003) a suitable candidate by this standard, but so are semantic models of service, such as CoSMoS (Fujii and Suda 2004), which can be integrated into our parameter-based service model. The syntactic description is relatively simple. The XML are enough to support this.

To explain the service model clearly, consider a salary system of a company using service-oriented architecture. In this system, three services are deployed: a database consultation service (S_1), an evaluation program service (S_2) and a calculation program service (S_3). Figure 2 shows the process.

Figure 3 represents the service model of the calculation program (S_3) service. It is trivial that $V_i = \{P_1, P_2\}$, $V_s = \{F\}$, $V_o = \{P_3\}$, $E = \{(P_1, F), (P_2, F), (F, P_3)\}$. The attributes are written next to the corresponding nodes.

The semantic unit in our service model is a parameter. The function of service is also introduced in this model. This model provides comprehensive information of service at both the functional and the instance level. This will aid in aggregating services in a graph structure and searching the corresponding composite path as shown in the following sections.

Service aggregation

Typically in a pervasive environment, every new service registers on a certain node, where all the service information is stored (Chakraborty et al. 2004; Guttman 1999. The information can also be stored in a distributed way such as in Waldo (1999). For simplicity, we assume the meta-data associated with the services are stored in the SSR and our approach can be adopted in a distributed fashion easily.

In this section, we focus on creating the aggregation graph G_a to store all the behaviors and relations of services from the registered service graph G_s. The aggregation graph G_a is a two-layered graph. One layer is G_{sem} representing the functional relations of services and parameters. The other is G_{syn}, representing the service instances and parameter types.

The aggregation process creates a structure similar to G_s in both the semantic layer and the syntactic layer, avoiding duplicate node with the same description. The dependency of service is determined by the shared parameters. The connection between two layers also helps us determine the instances for specific functions. The details of the algorithm are described in Algorithm 1.

Algorithm 1 Service Aggregation Algorithm

```
1: Initial G_a = null
2: for each G_s do
3:   SemNode Vs_sem = findSemNode(Vs.Stype);
4:   SemNode Vo_sem = findSemNode(Vo.Stype);
5:   SynNode Vs_syn = findSynNode(Vs.Stype, Vs.Type);
6:   SynNode Vo_syn = findSynNode(Vo.Stype, Vo.Type);
7:   for each input parameter Vi_k of G_s do
8:     SemNode Visem = findSemNode(Vi_k.Stype)
9:     SynNode Visyn = findSynNode(Vi_k.Stype, Vi_k.Type)
10:    addEdge(Vi_sem, Vs_sem);
11:    addEdge(Vi_syn, Vs_syn);
12:    addDoubleEdge(Vi_sem, Vi_syn);
13:  end for
14:  addEdge(Vs_sem, Vo_sem);
15:  addEdge(Vs_syn, Vo_syn);
16:  addDoubleEdge(Vs_sem, Vs_syn);
17:  addDoubleEdge(Vo_sem, Vo_syn);
18: end for
```

The $SemNode$ and $SynNode$ refer to the node in G_{sem} and G_{syn}, respectively. The function $findSemNode(v)$ in line 3 returns the node in G_{sem} whose semantic description is v. The function $findSynNode(v, w)$ in line 5 returns the node with semantic description v and syntactic description w. If $findSemNode()$ or $findSynNode()$ does not find the required node, a new corresponding node will be added in and returned. Function $addEdge(v, w)$ adds a directed edge $v \rightarrow w$ and $addDoubleEdge(v, w)$ adds both $v \rightarrow w$ and $w \rightarrow v$ in the graph.

The function $findSemNode()$ travels all the nodes in G_{sem}. So its complexity is $O(m)$, where m is the scale of G_{sem}. Since edges exist between corresponding semantic nodes and syntactic nodes, $findSynNode()$ does not need to visit each node in G_{syn} but visits the linked nodes of semantic nodes. So its complexity is also $O(m)$. In sum, the complexity of adding a new service is $O(m)$.

When services are aggregated in G_a, the following statements hold true:

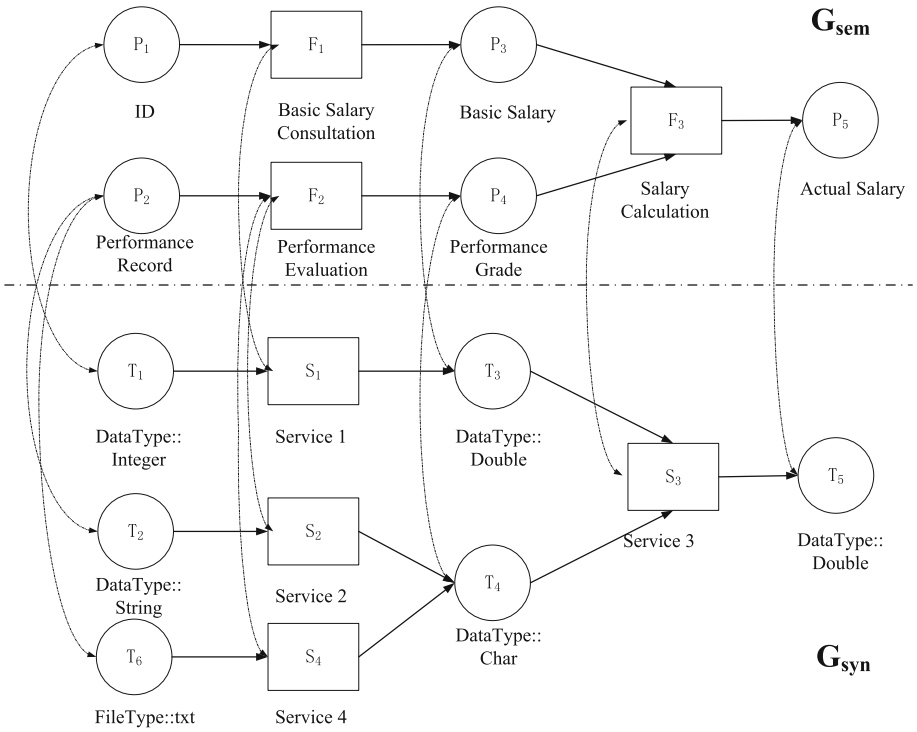

Fig. 4 The aggregation graph of all the services in the salary system

1. The predecessor of a parameter node is a service node, while the predecessor of the service node is the parameter node in the same layer.
2. No two nodes in G_{sem} represent the same semantic information.
3. No two nodes in G_{syn} both represent the same syntactic information and point to the same node in G_{sem}.
4. Each node in G_{sem} has at least one corresponding node in G_{syn} while each node in G_{syn} has and only has one corresponding node in G_{sem}.

The aggregation graph of all the services in the salary system above is shown in Fig. 4. The text below the node marks its semantic or syntactic information. S_4 is another evaluation program accomplishing same function as S_2 but has input parameters with different data type.

Task resolution

Task representation

A task refers to the function that the composite service needs to accomplish. SpiderNet Gu et al. (2004) requires users to input the function graph, which makes the system difficult to use. SeGSeC Fujii and Suda (2004) assumes the task is described in a natural language sentence. Although it has a natural UI, it makes a complicated request difficult to deal with due to the vague expression of natural language and little useful parsed information to guide the service composition.

In another perspective, if the input and output of a function are defined accurately, the function is determined. In this paper, the model used to represent the task includes three parts: (1) the input parameters, (2) the output parameters and (3) the QoS requirement. The input and output parameters are the same as in the service model including both the semantic and syntactic descriptions of parameters. In other words, a user should describe the function of a service by illustrating what one can supply and what is the expected output. In fact, users do not need to provide the complete semantic and syntactic descriptions of the parameters. Most of them are perceived by the context-aware interface.

The QoS requirements restricts the available services and provides the standard with which to choose a composition path among candidates. This is a complex problem and will not be discussed here for it is concerned with the design of the composition execution engine, and outside the scope of this paper.

Task resolution scheme

When a task is submitted to the SCE, a direct match is checked first. If there is a direct match between the component service and task, this task can be resolved directly. On the other hand, if such a direct match does not exist, we need to combine several components to accomplish the task.

Two service components S_1 and S_2 can be directly combined if

(1) Output of S_1 can be consumed by S_2;
(2) The syntactic description of the output of S_1 can be accepted by S_2.

In the aggregation graph G_a, these conditions can be expressed as follows:

(1) A path exists between $semNode\ S_1$ and $semNode\ S_2$ in G_{sem}, on which exits only one node p_{sem}.
(2) A path exists between $synNode\ S_1$ and $synNode\ S_2$ in G_{syn}, on which exists only one node p_{syn}.
(3) An edge exists between p_{sem} and p_{syn}.

The input of the task resolution algorithm includes two kinds of composite service parameter set. One is the sets of input parameters, denoted as $(p1_{in}^{sem}, p2_{in}^{sem}, \ldots, pn_{in}^{sem})$ for semantic and $(p1_{in}^{syn}, p2_{in}^{syn}, \ldots, pn_{in}^{syn})$ for syntactic. The other is the set of output parameters, denoted as $(p_{out}^{sem}, p_{out}^{syn})$.

The output of the task resolution is all the possible composite paths. A composite path is a sub-graph of G_a. Actually, the sub-graph should be a tree whose leaves are pairs of the input parameters of the composite service and the root is the output parameter. The task resolution algorithm aims at finding such a tree from the given leaves and root.

However, a service is executable only when all its input parameters are prepared. And a parameter can be obtained only if all the service nodes that can produce it are executable. So any node in the composite path has the following properties:

(1) If a service node is in the composite path, all its parameter predecessors are in the composite path.
(2) If a parameter node is in the composite path, its specific service predecessor is in the composite path.

These are the necessary conditions. Based on this, we provide the following standard to determine whether a node is in the composite path or nor.

Assume that the composite path exits. A node v of G_a is in the composite path if and only if it can satisfy one of the following conditions.

(a) Node v is the given node by user;
(b) Node v is a node satisfying the above condition (1) or (2), and a specific successor of v is in the composite path.

The necessity of these conditions is trivial. Obviously, the node satisfying (a) is in the composite path. Condition (b) means v could be in the composite path and leads to the final parameter of composite service.

Since the conditions are defined recursively, a natural approach to find all nodes in the composite path is traveling G_a in a recursive way. A modified Depth First Search (DFS) is used. A loop may be detected in the travel process. The nodes on it are not in the composite path. Since G_a is a two layered graph, the composite path should be searched in both G_{sem} and G_{syn}. The composite path found in G_{sem} is the functional composition and the path in G_{syn} refers to the corresponding service instance. If both of them exist, the task can be resolved. The task resolution algorithm is shown in Algorithm 2.

Algorithm 2 Task Resolution Algorithm

```
1: Initial all nodes with white color;
2: set the nodes representing input/output parameters of composite services as red color;
3: SemNode outPara_sem = SearchSemNode(p_out^sem);
4: SynNode outPara_syn = SearchSynNode(p_out^syn);
5: if outPara_sem = null or outPara_syn = null then fail() end if;
6: MarkSemNode(outPara_sem);
7: if GetColor(outPara_sem) ≠ red then fail() end if;
8: MarkSynNode(outPara_syn);
9: if GetColor(outPara_syn) ≠ red then fail() end if;
10: TreeNode root = CreateAndOrTree(outPara_sem)
11: if root = null then fail() end if;
12: return root;
```

The function $SearchSemNode(v)$ in line 3 returns the node whose semantic description is v or $null$ if no node is matched.The function $SearchSynNode(w)$ in line 4 is similar. The $MarkSemNode$ (see Algorithm 3) in line 6 is the procedure judging whether a node is in the composite path recursively. Its initial value is the node with semantic description p_{out}^{sem} and all the nodes in the composite path will be marked red finally.

The function $MarkSynNode()$ is similar with $MarkSemNode()$ except that node v is set gray in line 2 if node v is white and node w is red, where v is the node in G_{syn} and w is its predecessor in G_{sem}.

The task resolution algorithm consists of two procedures: marking nodes and building *and-or tree*. The marking procedure finds the node in the composition path. There are four colors representing different states of nodes: *white* nodes are unmarked nodes; *gray* nodes are being marked; *black* nodes have been marked and not in the composition path; and *red* nodes are marked and in the composition path.

In the marking procedure, all the nodes are initially white and the nodes corresponding to the user input parameters are red. The $MarkSemNode()$ and $MarkSynNode()$ mark the node in G_{sem} and G_{syn} according to the standard above using DFS. The red path in G_{sem} marks the function composition

Algorithm 3 Mark Semantic Node Algorithm

```
1: if getColor(v) = red or getColor(v) = black then return;
   end if
2: if getColor(v) = white then setColor(v, gray);
3: else return;
4: end if;
5: if v.nodeType= parameter then
6:   for each predecessor w of v in G_sem do
7:     MarkSemNode(w);
8:     if getColor(w) = red then setColor(w, red); end if
9:   end for
10:  if getColor(v) ≠ red then setColor(w, black); end if
11: else
12:  for each predecessor w of v in G_sem do
13:    MarkSemNode(w);
14:    if getColor(v) ≠ red then setColor(v, black); end if
15:  end for
16:  if getColor(v) ≠ black then setColor(v, red); end if
17: end if
```

and the corresponding red path in G_{syn} provides service instances for each function component.

$CreateAndOrTree()$, the procedure of building and-or tree, is based on the function composition. The tree structure is the sub-graph of G_{sem}. The syntactic description of the node in G_{syn} is attached to the corresponding tree node as attribute.

In this structure, each and-node represents a kind of service whose children are its input parameters and its attribute contains all the available services to accomplish this service function. Each or-node means a kind of parameter whose father and children are services. It is one input parameter of its father and it can be obtained by any child. Its real type is stored as its attribute. In fact, we classify the services by their functions and store this classification in the and-node. In this way, it is easy to replace a service with its peer.

Theoretical analysis

Considering the semantic unit (i.e. ontology) with size m and syntactic type with size k, if there are N services, G_{sem} contains m vertexes and $m^2/4$ edges at most. Because the edge only exits between different kinds of nodes, if there are m_1 parameter nodes and m_2 service nodes, the maximum edge is $m_1 \times m_2 \leq (m_1+m_2)^2/4 = m^2/4$. Similarly, G_{syn} contains $k \times m$ vertices and $k^2 \times m^2/4$ edges at most. The maximum number of edges between the nodes in G_{sem} and G_{syn} is $k \times m$. Generally, $m \ll N$ and k can be viewed as a constant for quite limited syntactic types.

Since $MarkSemNode()$ only travels G_{sem} once, the complexity is $O(m + m^2/4)$. Similarly, the complexity of $MarkSynNode()$ is $O(km + k^2m^2/4)$.The complexity of building the and-or tree is $O(m + m^2/4 + km)$. So the complexity of task resolution is $O(m^2)$. This means the complexity of task resolution is not varied with the service number, but rather is determined by the static ontology number, which makes it quite effective especially when a large number of redundant services exist in the pervasive computing environment.

Additional restriction

The task resolution algorithm requires the user to provide the semantic and syntactic descriptions of the input and output parameters of the composite service. Actually, the user may not be able to provide complete information. Sometimes they want to interfere with the automatic procedure to produce the expected result. All these above can be viewed as the additional restrictions on the task resolution algorithm. The algorithm can be varied in the following ways.

i) Incomplete user inputs. In this case, the output parameters are complete and input parameters are incomplete. The user only requires the service to accomplish a certain semantic function and has no syntactic restriction on the output. For the incomplete inputs, the task resolution algorithm is adapted to call $MarkSynNode()$ for each corresponding node of the output parameter in G_{syn}.
ii) Additional service and data restriction. For some users, they expect the composition to take some atomic services or produce certain interim data. They can give some functional guidance to the composition process sometimes. They may even expect to specify the entire process of the service composition.

A task tree is introduced to satisfy this need. A task tree is also an and-or tree as mentioned before. The only difference between the two is the task tree does not need to describe the entire composition process. It only needs to point out the existing service or data and the order of them.

Thus, a task tree could be the input of the task resolution algorithm, as well as the input and output parameter specifications. After obtaining the and-or tree from the execution of the task resolution algorithm, any sub-tree that does not satisfy the existence or order of required services and data represented by task tree should be cut off.

Performance evaluation

In order to empirically test and evaluate our approach, we implemented a simulator in Java using J2SE 1.6.2. The numbers of functional semantics and parameter semantics, services and parameter types are initialized by the user. Services are randomly constructed at runtime according to the above variables. The tasks, the user supplied parameters and required parameters, are also created randomly.

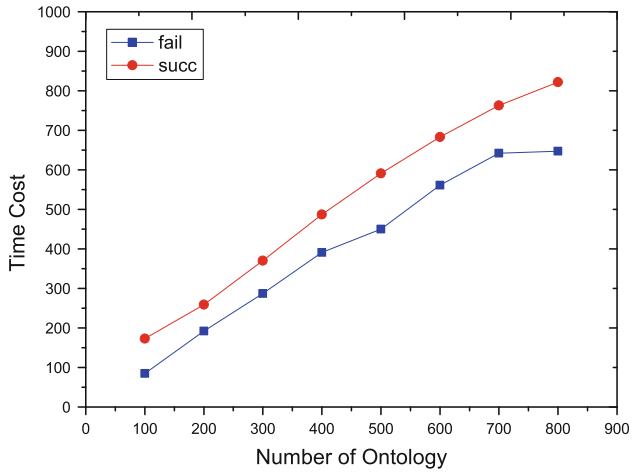

Fig. 5 Time cost for different ontology scale

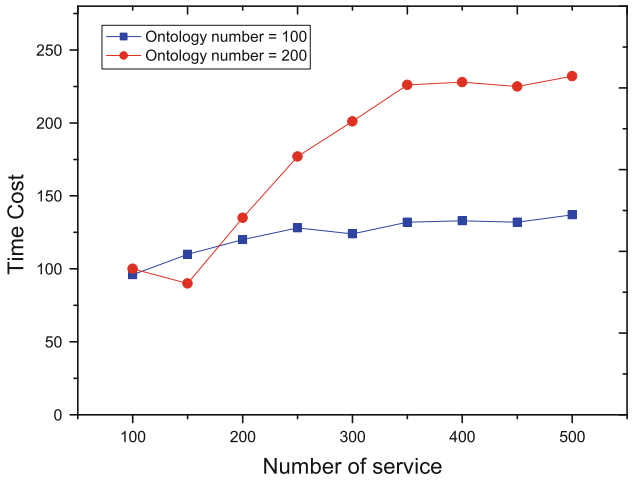

Fig. 6 Time cost for different service scale

Based on these services and the user tasks, the simulator aggregates the services and resolves the tasks using the algorithms proposed in the above sections. To measure the performance of the task resolution algorithm, we consider visiting a node as the basic operation and record the number of the times nodes are visited during the running time. To be more accurate, each number of basic operation is counted forty times with the same scale and the value, and the result shown in the figure is the average of these forty results; The complexity of successful and failed task resolution are counted.

In Fig. 5, the service number is fixed and is set as one thousand while the ontology number increases. As the number of the ontologies increase from 100 to 900, the complexities of both successful and failed cases rise linearly. The number of ontologies corresponds to the number of different parameters. More parameters for the fixed number of services means that more services may share the same parameters. Thus, there are much more relationships among the services and the aggregation graph is more complicated. This is why it costs more time to get the results.

In Fig. 6, the ontology number is stable while the service number increases. The ontology numbers of the two curves are 100 and 200, respectively. Obviously, the first curve is approximately horizontal. For the second one, the first section rises as the node number increases with the services and even as the service scale continues to rise the remaining section sees little fluctuation. Thus, if the number of ontologies is small, as the number of services increases, the time cost increase slightly. The number of ontologies implies the number of the types of parameters. If the number of ontologies is small, few services share the same ontologies with other services. Consequently, even if the service number is large, the scale of the connected branches of the aggregation graph is still not big, which reduces the time cost of the algorithm.

Observing these two figures, we reach the following conclusion: (1) the average complexity of task resolution is approximately linear; (2) the time cost does not increase as the service scale expands. Since the ontology number is quite limited and stable, our approach is proved quite efficient and well controllable.

Discussion

In this section, we mainly discuss two issues on aggregation graph since it is the foundation of our research. One is the necessity of our two-layered aggregation graph; the other is how to expand the semantic layer of the aggregation graph to represent other relations between concepts.

i) The services are represented in a two-layered graph in our approach. Although it seems enough to accomplish the required service composition only based on the semantic layer, the syntactic layer is indispensable if we consider the problem comprehensively. Finally, the composite paths are sent to the composite service execution engine. If only the semantic layer exists, the service instances can not be attached to them as attributes and the composite service execution has to find the matched services as the composite paths indicate among all registered services. This would undoubtedly increase the complexity of the whole process for the cost of this component varies with service scale.

ii) Although representing the semantic information and relations is not our concern, we still hope to better support this as it will facilitate our work and increase the success rate of task resolution. The aggregation graph can be easily expanded to represent some basic relations, such as the supertype/subtype and composition/decomposition, between the semantic concepts. The direct edge between parameter nodes can represent the fact that the

predecessor node is a subtype of the latter. For instance, the edge from *capital* node to *city* node means *capital* is a special *city*. To express the composition/decomposition relation between concepts, we can introduce a virtual service referring to the constructor as the service instance. All the component concepts are the input parameters and the composite concept is the output. Take the *address* concept as an example. The *address* concept includes the concept *city*, *street* and *post code*. A virtual service node called *adrConstuctor* is introduced whose inputs are *city*, *street* and *post code* and output is *address*. Fortunately, these two kinds of new relations do not lead to any changes in our task resolution algorithm.

Conclusion

In this paper, we define a Petri net based service model and implement it with a parameter-based service description. We organize all the service information in the repository as an aggregation graph and explain how this two-layered graph captures the behavior of services in terms of input and output parameters and their relations. Facilitated by the aggregation graph, we convert the service composition problem into the problem of finding a sub-tree within the aggregation this graph and provide an approach searching all the possible composite paths. To prove its efficiency, we explain the upper bound of its complexity and measure the average performance by simulation experiments. Both results show that our scheme has the ability to accomplish user requests in a short response time.

Acknowledgments This work is partially supported by the National Natural Science Foundation of China under Grant No. 90718031, 60721002; the National Basic Research Program of China (973) under Grant No. 2009CB320705; Jiangsu Natural Science Foundation under Grant No. BK2008264. A special thank to Lee Dan Bowman for helping us fixing the grammar mistakes.

References

Aalst, W., & Hee, K. (2002). *bworkflow management: Models, methods, and systems*. Cambridge: MIT Press.

Berardi, D., & Calvanese, D., et al. (2003). Automatic composition of E-services that export their behavior. In *Proceedings of International Conference of Service Oriented Computering (ICSOC)*, pp. 43–58.

Berardi, D., Calvanese, D., Giacomo, D., Hull, R., & Mecella, M. (2005). Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pp. 613–624. Trondheim, Norway.

Bultan, T., Fu, X., Hull, R., & Su, J. W. (2003). Conversation specification: A new approach to design and analysis of E-service composition. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, pp. 403–410. Budapest, Hungary.

Chakraborty, D., Yesha, Y., & Joshi, A. (2004). A distributed service composition protocol for pervasive environments. In *Proceedings of Wireless Communications and Networking Conference (WCNC)*, pp. 2575–2580.

Extensible Markup Language (2008). 1.0, www.w3.org/TR/REC-xml/.

Fujii, K., & Suda, T. (2004). Dynamic Service Composition Using Semantic Information," *Proceedings. of 2nd International Conference on Service Oriented Computing (ICSOC)*, pp. 39–48.

Guttman, E. (1999). Service location protocol: Automatic service discovery of IP network services. *IEEE Internet Computing, 3*, 71–80.

Gu, X., Nahrstedt, K., & Yu, B. (2004). SpiderNet: An integrated peer-to-peer service composition framework. In *Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing*, pp. 110–119.

Hashemain, S. V., & Mavaddat, F. (2005). A graph-based approach to web services composition. In *Proceedings of the 2005 IEEE/IPSJ International Symposium on Applications and the Internet (SAINT)*, pp. 183–189.

Hashemain, S. V., & Mavaddat, F. (2006). A graph-based framework for composition of stateless web service. In *Proceedings of 4th European Conference on Web Services (ECOWS)*, pp. 75–86.

Hull, R. (2005). Towards a unified model for web services composition. In *Proceedings of Advances in Computer Science—ASIAN2005*, pp. 1–10. Kunming, China.

Hull, R., & Su, J. W. (2005). Tools for composite web services: A short overview. *SIGMOD Record, 34*(2), 86–95.

Kalasapur, S., Kumar, M., & Shirazi, B. A. (2007, July). Dynamic service composition in pervasive computing. In *Proceedings of the IEEE Transactions on Parallel and Distributed Systems (TPDS)*, pp. 907–918.

Kozen, D., & Tiuryn, J. (1990). *Logics of programs. Handbook of theoretical computer science—formal models and semantics*. Amsterdam: ESP.

Martin, D. (2003, November). OWL-S: Semantic Markup for Web Services. http://www.daml.org/services/owls/1.0/owl-s.html.

Parrow, J. (2001). An introduction to the pi-calculus. *Handbook of Process Algebra*, pp. 479–543.

Raman, B., & Katz, R. H. (2003). An architecture for highly available wide-area service composition. *Computer Communications, 26*(15), 1727–1740.

Raman, B., & Katz, R. (2003). Load balancing and stability issues inalgorithms for service composition. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM03)*, pp. 1477–1487. USA.

Russell, S., & Norvig, P. (1995). *Artificial intelligence: A modern approach*. Englewood Cliffs: Prentice Hall.

Waldo, J. (1999). The jini architecture for network-centric computing. *Communications of the ACM, 42*(7), 76–82.

WWW Consortium, Web Services Description Language (WSDL) Version 2.0, W3C Working Draft, (2006, January) http://www.w3.org/TR/wsdl20/.

Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., & Sheng, Q. (2003). Quality driven web services composition. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, pp. 411–421. Budapest, Hungary.