A Parameter-based Scheme for Service Composition in Pervasive Computing Environment

Zhenghui Wang[†], Tianyin Xu[†], Zhuzhong Qian[†‡], Sanglu Lu[†]
[†]State Key Lab. for Novel Software and Technology
[†]Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China
[‡]School of Computer Science and Engineering, The University of Aizu, Aizu-Wakamastu, Japan

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 dynamic characteristic and heterogeneity in pervasive computing environment. In this paper, we propose a parameter-based service model to accurately describe pervasive services. Based on the model, pervasive services are aggregated in a two-layer graph according to both semantic and syntactic information of the input and output parameters. Moreover, we design a novel service composition scheme to accomplish the user task while satisfy the QoS requirements. Both theoretical analysis and simulation experiments show that this service composition mechanism is effective in pervasive environment.

1. Introduction

Pervasive computing is a promising computing paradigm by which users can access resources they need anywhere at anytime in the pervasive environment. Service-oriented architectures are suitable for designing and deploying pervasive computing environment in which each resource is encapsulated as a pervasive service. Typically, a single service is relative simple while the user tasks are more complex and variable. So it is necessary to combine these services. The mechanism for combining two or more services to form a new service is known as the *service composition*. The combination can be determined according to the predicted user tasks when the system is deployed. However, this static design limits the possible usage of resources and does not adapt to the fluctuation of pervasive computing environment as well as user requirements.

Dynamic service composition, aiming at combining the available pervasive services at runtime to facilitate user tasks, can meet the requirements of the dynamic characteristic and heterogeneity in pervasive computing environment. Moreover, it can also decrease the cost of developing pervasive software components in terms of money, time and human resources by accomplishing new functions based on the existing pervasive services. Thus, the dynamic service composition in pervasive environment attracts great research interests in recent years.

A number of pioneer work to service composition problem has been proposed [5-10]. Some of them (e.g. [5][10]) require users to provide the function graphs, service templates or logic formulas to describe their tasks, which is an obstacle for users to use such systems. Some approaches (e.g. [8][6]) handle this problem by collecting semantic and syntactic information about services in a graph and automatically finding the composite path according to users' requirements. However, the limitation is that they only allow services to have single input parameter which makes these approaches impractical. Other work that models the multiple parameters are too complicated for pervasive environment.

Since most existing service composition approaches are not suitable when being used in pervasive computing environment, we propose a novel parameter-based service composition scheme which is an effective dynamic composition scheme for pervasive environments.

Our main contributions can be summarized as follows:

(1) We propose a novel parameter-based service model, which can accurately describe both semantic and syntactic information of services with multiple parameters.

(2) We design a graph-based service aggregation approach to organize service information in the pervasive environment. Based on this approach, a dynamic service composition scheme is introduced.

(3) We demonstrate the effectiveness of our scheme through both theoretical analysis and simulation experiments. The results show that our scheme is effective in pervasive environment.

978-0-7695-3575-3/09 $25.00 © 2009 IEEE
DOI 10.1109/CISIS.2009.50

2. Related Work

Similar versions of composition problem have been studied in a great number of papers. They are varied with the information captured from the network, the service model, assumed form of user tasks and the level of abstraction, concreteness and complexity of the solutions.

KarmaSIM [9] can automatically generate a *Petri-Net* according to the DAML-S[4] description of services. Based on Petri-Net, this simulator can perform desirable analysis and service composition. However, the objective is to simulate, compose and verify the composition of services. It is complicated to only achieve the goal of service composition by simulating the whole network.

SpiderNet [5] views the service composition as an extension of service discovery. It accepts a function graph as input and the objective is to find corresponding services in the network. But the user may not be able to determine the function graph for some complicated problem. Even if the function graph can be determined, it could be too special to match any available service.

PICO [8] and dependency graph approach in [6] address the above problem based on storing the service behaviors in a graph structure. The nodes in the graph capture the entire inputs and outputs of services, while its edges represent the input-output dependencies imposed by each service. Although these approaches convert service composition to the shortest path problem, they limit the input and output to one parameter, which renders them impractical.

The method in [7] expands the graph-based approach to handle services with multiple parameters. 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.

However, little service composition approach is designed specially for pervasive computing environment.

3. System Architecture

Generally, the system architecture of pervasive environments is composed of five parts, as depicted in Fig. 1.

We briefly describe the key modules. (1) *Context-aware User Interface* (CUI) perceives the user behavior on voice, expression and action and then extract the users' requirements as the user tasks. (2) *Service Specification Repository* (SSR) aggregates the necessary information of all available service in a specific data structure. (3) *Service Specification Extractor* automatically extracts and store information of services 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*

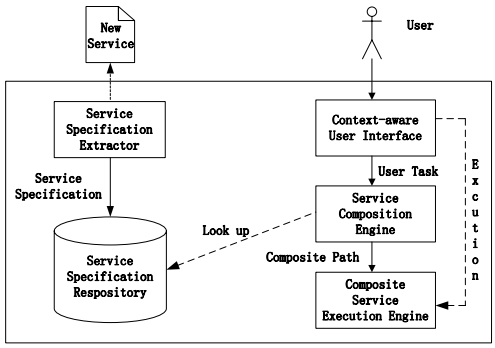

Figure 1. The system architecture of pervasive computing environment.

selects one composite path from all the candidates considering the QoS requirements and network conditions. It manages the order, correctness and efficiency of the execution.

In this paper, we focus on the SSR and SCE level. A new service model is proposed to describe the atom service and an aggregation algorithm is designed to represent the atom services and their relations in SSR. As the core of the SCE, the task resolution algorithm finds all possible composite paths according to use tasks and SSR.

4. Parameter-based Service Model

The main difference between a pervasive service and a pervasive software application is the former has standard specifications to describe its behavior. This standard specification is the *service model*. The WSDL [3] is a widely used service model in industry. Since it lacks the sematic information about services, it cannot support the dynamic service composition. The service models defined in [8][6] capture the functions of service by the semantic information of the input interfaces and output interfaces. However, if the input of one service depends on two or more services, the interface integration on semantic level is inevitable, which leads to misunderstanding.

In our approach, each pervasive service is described as a simple graph $G_s = \{V_s, V_i, V_o, E, A_s, A_i, A_o\}$, where V_s is a single element set representing the service itself; Each element in set V_i represents one input parameter; V_o is the single element set representing the output of a service; E represents 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 is the attributes of Vs including the semantic description (function) and the QoS assurance such as reliability, delay and so forth; 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 represents the attributes of

V_o including the semantic description and data type.

The key point of semantic description is how to represent the semantic information of pervasive service. Since there are a lot of work in semantic markup language and each work has its own advantages, we do not appoint any specific language. The request of the semantic description tool is that it should describe a set of classes, properties and service functions. Thus, not only the OWL-S [1] and DAML-S [4] are suitable candidates by this standard, but also some existing semantic models of service can be integrated into our parameter-based service model. The syntactic description is relatively simple. The XML [2] is enough to support this.

To explain the service model clearly, consider a company's salary system using service-oriented architecture. In this system, three services are deployed: a database consultation service (S1), an evaluation program service (S2) and a calculation program service (S3). Fig. 2 shows the work-flow.

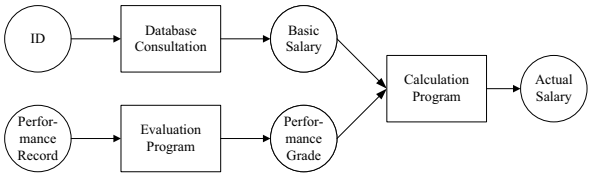

Figure 2. The work-flow of the salary system.

Fig.3 represents the service model of the calculation program (S3) service. It is trivial that $V_i = \{P1, P2\}$, $V_s = \{F\}$, $V_o = \{P3\}$, $E = \{(P1, F), (P2, F), (F, P3)\}$. The attributes are written near the corresponding nodes.

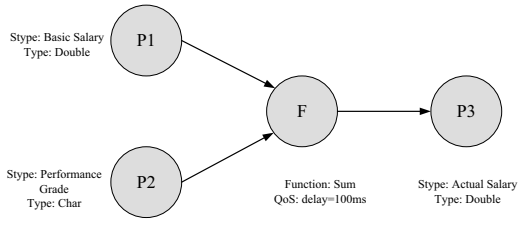

Figure 3. The service model of the calculation program service.

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

5. Service Aggregation

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-layer 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. For simplicity, we assume all the metadata of the services are stored in centralized SSR and our approach can be adopted in a distributed fashion easily.

During the aggregation process, a similar structure is created as G_s in both semantic layer and syntactic layer avoiding duplicate node with same description. The dependency of service is determined by the shared parameters. The connection between two layers also help us find the instances for certain function. 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 Vi_sem = findSemNode(Vi_k.Stype);
9:         SynNode Vi_syn
           = 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, $findSynNode()$ do not need to visit each nodes in G_{syn} but visit the linked nodes of semantic nodes. So its complexity is also $O(m)$. In sum, the complexity of adding a new service in G_a is $O(m)$.

When services are aggregated in G_a, the following statements hold true: (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 is pointed by 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}.

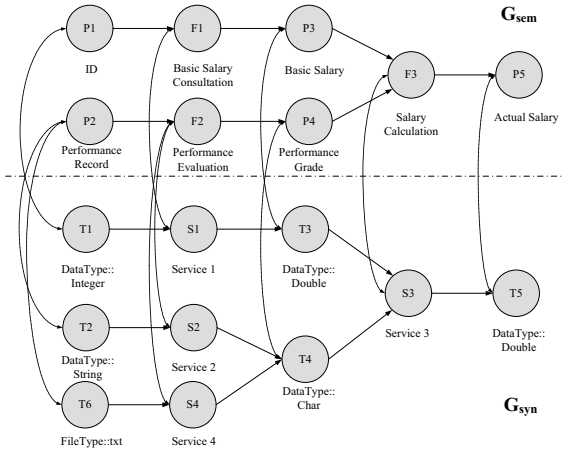

Figure 4. The aggregation graph of all the services in the salary system.

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. S4 is another evaluation program accomplishing same function as S2 but the data type of its input parameter is different.

6. Task Resolution

6.1. Task Representation

A task refers to a function that needs to accomplish by a pervasive service. 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) input parameters, (2) output parameter and (3) QoS requirements. The input and output parameters are the same as in the service model including both the semantic and syntactic descriptions of parameters. In another word, a user should describe the function of a service by illustrating what one can supply and expects to obtain as an 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 concern on the service instances. It restricts the available services and provides the standard to choose composition path among candidates. This will not be discussed in this paper for it is concerned by the design of the composition execution engine.

6.2. Task Resolution Scheme

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

Two service components S1 and S2 can be directly combined if: (1) Output of S1 can be consumed by S2; (2) The syntactic description of S1's output can be accepted by S2.

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

(1) A path exists between $semNode$ S1 and $semNode$ S2 in G_{sem}, on which only one node p_{sem} exits.

(2) A path exists between $synNode$ S1 and $synNode$ S2 in G_{syn}, on which only one node p_{syn} exists.

(3) An edge exists between p_{sem} and p_{syn}.

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

The output of 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 the input parameters of composite service and 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 if any service that can produce it is 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 certain service predecessor is in the composite path.

These are the necessary conditions. Based on this, we give the following standard judging whether a node is in the composite path.

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 certain 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. Loop is possibly detected in the travel process. The nodes on it are not in the composite path. Since G_a is a two layer 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 that in G_{syn} refers to the corresponding service instance. If both of them exists, 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** $GetColor(outPara_{sem}) \neq red$ **then** $fail()$; **end if**
8: $MarkSynNode(outPara_{syn})$;
9: **if** $GetColor(outPara_{syn}) \neq red$ **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.

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** $getColor(v) \neq red$ **then** $setColor(w, black)$; **end if**
11: **else**
12: **for** each predecessor w of v in G_{sem} **do**
13: $MarkSemNode(w)$;
14: **if** $getColor(v) \neq red$ **then** $setColor(v, black)$; **end if**
15: **end for**
16: **if** $getColor(v) \neq black$ **then** $setColor(v, red)$; **end if**
17: **end if**

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 corresponding one 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; *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 white initially 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 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 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 by its peer.

6.3. 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 $k \times m$ vertexes and $k^2 \times m^2/4$ edges at most. The maximum edge 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 exist.

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 is determined by the static ontology number, which makes it quite effective especially when a large number of redundant services exist in the pervasive environment.

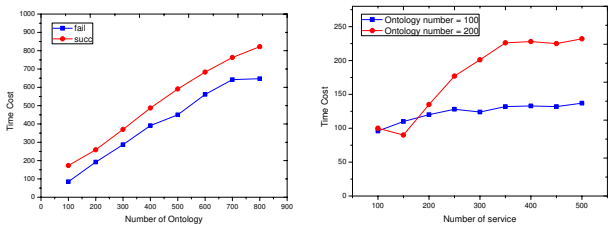

Figure 5. Time cost for (a) different ontology scale and (b) service scale.

7. Performance Evaluation

In order to empirically 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 these variables above. The user tasks, the user supplied parameters and required parameters, are also created at random.

Based on these services and the user task, the simulator aggregates the services and resolves the tasks using the previous algorithms. To measure the performance of task resolution algorithm, we view the node visit as the basic operation and count the times. To be more accurate, each time of basic operation is countered forty times with the same scale and the value shown in the figure is their average; The complexity of successful and failed task resolution are countered respectively.

In Fig. 5(a), the service number is maintained at 1000 with the rise of the ontology scale. The complexities of both the successful and failed cases rise linearly. In Fig. 5(b), 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 part rises for the node number increases with the services and the rest part fluctuates little even the service scale goes up.

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

8. Conclusion

In this paper, we propose a pervasive service model including both semantic and syntactic descriptions. We organize all the service information in the repository as aggregation graph and explain how this two-layer graph captures the behavior of pervasive services in terms of the input and output parameters and their relations. Facilitated by the aggregation graph, we convert the service composition problem to the problem of finding a sub-tree in this graph and provide an approach searching all the possible composite paths. To prove its efficiency, we theoretically determine the upper bound of its complexity and measure the average performance by simulation. Both of the results show that our scheme has the ability to accomplish user tasks in a very short response time.

9. Acknowledgment

This work is partially supported by the National High-Tech Research and Development Program of China (863) under Grant No. 2006AA01Z199; the National Natural Science Foundation of China under Grant No. 90718031, 60721002; the National Basic Research Program of China (973) under Grant No. 2009CB320705.

References

[1] OWL-S: Semantic markup for web services. http://www.daml.org/services/owls/1.0/owl-s.html, 2003.

[2] Extensible markup language (XML) 1.0 (fifth edition). www.w3.org/TR/REC-xml/, Jan 2006.

[3] Web services description language (WSDL) version 2.0, W3C working draft. http://www.w3.org/TR/wsdl20/, 2006.

[4] M. H. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. DAML-S: Web service description for the semantic web. In *Proc. of 1st International Semantic Web Conference on The Semantic Web*, Oct 2002.

[5] X. Gu, K. Nahrstedt, and B. Yu. Spidernet: An integrated peer-to-peer service composition framework. In *Proc. of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 110–119, Jun 2004.

[6] S. V. Hashemain and F. Mavaddat. A graph-based approach to web services composition. In *Proc. of the 2005 IEEE/IPSJ International Symposium on Applications and the Internet (SAINT'05)*, pages 183–189, Jan 2005.

[7] S. V. Hashemain and F. Mavaddat. A graph-based framework for composition of stateless web service. In *Proc. of 4th European Conference on Web Services (ECOWS'06)*, pages 75–86, 2006.

[8] S. Kalasapur, M. Kumar, and B. A. Shirazi. Dynamic service composition in pervasive computing. In *Proc. of the IEEE Transactions on Parallel and Distributed Systems (TPDS)*, volume 18, pages 907–918, Jul 2007.

[9] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. of the 11th International Conference on World Wide Web (WWW'02)*, pages 77–88, Jan 2002.

[10] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *Proc. of the 11th World Wide Web Conference (WWW'02)*, May 2002.