INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS
Int. J. Commun. Syst. 2009; **22**:1123–1151
Published online 16 April 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/dac.1012

A novel service evolution approach for active services in ubiquitous computing

Peng-Wei Tian[1,2,*,†], Yao-Xue Zhang[1,2], Yue-Zhi Zhou[1,2], Laurence Tianruo Yang[3], Ming Zhong[1,2], Lin-Kai Weng[1,2] and Li Wei[1,2]

[1]*Tsinghua National Laboratory for Information Science and Technology, Beijing, China*
[2]*Department of Computer Science and Technology, Tsinghua University, Beijing, China*
[3]*Department of Computer Science, St Francis Xavier University, Antigonish, Canada*

SUMMARY

With the emergence of more and more personalized service requirements, service customization has become a compelling problem in ubiquitous computing. As a new paradigm for service customization, the *Active Services* reuses existing services and obtains the user-needed service evolved from them. In this paper, a novel service reuse approach is proposed for service evolution process in the active services paradigm. Besides the entire service reuse realized in existing works, our approach can also achieve partial reuse of existing services at functionality level. To generate a user-needed service, the reusable parts of each existing service are extracted with an index strategy and utilized directly, and then the missing functionalities to satisfy the service requirement are further implemented. As quality of service (QoS) is very important for ubiquitous services, in this paper, based on the general service evolution process, two types of QoS-aware service reuse methods are also introduced. The proposed methods have been implemented and extensive experiments were done to evaluate them. The experimental results demonstrate the superiority of our methods in several measurements of service reuse: computation cost, success rate, and the quality of generated services. Copyright © 2009 John Wiley & Sons, Ltd.

Received 22 October 2008; Revised 23 January 2009; Accepted 13 February 2009

KEY WORDS: service evolution; active services; service reuse; ubiquitous computing

1. INTRODUCTION

With the development of information technology and people's proficiency in computer skills, the personalized service becomes more and more prevalent. Ubiquitous computing mainly focuses on human-centric applications, which aim at providing on-demand services and helping users to accomplish the personalized goals. In a ubiquitous computing environment, a large numbers of

*Correspondence to: Peng-Wei Tian, Department of Computer Science and Technology, Tsinghua University, Beijing, China.
†E-mail: tpw04@mails.tsinghua.edu.cn

Copyright © 2009 John Wiley & Sons, Ltd.

devices are usually deployed to provide various services. However, it is rare that there is always an existing service exactly meeting the user requirements. How to generate the user-needed services based on device-provided services has become an important problem to achieve the ubiquitous services accessible to anyone, anytime, and anywhere [1]. In [2], a paradigm called *Active Services* is proposed to address the service customization problem based on service reuse techniques. In this paradigm, existing services approximately matching user requirements are retrieved as reusable services and the user-needed services are then generated by reusing them, which is usually called a service evolution process from reusable services to user-needed ones [2]. This paradigm can be utilized for service customization in ubiquitous computing environments.

In the active services paradigm, service evolution is actually a service reuse process, for which considerable related works have been done. In software engineering, each service is considered as a software component and the service reuse is achieved with component-based software development (CBSD), in which traditional concepts in software engineering are widely used, including the software architecture theory [3, 4], formal specification [5–7], case-based design [8, 9], and so on. Most of the approaches in software engineering are proposed for software development process and are usually abstract and computation-intensive, such as the axiomatic formulas matching in [7], the theorem proving in SPARTACAS [5], and the knowledge reasoning in [9]. In a ubiquitous computing environment, such approaches usually make the services difficult for normal users to use and thus bring bad user experiences.

The emergence of web services and the thriving of service-oriented architecture (SOA) bring us a lightweight approach for service reuse. Each service exposes the functionality based on commonly agreed standards, such as WSDL, Universal Description, Discovery, and Integration (UDDI), and SOAP [10], which are all defined as XML notations and are easy to understand and process. Based on the standard service interfaces, the business process model is widely used to achieve service reuse in a composition way, in which existing services are executed one by one according to users' business processes. Many business process description languages have been proposed for service composition [11], such as Web services flow language, business process modeling language, and business process execution language for Web services (BPEL/BPEL4WS) [12].

The business process model in SOA provides an appropriate service reuse solution to utilize the active services paradigm in ubiquitous computing. Device-provided services can be encapsulated with web service standards and then composed to generate the user-needed services, in which the service composition plan can be defined by existing business process description languages. Given a service requirement, service composition is usually modeled as a service search process. In this process, reusable services are recursively retrieved until the result set satisfies user requirements in specified functionality or signature metrics. Many approaches have been proposed for service composition, such as the forward chaining in Proteus [13], the backward chaining [14], and the A* algorithm-based graph search [15]. In these existing approaches, a service can only be entirely reused in a black-box way, which means that an existing service is regarded as reusable only if it provides identical user-needed functionalities or a subset of the functionalities. With these approaches, device-provided services can always be reused as the basic ubiquitous services. In a ubiquitous computing environment, there are usually another two types of services: the predefined services based on potential business processes and the previously generated services according to user requirements. These services account for increasing percentage of existing services when the number of processed service requirements increases. However, with existing approaches, these two types of services can hardly be reused, because they have relatively complex functionalities that are usually neither identical with nor contained in the user-needed functionalities.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

In this paper, we pay attention to two facts that are usually omitted by existing works: 1. the predefined or previously generated services usually have functionalities overlapping with the user-needed functionalities; 2. a service composed by others can be partially reused based on the composition plan. Grounded on these two facts, we propose a new service reuse approach. Besides the entire service reuse realized in existing works, the proposed approach can also achieve partial reuse of existing services at functionality level. In the approach, device-provided services and the predefined or previously generated services are all indexed based on their functionalities with a two-level index strategy. With the index, existing services having overlapped functionalities with the service requirement can be retrieved as reusable services. Then the functionality similarity and difference are obtained to denote the reusable parts and missing functionalities, respectively, for each reusable service to satisfy the service requirement. Given a reusable service, the similarity parts can be directly utilized, and we only need to add implementation of the difference parts to obtain a user-needed service that is evolved from the reusable service.

Besides the functional service requirements, users' non-functional requirements, i.e. quality of service (QoS), are also very important in ubiquitous computing. In this paper, based on the general service reuse process, two types of QoS-aware service reuse methods are also proposed based on multiple attribute decision making (MADM) model and the branch and bound algorithm.

We implement the proposed methods and perform extensive experiments to evaluate them. The experimental results demonstrate the superiority of our proposed approach in computation cost, service reuse success rate, and the quality of generated services, respectively.

The remainder of the paper is organized as follows. Section 2 introduces the basic concepts used in our approach with an emphasis on the functionality representation and QoS properties. Section 3 provides an overview of the proposed approach. Section 4 describes the functionality-based index process and the two-level index strategy in detail. In Section 5, the general service evolution process and QoS-aware service reuse methods are introduced, respectively. Certain implementation details are given in Section 6 and the experimental results are shown in Section 7. Finally, Section 8 draws a conclusion of the paper.

2. PRELIMINARIES

In this section, we present several basic concepts used in our proposed approach.

2.1. Ubiquitous service description

In the ubiquitous computing environment, each service can be generally described as shown in Figure 1, in which multiple aspects are referred. *Service ID* is the unique identifier of a service. The potential applications in each ubiquitous computing environment usually belong to an *application domain*, and the *domain functionality specification* defines a functional terminology that is commonly used in the application domain for function description. With the terminology, the *functionality representation* describes the functionality of the service in detail. In addition to the functionality, a service always has certain important non-functional properties, such as time cost, price, and so on. The *QoS properties* are used to capture those non-functional properties of a ubiquitous service. The *service descriptor* gives the service interface specification and presents the details for service invoking. In real applications, a service descriptor can be implemented as

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

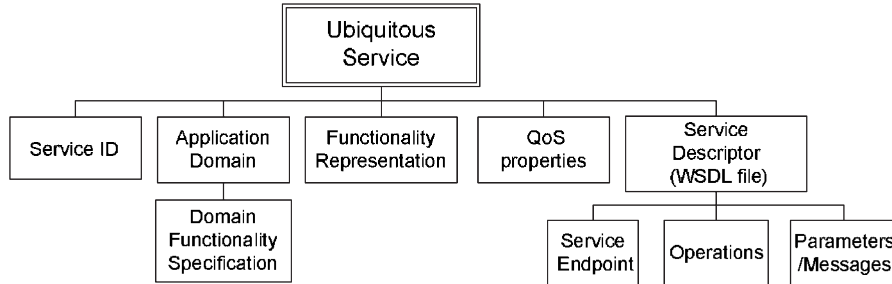

Figure 1. A general description framework for ubiquitous services, including multiple aspects.

a standard WSDL file, which includes the *service endpoint* information, the *operations* contained in the service, and the corresponding *parameters*.

Based on the service implementation process, a ubiquitous service can be classified as an atomic service or a composite service [16]:

- *Atomic service*. The implementation of an atomic service is self-contained and does not invoke any other services. An atomic service can have multiple operations depending on its implementation.
- *Composite service*. The composite service is implemented by composing and invoking other services. A *composition plan* is always attached to a composite service to define its implementation. A composite service usually contains one operation specified by the composition plan.

2.2. Domain functionality specification

The domain functionality specification is usually defined based on XML notations and its schema is shown in Figure 2. It defines a *DomainNum*, denoting the unique identifier number of the domain, a *FunctionalityDesc*, giving a functionality overview of the domain, and a set of *FunctionalUnit*, defining the basic functional units of the domain and serving as the building blocks for functionality description. Each *FunctionalUnit* element is further specified with a *FunctionalUnitNum*, giving the unique identifier number of the functional unit in the domain, a *FunctionalUnitDesc*, briefly describing the functional unit, and a set of *KeywordSynonym*, defining synonyms to describe the functional unit.

2.3. Functionality representation

This section gives the functionality representation for acomic/composite service and service requirement, respectively.

2.3.1. Atomic service functionality. According to the self-contained implementation, the functionality of an atomic service can be specified as a collection of functional units combined based on certain control flows. In the paper, three control-flow structures are introduced to model the functional unit combination [5]: sequential, alternative, and parallel. Here, the loop structure is not included, which can be 'unfolded' into acyclic structures by cloning the loop branches as many times as the loop is taken. The details of the 'unfolding' process can be found as the '*Unfolding*

Copyright © 2009 John Wiley & Sons, Ltd.

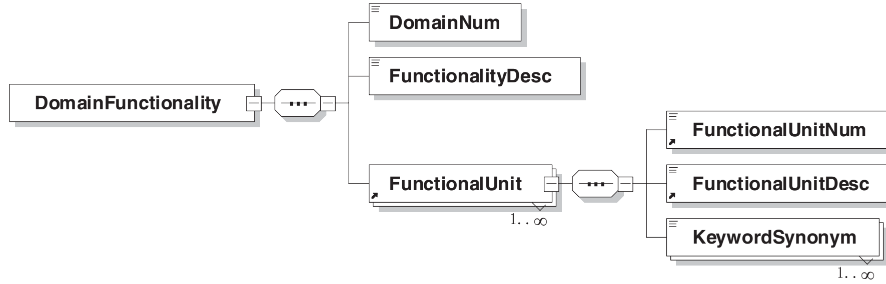

Figure 2. The XML schema definition for domain functionality specification.

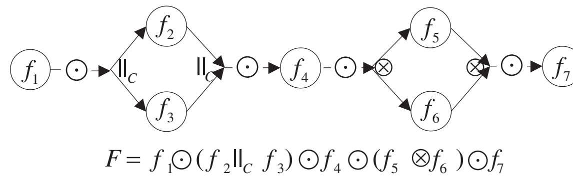

Figure 3. A functional unit combination F and its flow diagram representation.

Cyclic Statecharts' algorithm in [17]. The functional unit combination can be defined by the Backus–Naur form-like notation [18]:

$$F ::= \varepsilon | f | F \odot F | F \|_C F | F \otimes F \tag{1}$$

where ε denotes an empty functional unit, f represents a functional unit constant, denoting a single functional unit-based combination, $\odot$, $\|_C$, and $\otimes$ are three operators denoting the *sequential* structure, *alternative* structure (with control condition C), and *parallel* structure, respectively.

The algebraic notation in (1) verifies the closure property, which means that we can obtain more complex functional unit combinations by aggregating existing ones with the three defined operators. Based on the control-flow structures, a functional unit combination can be represented as a flow diagram, and an example is shown in Figure 3.

Then, given an operation op of an atomic service s, the functionality $fun(s, op)$ can be expressed as $fun(s, op) ::= F$. Based on the number of contained functional units, an operation op in the atomic service can be classified as a simplex or complex functional operation:

- *Simplex functional operation*. The functionality of an simplex functional operation contains only the empty function unit or a functional unit constant, i.e. $fun(s, op) ::= \varepsilon | f$.
- *Complex functional operation*. A complex functional operation is a non-null operation whose functionality contains multiple functional units combined by the operators $\odot$, $\|_C$, and $\otimes$ in (1).

2.3.2. Composite service functionality. For a composite service, the composition structures can also be modeled based on the three control-flow structures [5]: sequential, alternative, and parallel. Then with the control-flow operators introduced in (1), a composite service based on the composition plan cp can be represented as $cp ::= \phi | \langle s, op \rangle | cp \odot cp | cp \|_C cp | cp \otimes cp$, where ϕ is an empty

Copyright © 2009 John Wiley & Sons, Ltd.

atomic service, $\langle s, op\rangle$ is a service/operation pair, in which *op* denotes the performed operation in service s.

For a composite service specified by the composition plan *cp*, its functionality $fun(cp)$ can be obtained by combining the functionality of each service contained in *cp*: $fun(cp) ::= \sigma | fun(s, op) | fun(cp) \odot fun(cp) | fun(cp) \parallel_C fun(cp) | fun(cp) \otimes fun(cp)$, where $\sigma = fun(\phi)$. As $fun(s, op) ::= F$, it can be transformed to $fun(cp) ::= \sigma | F | fun(cp) \odot fun(cp) | fun(cp) \parallel_C fun(cp) | fun(cp) \otimes fun(cp)$. The definition of F in (1) has been verified with its closure property over the three operators, and thus finally we can obtain $fun(cp) ::= F$, which means that the composite service functionality can be represented as the function unit combination.

2.3.3. Functional service requirement. The functional service requirement comprises two parts: user-needed functions and the business process to perform the functions. User-needed functions can be expressed by the functional units defined in the application domain. Control-flow operators introduced in (1) can be used to model the business process. Then the functional service requirement *FSR* can also be represented as the functional unit combination, i.e. $FSR ::= F$.

2.4. QoS properties

This section introduces the QoS criteria for atomic and composite services, respectively.

2.4.1. QoS properties of atomic services. For web services deployed in the internet environment, many QoS properties have been introduced [17, 19], such as price, time cost, availability, reputation, successful execution rate, and so on. In the ubiquitous computing environment, each device is manageable and maintainable, and if invoked correctly the services can usually be ensured to operate smoothly. For an atomic service, we consider two generic QoS properties of time cost and price, without considering the properties measuring the service reliability.

- *Time cost*. The time cost includes two parts: data processing time and the transmission time [17]. A ubiquitous computing environment usually forms a small and manageable LAN, and the transmission time is expected to be negligible comparing with the processing time. In our approach, given an atomic service s and the performed operation *op*, the time cost $q_{tc}(s, op)$ is estimated by the data processing time of the operation.
- *Price*. Some devices may be not free to use. Given an operation *op* of an atomic service s, the price $q_{pr}(s, op)$ is the fee charged for invoking the operation *op* in service s.

The time cost and price are both static properties and can be specified in the service description. With the QoS properties above, the quality vector of an operation *op* in the atomic service s is defined as $q(s, op) = \langle q_{tc}(s, op), q_{pr}(s, op)\rangle$.

2.4.2. QoS properties of composite services. The time cost and price are also employed to evaluate the quality of composite services. The QoS properties of a composite service are estimated based on the services contained in the composition plan and their performed operations:

- *Time cost*. The time cost of a composite service can be estimated with the critical path method (CPM) [17]. For a composition plan *cp*, the critical path is a service sequence from the beginning to the end of *cp*, in which $q_{tc}(s, op)$ of each service s adds up to the largest overall time cost, and this overall time cost in the critical path is used to estimate the time cost of the composition plan: $q_{tc}(cp) = \sum_{\langle s, op\rangle \in critical_path(cp)} q_{tc}(s, op)$, where the $critical_path(cp)$

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

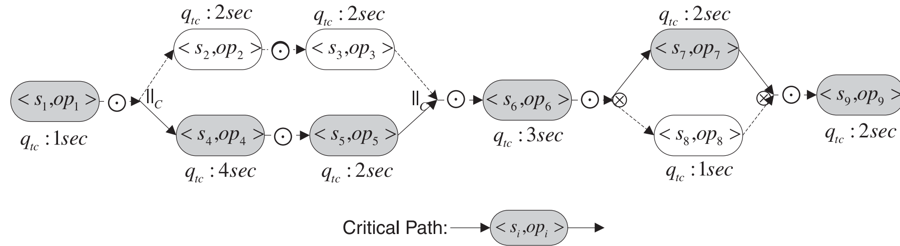

Figure 4. An example to estimate the time cost of a composite service based on CPM. The critical path is $[\langle s_1, op_1\rangle, \langle s_4, op_4\rangle, \langle s_5, op_5\rangle, \langle s_6, op_6\rangle, \langle s_7, op_7\rangle, \langle s_9, op_9\rangle]$ with the time cost 14 s.

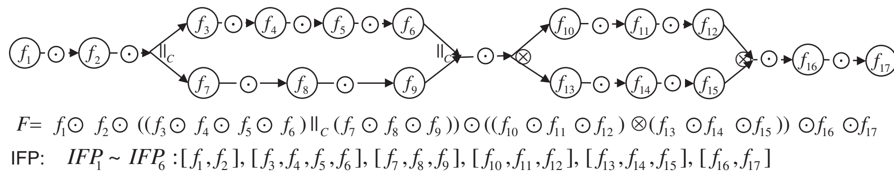

Figure 5. An example of functional unit combination F and its independent function paths.

denotes the critical path of composition plan *cp*. Figure 4 gives an example of estimating the time cost of a composition plan based on CPM. More details can be found in [17].

- *Price*. For a composition plan cp, the price $q_{pr}(cp)$ is the sum of the prices of all operations invoked over the services contained in $cp : q_{pr}(cp) = \sum_{\langle s,op\rangle \in cp} q_{pr}(s, op)$.

Given the QoS properties above, the quality vector of a composite service specified by the composition plan cp is defined as $q(cp) = \langle q_{tc}(cp), q_{pr}(cp)\rangle$.

2.5. Independent function path

Within a functional unit combination, we introduce the concept of *independent function path*, which will be used to generate the functionality-based index of existing services.

Definition 1 (*Independent function path*, *IFP*)
An independent function path in a functional unit combination F is a sequence of functional units $[f_1, f_2, \ldots, f_i, \ldots, f_n]$ such that for each functional unit $f_i (1 \leqslant i \leqslant n)$:

- In F, f_{i+1} is one of the immediate successor functional units of f_i.
- If $i = 1$, f_i is the initial functional unit of F, the beginning functional unit of an alternative or parallel branch in F, or the immediate successor functional unit of an alternative or parallel structure in F.
- There is no functional unit f_j $(1 \leqslant j \leqslant n)$ in $[f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n]$ such that f_i and f_j belong to different alternative or parallel branches in F.

Based on the concept of independent function path, a functional unit combination can usually be simplified. Figure 5 shows a functional unit combination F and its six independent function paths, in which F can be simplified as $F = IFP_1 \odot (IFP_2 \|_C IFP_3) \odot (IFP_4 \otimes IFP_5) \odot IFP_6$.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

3. OVERVIEW OF OUR APPROACH

The flowchart of our approach is illustrated in Figure 6, which comprises two parts: service register and functionality-based index, and user-needed service generation. In a ubiquitous computing environment, existing services are all registered into the service registry. Then, based on a two-level index strategy, the functionality representations of registered services are used to build the functionality-based service index. The index building is a computation-intensive task and usually performed in the offline way. For a newly submitted service requirement, the reusable services are retrieved based on the functionality-based index. Then two metrics, i.e. functionality similarity and difference, are obtained to denote the reusable parts and the missing functionalities, respectively, for each reusable service. With the two metrics, the service generation process is performed to generate a user-needed service evolved from the reusable services. Finally, the generated service is returned to the user/user application, and meanwhile added to the functionality-based service index to incrementally enlarge the index.

In following sections, we will discuss the functionality-based service index and the user-needed service generation processes in detail.

4. FUNCTIONALITY-BASED SERVICE INDEX

For the functionality-based service index, two elements should be considered: the indexed data and the index key. In our approach, reusable service information serves as the indexed data, and for the index key, we employ two different reusable granularities, i.e. a single functional unit and the independent function path (a functional unit sequence), to, respectively, generate keys to index the services whose functionality representation contains them. This is called a two-level index strategy.

4.1. Functional unit-based index

In functional unit-based index, we assign each functional unit an global identifier number as the index key. Given a functional unit f, in the domain functionality specification shown in Figure 2, it belongs to an application domain with the domain identifier $DomainNum(f)$, and f also has its identifier in the domain $FunctionalUnitNum(f)$. These two integer identifiers can be combined to generate the global identifier number of functional unit f as $IdNum(f) = DomainNum(f) \times C +$

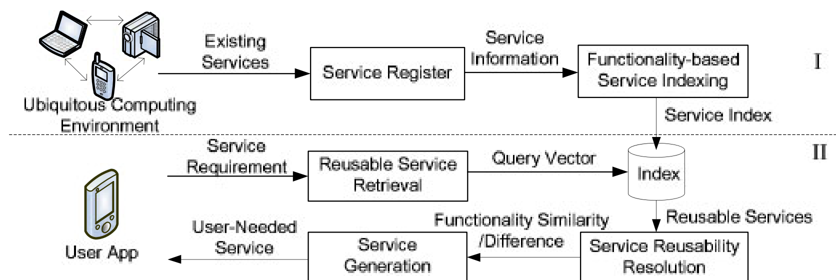

Figure 6. Overview of the proposed approach, two parts included: I. existing service register and functionality-based index (offline) and II. user-needed service generation.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

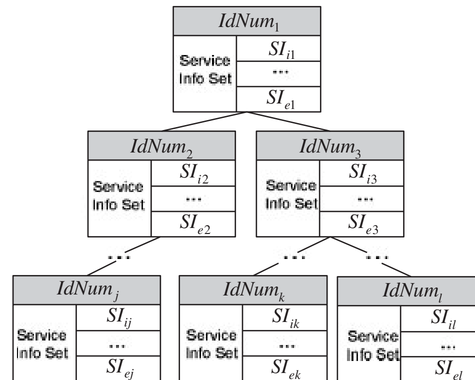

Figure 7. The binary search tree-based structure for functional unit-based service index.

FunctionalUnitNum(f), in which C is an integer constant and any integer number more than the maximum number of the functional units in each domain can be selected as its qualified value. This assignment ensures that any two global identifier numbers are different. In real applications, C can be simply assigned as an exponent of base 10.

The global identifier number of each functional unit is used as the index key to index the services containing the functional unit, and the functional unit-based index is built based on the binary search tree (BST) structure, as shown in Figure 7. Each node of the BST stores the $\langle$*global identifier number*, *service information set*$\rangle$ entries. The *global identifier number* ($IdNum_j$) serves as the key of the BST node. The *service information set* stores all the service information ($SI_{ij}, \ldots, SI_{ej}$) indexed by the global identifier number. With the BST structure, the index building process and the reusable service retrieval process can be achieved with the traditional BST algorithms [20].

4.2. Independent function path-based index

To generate the index key, an independent function path should be quantified first. Given an independent function path $IFP = [f_1, f_2, \ldots, f_i, \ldots, f_n] (1 \leqslant i \leqslant n)$, each functional unit f_i can be uniquely depicted with the global identifier number $IdNum(f_i)$, and then *IFP* can be quantitatively represented by the vector $QIFP = \langle IdNum(f_1), IdNum(f_2), \ldots, IdNum(f_i), \ldots, IdNum(f_n) \rangle$. The quantitative vector can be used to generate the index key for independent function path-based index.

A quantitative vector is usually a multidimensional vector and can hardly be used directly as the index key in most data structures, including the BST. Actually, the multidimensional index is a traditional research problem and has been studied for decades. Space partitioning-based methods have been widely used, such as K-D tree [21, 22], R-tree [23], SR-tree [24], and so on. But experimental results show that most of the space partitioning-based approaches are not time-efficient for high-dimensional data [25, 26]. The index vector length in our approach is not a fixed number and may be assigned a large number according to the services to index.[‡] So the space partitioning-based methods are not suitable to use in our approach. Recently, locality

[‡]In the experiment of the paper, the index vector length is assigned as the average length of the independent function paths contained in existing services.

Copyright © 2009 John Wiley & Sons, Ltd.

sensitive hashing (LSH) [27] has been proposed and proven to be the best so far solution for multidimensional index [26], and we use it for independent function path-based index.

4.2.1. Locality sensitive hashing. The basic idea of LSH is hashing the points into a set of buckets, such that the points close to each other can fall into the same bucket rather than the points with far apart distances. In LSH approach, the crucial concept is the *LSH functions*, which are used to hash each point into the buckets. Given a point in certain metric space, a LSH function can usually transform the point to a single number.

Definition 2 (*LSH functions* [*28*])
Given a metric space S and the distance metric D, a family of hash functions $\mathscr{H}=\{h:S\rightarrow U\}$ is called (r_1, r_2, p_1, p_2)-sensitive $(r_1\langle r_2, p_1\rangle p_2)$ for D if for any points $v, q\in S$

- If $v\in B(q,r_1)$, then $P_{r_{\mathscr{H}}}[h(q)=h(v)]\geqslant p_1$.
- If $v\notin B(q,r_2)$, then $P_{r_{\mathscr{H}}}[h(q)=h(v)]\leqslant p_2$.

In the definition, $P_{r_{\mathscr{H}}}[h(q)=h(v)]$ represents the probability of $h(q)=h(v)$ and $B(q,r)=\{p\in S|D(q,p)\leqslant r\}$, which denotes a ball of radius r centered at q.

The LSH method is proposed to solve the high-dimensional nearest neighbor search problem, i.e. (R,λ,γ)-search problem [29]: given a metric space S, a point set P in S, and the distance metric D, for any query point $q\in S$, each point $v\in P$ satisfying $D(q,v)\leqslant R$ should be retrieved with probability at least λ and each point $p\in P$ satisfying $D(q,p)>R$ should be retrieved with the probability not more than γ. Based on the LSH method, the (R,λ,γ)-search problem can be solved with following steps: 1. based on the parameters of R, λ, and γ, generate an LSH hash family $\mathscr{H}$ with $L\times K$ LSH functions; 2. divide the hash functions into L groups, and in each group, the K functions are combined and entirely serve as one hash function, based on which L hash tables are generated; 3. each point in point set P is hashed into L hash tables as indexed points; 4. given a query point q, it is also hashed into the L hash tables, and in each hash table, the indexed points having same hash code with q will be regarded as the neighbor points.

In [28, 29], the generation of LSH functions and estimation of L and K are given, respectively. We will discuss the related issues in the implementation section of the paper.

4.2.2. LSH-based index process. The independent function path-based index and retrieval can be modeled as an (R,λ,γ)-search process. The index building process is shown in Figure 8. As independent function paths usually vary in their lengths, each quantitative vector $QIFP=\langle IdNum(f_1),\ldots,IdNum(f_n)\rangle$ is first uniformly partitioned into $N(1\leqslant N\leqslant n)$-dimensional sub-vectors, which will be used as the index vectors. In the partition process, $N-1$ elements are overlapped between two adjacent index vectors, and then the quantitative vector *QIFP* divides into an index vector set: $IdxVecS=\{\langle IdNum(f_1),\ldots,IdNum(f_N)\rangle,\langle IdNum(f_2),\ldots,IdNum(f_{N+1})\rangle,\ldots,\langle IdNum(f_{n-N+1}),\ldots,IdNum(f_n)\rangle\}$. Each index vector can be regarded as a point in N-dimensional space and the index vectors of all independent function paths form a point set P. Then a number R is assigned to define the equality between two index vectors, and the values of λ and γ are selected to, respectively, define the expected recall value and fault rate of the retrieval operations on the index.

With the parameters of R, λ, and γ, a nearest neighbor search problem is formed. In our approach, we use l_2-norm distance (i.e. the standard Euclidean distance) as the distance metric. Then the index process can be performed similar to the solution for (R,λ,γ)-search problem. First,

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

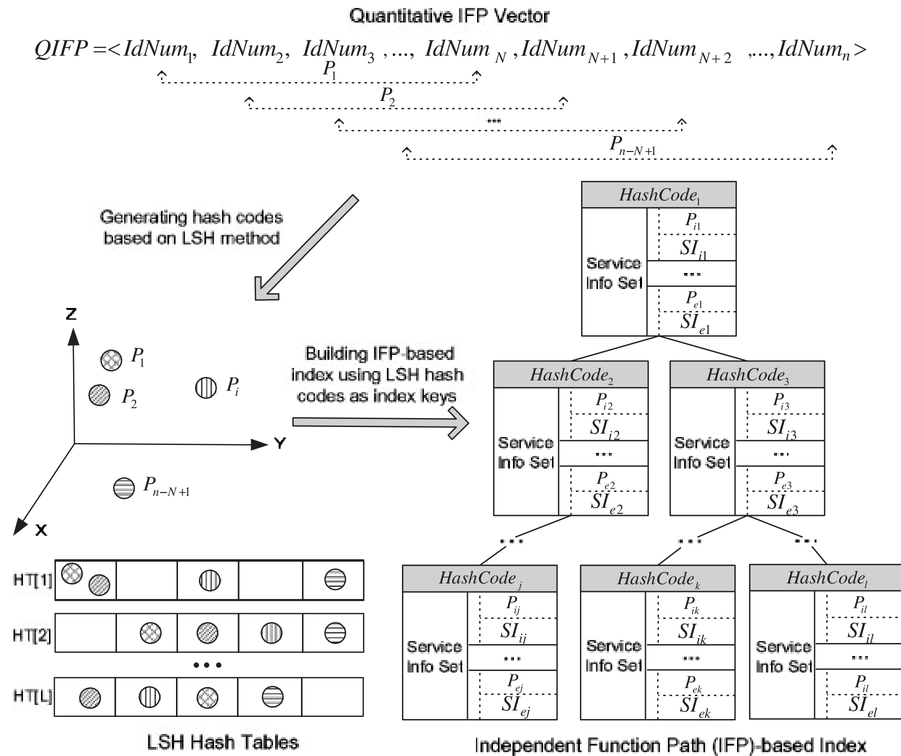

Figure 8. The independent function path (IFP)-based index building process.

L hash tables are built based on $L \times K$ LSH functions. With these hash tables, each index vector in point set P is assigned with L hash codes and each hash code is usually an integer number. Then each hash code can be used as an index key to build the independent function path-based index. In the independent function path-based index, each index vector can generate L index keys and thus index the reusable service information in L indexed items.

Similar to the functional unit-based index, BST is also used as the structure of independent function path-based index. Each node in BST stores the entries of $\langle hashcode, service\ information\ set\rangle$. As shown in Figure 8, in each item of *service information set*, a vector $\langle P_{ij}, SI_{ij}\rangle$ is stored, in which P_{ij} is the N-dimensional index vector generating current index key and SI_{ij} is the indexed service information. As LSH is a probability-based method, when a service information item retrieved, P_{ij} should be compared with the query vector to verify the equality between them. Parameter N can be experimentally selected and will be given in the experimental part of the paper.

4.3. Details of indexed service information

In the functionality-based index, the indexed data (SI_{ij}) is the information of reusable services whose functionality contains the functional unit (sequence) used to generate the index key. As shown in Table I, mainly two parts are included: the reusable service description and the location information of the functional unit (sequence) generating the index key in the reusable service's

Copyright © 2009 John Wiley & Sons, Ltd.

Table I. Details of indexed reusable service information.

Information type	Operation	Indexed service information	
		Fields	Description
Service description	Simplex functional operation	*s_id*	*s_id* is the service ID of the indexed reusable service. It can be used to locate the indexed service (or the composition plan of the indexed composite service).
		s_seq	*s_seq* is a sequence of service/operation pairs in the reusable service, whose functionality is represented by the functional unit (sequence) generating the index key. If a composite service is indexed, the parts containing *s_seq* in the composition plan can be extracted as reusable parts.
	Complex functional operation	*s_beg/s_end*	$s_beg/s_end \in \{0, 1\}$. If the functional unit (sequence) used to generate the index key locates at beginning/end of the functionality of an atomic service, $s_beg/s_end = 1$; otherwise, they are assigned value 0.
Location information	Simplex functional operation	*ifp_id*	*ifp_id* is the identifier of the IFP containing the functional unit (sequence) generating the index key.
		loc	*loc* denotes location of the functional unit (sequence) generating the index key in the IFP identified by *ifp_id*.
		fnum	*fnum* is the number of the functional units contained in the functional unit (sequence) used to generate the index key.
	Complex functional operation	*pre_op/ after_op*	$pre_op/after_op \in \{\odot, \|_C, \otimes\}$. If the functional unit (sequence) generating the index key locates at the beginning/end of an alternative or parallel branch, they are assigned $\|_C$ or $\otimes$; otherwise, $pre_op/after_op = \odot$.
		partner_beg_ifps	When $pre_op = \|_C$ or $\otimes$, *partner_beg_ifps* identifies the beginning IFPs of other branches in the same alternative or parallel structure with the IFP identified by *ifp_id*. If *pre_op* equals $\odot$, *partner_beg_ifps* is a empty set.
		pre_ifp	*pre_ifp* identifies the previous IFP that directly connects to the IFP identified by *ifp_id* based on the operator $\odot$.
		ifps	*ifps* denotes an IFP sequence. It is mainly used in the service generation process for service information aggregation. By default, *ifps* = [*ifp_id*].

functionality representation. The reusable service description is mainly used to identify and locate the reusable atomic services or the reusable parts of composite services. In addition, different informations are indexed to depict simplex and complex functional operations, respectively.

4.4. Two-level index strategy-based index process

Based on the two types of indexes, the functionality-based service index process is performed in the offline way. Given a reusable service S and its functionality representation F, the index process

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

is performed based on each independent function path *ifp* contained in F. If the length of *ifp* is not less than the dimension of the index vector in IFP-based index, the reusable service information of S will be indexed in the IFP-based index based on *ifp*. As an atomic service, S can only be entirely reused, and if it is indexed by *ifp*, the functional units contained in *ifp* will not index it again. If S is a composite service, it will be indexed by both the independent function path *ifp* and the functional units contained in *ifp*. With the index process, functionality-based service index is built.

5. SERVICE GENERATION

With the functionality-based service index, reusable services can be retrieved and then used to generate user-needed services. In this section, a service generation method is first proposed, based on which two QoS-aware service generation methods are then introduced.

5.1. The basic idea

The basic idea for service generation is using reusable services to implement the user-needed functionalities and finally transform the functional service requirement into a composition plan specifying the user-needed service, which is shown in Figure 9. The concept *Implementation Action* is used to denote the procedure that a reusable service implements the user-needed functionalities contained in it. In Figure 9, five implementation actions are performed in the service generation process.

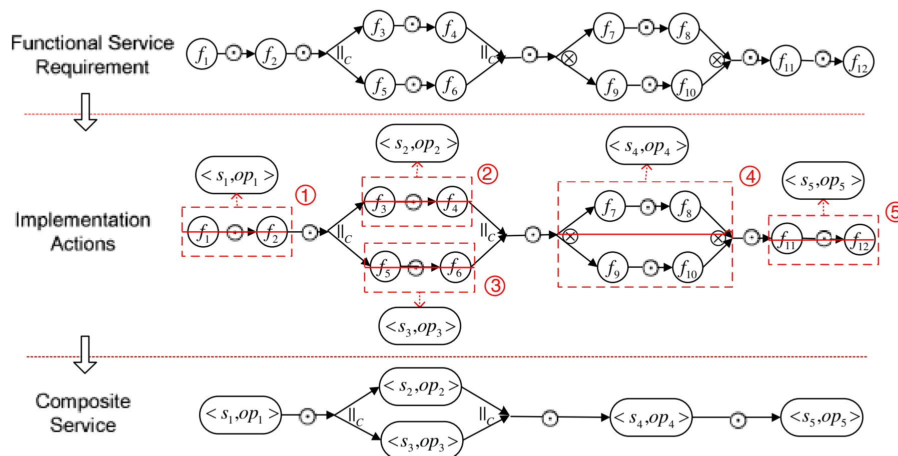

Figure 9. The basic idea of service generation; five implementation actions are performed to transform the functional service requirement to the composition plan of the user-needed service.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

5.2. Reusable service retrieval

In our approach, existing services having functionality overlapping with the functional service requirement are regarded as reusable services. Given a functional service requirement, the contained independent function paths and functional units are, respectively, used to retrieve reusable services.

For each independent function path in the functional service requirement, the reusable service information is retrieved from the independent function path-based service index, as demonstrated in Algorithm 1. The LSH-based hash tables and the input parameters N and R used in the retrieval process are same with those used in the IFP-based index process shown in Figure 8. As the query, the independent function path is represented by its quantitative vector *QIFP*, and *QIFP* is partitioned into a set of N-dimensional query vectors. For a query vector *QueVec*, its hash code in each hash table is obtained to retrieve the reusable service information items in the IFP-based index. The vector P stored in each item is used to verify whether it is actually equal to *QueVec*. When the retrieval processes for all independent function paths in the functional service requirement have been performed, the retrieval results are returned as a set of query vector/service information set pairs: $IFPRes = \{\langle QueVec, SISet\rangle\}$, in which *QueVec* is a query vector in the functional service requirement and *SISet* denotes the service information set retrieved by *QueVec*.

Algorithm 1 Independent function path-based service retrieval.

Input: The quantitative vector of an independent function path *QIFP* ;
The number N, denoting the dimension of the index vector in IFP-based index;
The number R, defining the equality between two index vectors;
LSH hash table set HT with hash functions: $G = \{g : S \rightarrow U^K\}$, $|HT| = |G| = L$;
Output: The retrieved service information set based on each index vector in *QIFP*;

1: $IFPRes \leftarrow \emptyset$;
2: **if** $|QIFP| < N$ **then**
3: return *IFPRes*;
4: **end if**
5: **for** $i = 0$ to $|QIFP| - N + 1$ **do**
6: $SISet \leftarrow \emptyset$;
7: $QueVec = \langle QIFP[i], QIFP[i+1], \ldots, QIFP[i+N-1]\rangle$;
8: **for** $j = 0$ to $L-1$ **do**
9: In hash table $HT[j]$,generate the hash code of *QueVec* as $g_i(QueVec)$;
10: $IdxItems \leftarrow$ items indexed by $g_i(QueVec)$ in IFP-based index (with BST algorithm);
11: **for** $k = 0$ to $IdxItems.count - 1$ **do**
12: **if** $IdxItems.P_k == QueVec$ **then**
13: $SISet = SISet \cup \{IdxItems.SI_k\}$;
14: **end if**
15: **end for**
16: **end for**
17: $IFPRes = IFPRes \cup \{\langle QueVec, SISet\rangle\}$;
18: **end for**
19: return *IFPRes*;

For each functional unit f in the functional service requirement, the global identifier number $IdNum(f)$ is used as the query for reusable service information retrieval in the functional

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

unit-based index based on traditional BST algorithm [20]. The retrieval results for all functional units contained in the functional service requirement constitute the functional unit-based retrieval results: $FURes = \langle\langle IdNum(f)\rangle, SISet\rangle$, in which $\langle IdNum(f)\rangle$ is a query vector containing the identifier number of a functional unit f in the functional service requirement and $SISet$ is a set of reusable service information indexed by $IdNum(f)$ in the function unit-based index.

The independent function path-based retrieval results, i.e. *IFPRes*, and functional unit-based retrieval results, i.e. *FURes*, together form the reusable service information set for the given service requirement.

In the retrieval results above, based on different query vectors, the reusable service information (*SI*) may not denote complete atomic services and thus cannot be reused directly. The service information items denoting incomplete atomic services should be aggregated such that their aggregations can denote complete ones. As we have said, a service information item in the retrieval results is in the form of a vector $\langle QueVec, SI\rangle$. If $SI.s_beg = 1$, $SI.s_end = 1$, and *partner_beg_ifps* is empty, the reusable parts denoted by *SI* are complete atomic services. Otherwise, *SI* denotes incomplete atomic services and should be aggregated. For any two retrieved items $\langle QueVec_1, SI_1\rangle$ and $\langle QueVec_2, SI_2\rangle$, when SI_1 and SI_2 belong to the same reusable service, i.e. $SI_1.s_id = SI_2.s_id$, they can be aggregated. In the aggregate process, a new service information item is generated, for which the aggregation of $SI_1.s_seq$ and $SI_2.s_seq$ serves as the s_seq field and $QueVec_1$ and $QueVec_2$ are merged to be the query vector. Service information aggregate processes are performed on *IFPRes* and *FURes*, respectively. After that, items still denoting incomplete atomic services will be removed. Finally, all the service information items in *IFPRes* and *FURes* denote complete atomic services for reuse.

5.3. Service reusability resolution

Based on the retrieved service information, the functionality similarity/difference and reusability can be obtained for each reusable service.

5.3.1. Functionality similarity. The functionality similarity between the functional service requirement and each reusable service figures out what functionalities are commonly contained in them and which parts of the reusable service can be utilized to implement the commonly contained functionalities.

The functionality similarity is computed with the independent function path-based retrieval results *IFPRes*. The reusable service information in *IFPRes* is classified by query vectors, to compute the functionality similarity; we first cluster the information based on the reusable services. The clustering process is shown in Algorithm 2. A hash table *ServRes_Table* is returned as the clustering results, in which the service id *Sid* is used as the key and service information set *SISet* is stored as values.

With the clustered reusable service information, the functionality similarity can be obtained based on each independent function path of the functional service requirement. For the functional service requirement *FSR* and a reusable service S (an atomic service or a composite service specified by the composition plan), the functionality similarity can be defined as

$$sim(FSR, S) = \{R_i \mid R_i = \langle ifps_i, begloc_i, endloc_i, s_seq_i\rangle, i \geqslant 0\} \quad (2)$$

where R_i is the ith functionality overlapping between *FSR* and S, $ifps_i$ identifies a sequence of adjacent independent function paths belonging to the functional service requirement and contained

Algorithm 2 Service information clustering based on reusable services.

Input: The independent function path-based service retrieval result set: *IFPRes*;
Output: The service information set based on each reusable service;

1: $ServRes_Table \leftarrow$ empty hash table;
2: **for** $i = 0$ to $IFPRes.count - 1$ **do**
3: $SISet \leftarrow IFPRes[i].SISet$;
4: **for** $j = 0$ to $SISet.count - 1$ **do**
5: $Sid \leftarrow SISet[j].s_id$;
6: **if** $ServRes_Table$ contains the key Sid **then**
7: $ServRes_Table[Sid] = ServRes_Table[Sid] \cup \{SISet[j]\}$;
8: **else**
9: $ServRes_Table.Add(Sid, \{SISet[j]\})$;
10: **end if**
11: **end for**
12: **end for**
13: return $ServRes_Table$;

in R_i, $begloc_i$ denotes the starting position of R_i in the first independent function path of $ifps_i$, i.e. $ifps_i[0]$, $endloc_i$ denotes the end position of R_i in the last independent function path of $ifps_i$, i.e. $ifps_i[ifps_i.length - 1]$, s_seq_i is a sequence of adjacent service/operation pairs in reusable service S, whose functionality contains R_i. If S is an atomic service, s_seq_i identifies the performed operation of S.

Usually, R_i denotes that, in reusable service S, the parts containing the service/operation sequence s_seq_i can be reused to implement the adjacent independent function paths identified by $ifps_i$ in the functional service requirement from the $begloc_{ith}$ functional unit of $ifps_i[0]$ to the $endloc_{ith}$ functional unit of $ifps_i[ifps_i.length - 1]$.

Based on the reusable services clustering results *ServRes_Table*, the similarity between the functional service requirement *FSR* and each reusable service *S* (with the identifier *Sid*) contained in *ServRes_Table* can be computed. For each service information item *SI* ($SI \in SISet$ indexed by *Sid*) in *ServRes_Table*, a functionality overlapping $R_i = \langle ifps_i, begloc_i, endloc_i, s_seq_i \rangle$ in the functionality similarity $sim(FSR, S)$ can be computed as $ifps_i = SI.ifps$, $begloc_i = SI.loc$, $s_seq_i = SI.s_seq$, if $SI.pre_op$ and $SI.after_op$ both equal $\|_C$ or $\otimes$; $endloc_i$ denotes the end functional unit of the final independent function path in $ifps_i$, i.e. $endloc_i = |ifps_i[ifps_i.length - 1]|$, and otherwise $endloc_i = SI.loc + SI.fnum$.

5.3.2. Service reusability and similarity vector. With the functionality similarity $sim(FSR, S)$ defined in (2), the reusability of service S can be measured by the number of the functional units contained in all the overlapped functionalities between *FSR* and *S*:

$$Reusability(FSR, S) = \sum_{i=0}^{sim(FSR,S).count-1} \left(|ifps_i[0]| - begloc_i + 1 + endloc_i + \sum_{k=1}^{ifps_i.length-2} |ifps_i[k]| \right) \quad (3)$$

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

With the reusability metric, similarities between the functional service requirement and each reusable service contained in *ServRes_Table* form a similarity vector: $SIM(FSR, ServRes_Table) = \langle sim(FSR, S_j) | S_j \text{ is contained in } ServRes_Table, j \geqslant 0 \rangle$, in which each element $sim(FSR, S_j)$ is ranked based on $Reusability(FSR, S_j)$ such that given two services S_m and $S_n (m, n \geqslant 0)$, if $Reusability(FSR, S_m) > Reusability(FSR, S_n)$, $sim(FSR, S_m)$ will strictly precede $sim(FSR, S_n)$ in $SIM(FSR, ServRes_Table)$. In service generation process, the service (S_m) whose similarity has smaller index in $SIM(FSR, ServRes_Table)$ will be reused in higher priority.

5.3.3. Functionality difference. Functionality difference between the functional service requirement and each reusable service denotes the functionalities contained in the service requirement but not covered by the reusable service, i.e. the missing functionalities for the reusable service to satisfy the service requirement. Given the functional service requirement *FSR* and a reusable service *S*, the functionality difference $diff(FSR, S)$ is constituted by each functional unit sequence contained in *FSR* but not in *S*:

$$diff(FSR, S) = \{D_i | D_i = \langle ifp_i, begloc_i, endloc_i \rangle, i \geqslant 0\} \tag{4}$$

where D_i denotes the ith functional unit sequence, which is contained in *FSR* but not in *S*; ifp_i identifies the independent function path in *FSR*, which contains D_i; $begloc_i$ and $endloc_i$, respectively, denote the beginning and end position of D_i in ifp_i.

Intuitively, the functionality difference can be computed based on the substraction between the functional service requirement and functionality similarity, i.e. $diff(FSR, S) = FSR] - sim(FSR, S)$. For an independent function path *ifp* in *FSR*, we can search for it in each functionality overlapping $R_i = \langle ifps_i, begloc_i, endloc_i, s_seq_i \rangle$ of $sim(FSR, S)$. If *ifp* is equal to $ifps_i[0]$, the functional units from $begloc_i$ to the end will be removed from *ifp*, and if *ifp* is equal to $ifps_i[ifps_i.length - 1]$, functional units from beginning to $endloc_i$ will be removed from *ifp*; Otherwise, if *ifp* is equal to $ifps_i[j] (0 < j < ifps_i.length - 1)$, all functional units will be removed from *ifp*. Finally, all the functional unit sequences left in *ifp* will form $diff(FSR, S)$.

Corresponding to each element in similarity vector $SIM(FSR, ServRes_Table)$, the functionality difference between the service requirement and each reusable service form the difference vector, $DIFF(FSR, ServRes_Table)$, in which $DIFF(FSR, ServRes_Table)[j] = FSR - SIM(FSR, ServRes_Table)[j] (j \geqslant 0)$.

5.4. Service generation process

With the functionality similarity and difference obtained above, user-needed service generation process can be performed. In the process, the reusable services with high reusability will be reused in high priority. For a reusable service, the functionality similarity parts can be directly utilized to implement the overlapped functionalities. For the functionality difference, the reusable services obtained from the functional unit-based service retrieval results *FURes* will be used to implement them.

Algorithm 3 describes the service generation process in detail, which is an iterative process. In functionality similarity vector $SIM(FSR, ServRes_Table)$, the element with smaller index $sim(FSR, S)$ is first processed. The composition plan *Com_Plan* of the user-needed service is initialized to *FSR*, and two phases are included in the algorithm: the IFP-based and the functional

Copyright © 2009 John Wiley & Sons, Ltd.

unit-based reuse phase. During the first phase, all overlapped functionalities in *Com_Plan* are implemented by the reusable parts extracted from service S. In the second phase, the reusable services to implement the functional unit sequences in $diff(FSR, S)$ are obtained from *FURes* to perform implementation actions on *Com_Plan*. After the two phases, if there is no functional unit left in *Com_Plan*, the service generation process exits successfully and *Com_Plan* is returned as the composition plan of the user-needed service; otherwise, the next element in $SIM(FSR, ServRes_Table)$ will be used to start a new service generation process. After all functionality similarities have been processed, if no composition plan is generated, service generation fails and returns null. The service generation can be regarded as an evolution process from the reusable service S to the user-needed service specified by *Com_Plan*.

Algorithm 3 Service generation process.

Input: Functional service requirement:*FSR*;
Clustered reusable service information table:*ServRes_Table*;
Functionality similarity vector:$SIM(FSR, ServRes_Table)$;
Functionality difference vector:$DIFF(FSR, ServRes_Table)$;
Functional unit-based service information retrieval results:*FURes*;
Output: A user-needed service specified by the composition plan;

1: **for** $i=1$ to $|SIM(FSR, ServRes_Table)|-1$ **do**
2: $Com_Plan \leftarrow FSR$;
3: $S \leftarrow$ the reusable service that $sim(FSR, S)=SIM(FSR, ServRes_Table)[i]$;
4: {IFP-based service reuse phase};
5: $sim=SIM(FSR, ServRes_Table)[i]=\{R_j|R_j=\langle ifps_j, begloc_j, endloc_j, s_seq_j\rangle, j\geqslant 0\}$;
6: **for** $j=0$ to $sim.count-1$ **do**
7: {Perform functional implementation actions on *Com_Plan* based on $sim[j]$:};
8: From S, extract the reusable parts containing the service/operation sequence s_seq_j;
9: Using the extracted parts to implement the adjacent IFPs in *Com_Plan* from the $begloc_j th$ functional unit of $ifps_j[0]$ to the $endloc_j th$ one of $ifps_j[ifps_j.length-1]$;
10: **end for**
11: {Functional unit-based service reuse phase};
12: $diff=DIFF(FSR, ServRes_Table)[i]=\{D_k|D_k=\langle ifp_k, begloc_k, endloc_k\rangle, k\geqslant 0\}$;
13: **for** $k=0$ to $diff.count-1$ **do**
14: $ResSet \leftarrow \{\langle QueVec_m, SISet_m\rangle \in FURes|QueVec_m \in \langle ifp_k[begloc_k], \ldots, ifp_k[endloc_k]\rangle, m\geqslant 0\}$, and each element in $ResSet$ is ranked by $|QueVec_m|$;
15: **for** $m=0$ to $ResSet.count-1$ **do**
16: {Perform functional implementation actions on *Com_Plan* based on $diff[k]$:};
17: In *Com_Plan*, use the services denoted by $SISet_m$ to implement the functional unit sequence $QueVec_m$, and the composed services should be interface-compatible;
18: **end for**
19: **end for**
20: **if** *Com_Plan* contains only services and performed operations **then**
21: return *Com_Plan*;
22: **end if**
23: **end for**
24: return *null*;

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

5.5. *QoS-aware service generation*

As there usually exist services providing the same functionality but having different QoS properties, QoS-aware service generation has to obtain the service providing user-needed functionalities, meanwhile satisfying users' QoS preference. Our proposed service generation method is an iterative process and multiple potential solutions may exist to generate the user-needed service. In Algorithm 3, the first generated service is returned. Based on QoS properties, we can control the iterative process to generate a solution satisfying users' non-functional requirements. As introduced in the preliminaries, in this paper, two QoS properties are considered for each service s (an atomic service/performed operation or a composite service specified by the composition plan): time cost $q_{tc}(s)$ and price $q_{pr}(s)$, and the two properties form the quality vector of s: $q(s) = \langle q_{tc}(s), q_{pr}(s)\rangle$.

In the paper, we consider two types of QoS preferences: the QoS threshold-based preference and the optimal QoS preference. For the QoS threshold-based preference, users should set threshold values for each QoS property, and if all the QoS properties of a service hold better values than the corresponding threshold values, the service is regarded as satisfying the QoS requirement. In the optimal QoS preference, users usually want to obtain the services with optimal quality among the potential solutions.

5.5.1. QoS threshold-based preference. In QoS threshold-based preference, the threshold values are first set for each QoS property of the service according to user requirement. As the time cost and price are both cost criteria, i.e. the higher the value, the lower the quality, the maximum values should be set for them as threshold values: $tc_{\max}$ and $pr_{\max}$. Then QoS threshold-based preference can be added to the service generation process given in Algorithm 3. For the service generation process based on the functionality similarity $SIM(FSR, ServRes_Table)[i]$, after each implementation action in IFP-based reuse phase or functional-unit based reuse phase, quality vector of *Com_Plan* can be calculated as $q(Com_Plan) = \langle q_{tc}(Com_Plan), q_{pr}(Com_Plan)\rangle$. In the calculation, each functional unit f contained in *Com_Plan* is ignored, i.e. $q_{tc}(f) = 0$ and $q_{pr}(f) = 0$. Then if $q_{tc}(Com_Plan) \leqslant tc_{max}$ and $q_{pr}(Com_Plan) \leqslant pr_{max}$, the generation process can continue; otherwise, current process should be interrupted and the next element $SIM(FSR, ServRes_Table)[i+1]$ is used to start a new service generation process. With this QoS threshold-based service generation process, the first generated service meeting the QoS threshold requirement is returned.

5.5.2. Near-optimal QoS preference. To obtain the service with optimal quality, we should first know how to judge whether the quality of a service is optimal or not. In this paper, the time cost and price are together used to depict the service quality and the MADM [17, 30] model can be employed for judging the optimality of service quality. Many approaches have been proposed to solve the MADM problem [30], such as linear assignment method, simple additive weighting (SAW) method, Technique for Order Preference by Similarity to Ideal Solution (TOPSIS), and so on, among which SAW is a method with the least computation. In the paper, the SAW method is adopted for optimal quality judgement.

Given a set of candidate services $S = \{s_1, s_2, \ldots, s_i, \ldots, s_n\}(1 \leqslant i \leqslant n)$, the quality vectors of all candidate services constitute the quality matrix $Q(S) = [q_{ij} | q_i = q(s_i); 1 \leqslant i \leqslant n; 1 \leqslant j \leqslant 2]$. The SAW method requires a comparable scale for all elements in $Q(S)$. Both the time cost and price are

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

cost criteria in MADM, and the comparable value for each element can be obtained as

$$q'_{ij} = \frac{\min_{1\leqslant k\leqslant n}(q_{kj})}{q_{ij}} \tag{5}$$

In (5), when the number of the services in candidate set changes or the QoS properties of the services are modified, the $\min_{1\leqslant k\leqslant n}(q_{kj})$ must be recalculated. To reduce the computational cost and make the method applicable on changing services, in this paper, the value of $\min_{1\leqslant k\leqslant n}(q_{kj})$ is estimated with the minimum value of the jth dimension among all the services in the ubiquitous computing environment: $minq_j$, which is a static value and can usually be calculated beforehand. Then the comparable value for each element can be estimated by

$$q'_{ij} = \frac{minq_j}{q_{ij}} \tag{6}$$

With each q'_{ij}, the comparable value-based quality matrix is built as $Q'(S) = [q'_{ij} | 1\leqslant i\leqslant n; 1\leqslant j\leqslant 2]$. Then for a service s_i, its quality can be measured by the quality score:

$$Service_Quality_Score(s_i) = \sum_{j=1}^{2}(q'_{ij} \times W_j) \tag{7}$$

where $W_j \in [0, 1]$ and $\sum_{j=1}^{2}(W_j) = 1$. W_j denotes user-preferred weight on the jth quality dimension. In our method, the weights on time cost and price are both 0.5 by default, which can be adjusted by users' preferences. The quality score can be used for optimal quality judgement and in the candidate set, the service with highest quality score can be regarded as the service of optimal quality.

With the SAW method to judge the optimality of service quality, the branch and bound algorithm [31] is employed to model the optimal QoS-based service generation process. As shown in Figure 10, the process is revised from the general service generation method given in Algorithm 3. The process divides into two phases: the initialization phase and the branch and bound algorithm-based optimal service search phase. In the initialization phase, first T elements in the functionality similarity vector *SIM*(*FSR*, *ServRes_Table*) are selected for service generation and accordingly the candidate service set is formed with T services. The quality score of each candidate service *QSCORE*[i] is assigned 0 and the composition plan of each candidate service *CP*[i] is initialized to the functional service requirement *FSR*. In the branch and bound algorithm-based search phase, the candidate service *CP*[*Cur*] with the highest quality score is selected. The implementation actions are first performed on *CP*[*Cur*] based on the similarity *SIM*(*FSR*, *ServRes_Table*)[*Cur*], and each time the overlapped functionalities denoted by *Iterator*[*Cur*] are implemented. Then, the implementation actions are performed on the functional unit sequences denoted by functionality difference *DIFF*(*FSR*,*ServRes_Table*)[*Cur*]. In the process, after each implementation action, the quality score of *CP*[*Cur*] is recalculated and the candidate service with highest score is reselected for next implementation action. If a candidate service still contains unimplemented functional units after all the implementation actions, it will be removed from the candidate set. During the process, if a composition plan is generated with the highest quality score, it is returned as the result.

In the algorithm shown in Figure 10, the IFP-based reuse considers the quality optimality for each implementation action. In the functional unit-based reuse phase, to simplify the algorithm, all performed implementation actions are treated as a whole for optimal quality judgement, without

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

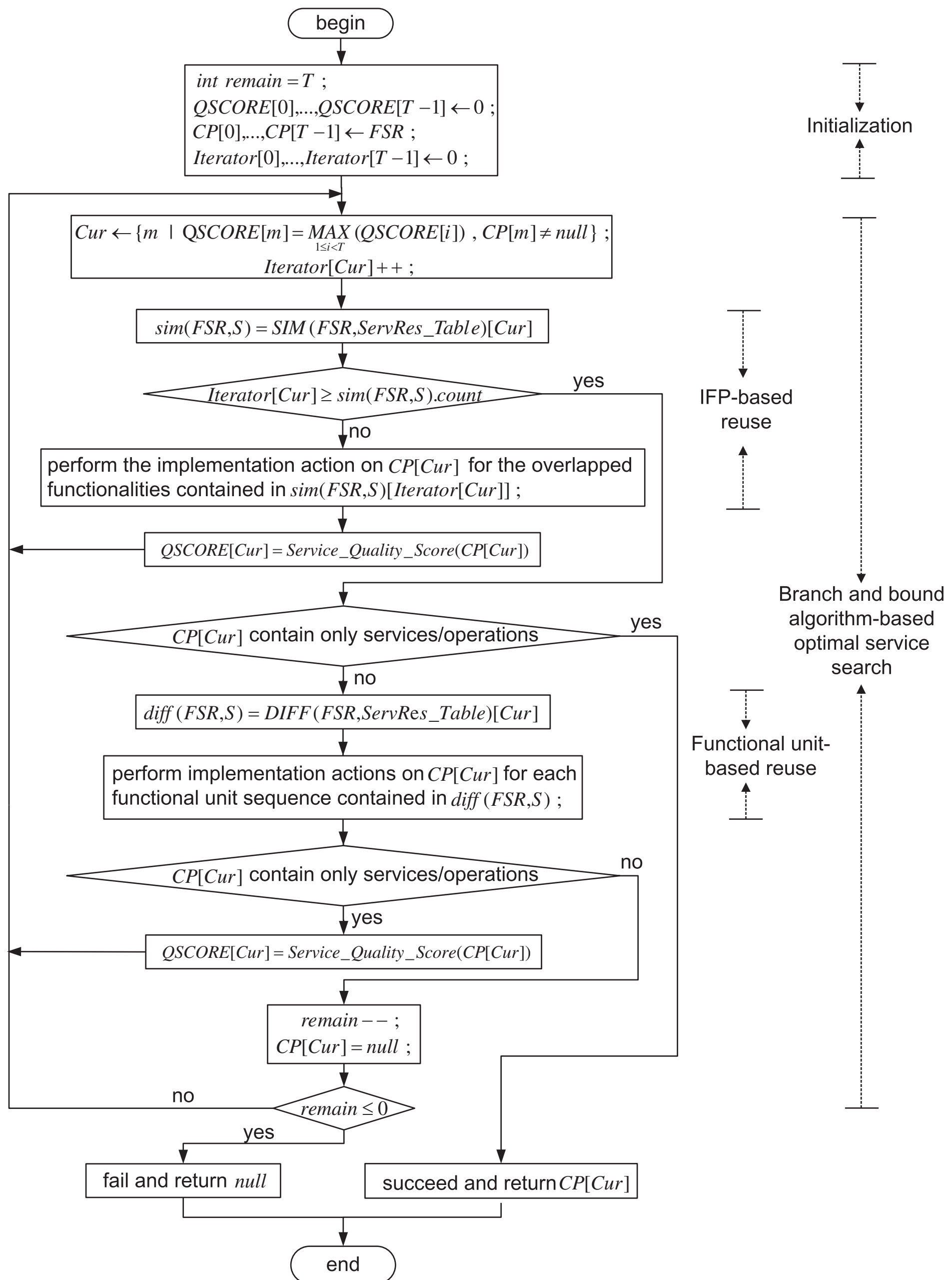

Figure 10. Near-optimal QoS-based service generation process.

Copyright © 2009 John Wiley & Sons, Ltd.
Int. J. Commun. Syst. 2009; **22**:1123–1151
DOI: 10.1002/dac

considering the local optimality for each of them. Then the proposed method is actually a near-optimal QoS solution.

6. IMPLEMENTATION DETAILS

6.1. Service registry

In our approach, besides the general service information included in a WSDL file, some detailed functionality information and QoS information are also needed. An extended UDDI registry is implemented in our proposed approach. The WSDL file of each service is registered to the UDDI registry and associated with a *tModel* [32] as usual. For each service, the *functionality representation* and *QoS properties* are respectively defined as XML documents. Then, besides providing the query interface for the service information included in a WSDL file, the UDDI registry also parses the XML files and provides the interface to query the functionality and QoS information of each service.

6.2. LSH function family generation

Many works have been done to address the LSH family generation issues based on different distance metrics. In our approach, the distance between index vectors is measured with l_2-norm distance, i.e. the standard Euclidean distance. In [28], an LSH family generation scheme is proposed for l_2-norm distance based on the standard Gaussian distribution with density function $g(x)=1/\sqrt{2\pi}\mathrm{e}^{-x^2/2}$. According to the scheme, in our approach, for an N-dimensional index vector v, hash function $h_{a,b}$ can be defined as $h_{a,b}(v)=\lfloor (a*v+b)/r \rfloor$, where a is an N-dimensional vector with the entries randomly chosen from the standard Gaussian distribution and b is a real number chosen uniformly from the range $[0,r]$. The value of r is experimentally discussed in [28], and in our approach, r is assigned with 4.0. In addition, the optimal values of the parameters L and K in LSH method can be obtained by solving an optimization problem, and the details can be found in [29].

6.3. Service composition plan and engine

We employ BPEL [12] as the service composition language in our approach. Three structured activities in BPEL are used to specify a composition plan: $\langle sequence \rangle$, $\langle switch \rangle$, and $\langle flow \rangle$, and the loop structure $\langle while \rangle$ is not directly used in our approach, and the open source BPEL engine ActiveBPEL [33] is used in our approach to execute the composition plans and generate user-needed services.

7. EXPERIMENTAL RESULTS

7.1. Experimental setup

In the experiments, the service registry, indexer, and generator were, respectively, implemented and deployed in a PC with the Intel Pentium IV 1.8 GHz CPU and 1 G RAM, a PC with the Intel Pentium IV 2.8 GHz CPU and 1 G RAM, and an IBM laptop computer with the Intel Core II 2 GHz CPU and 1 G RAM. Then, based on a reusable service library and the predefined parameters, the

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

Table II. Parameters used in the experiments.

Parameters	Description	Value
C	The integer constant to generate the functional unit global identifier	100
R	The number used to define the equality between index vectors	1
λ	Probabilistic recall value of LSH scheme	0.8
γ	False rate value adopted in LSH scheme	0.02
N	Dimension of the index vector	6
W_j	User-preferred weight on the jth quality dimension	0.5
T	The candidate service number in near-optimal QoS-based method	15
L	The hash table number used in LSH method	9
K	The number of LSH functions constituting one hash table	8

computation cost, the success rate, and the quality of generated services of the proposed methods were evaluated.

7.1.1. Service library. In the experiments, a collection of scientific computing-related services were developed and then the experimental application domain turned into scientific computing. Totally 10 commonly used basic computing operations were employed as the functional units in the environment, such as *add*, *subtract*, *multiply*, and so on, and the same functional unit can be iteratively used in the service functionality representation and the functional service requirement. Then 300 atomic services containing 2 or more functional units were implemented based on different control structures and served as the device-provided services, each of which provides only one operation. With the atomic services, 100 composition plans were then designed as existing composite services in the environment. For an atomic service containing n functional units, the QoS properties were randomly assigned with the time cost ranging from n to $60n$ s and the price ranging from n to $200n$. Then, in the experiments, the minimum time cost and minimum price both are equal to 2. The QoS properties of each composition plan were calculated based on the services contained in it. In addition, the functionality representation of each atomic service was manually defined in the service development process, and the functionality representation of each composition plan is automatically generated based on the services contained in it with a tool we developed.

7.1.2. Parameter settings. Table II summarizes the values of the major parameters used in the experiments. The average length of the independent function paths in the existing services and composition plans is approximately 6. Then the dimension of the index vector is assigned the value of 6, which makes the independent function path-based index to cover most of the existing services while retaining high reusability. The optimal values of L and K are calculated based on λ and γ, and the details can be found in [29]. Other parameters given in Table II are mainly empirically selected.

7.2. Evaluating the computation cost

To evaluate the computation cost of our proposed methods, we compared them against the generalized service search method, which is commonly used in existing works [5, 13, 14]. We implemented the generalized service search based on a functional unit-based search process, in which functional

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

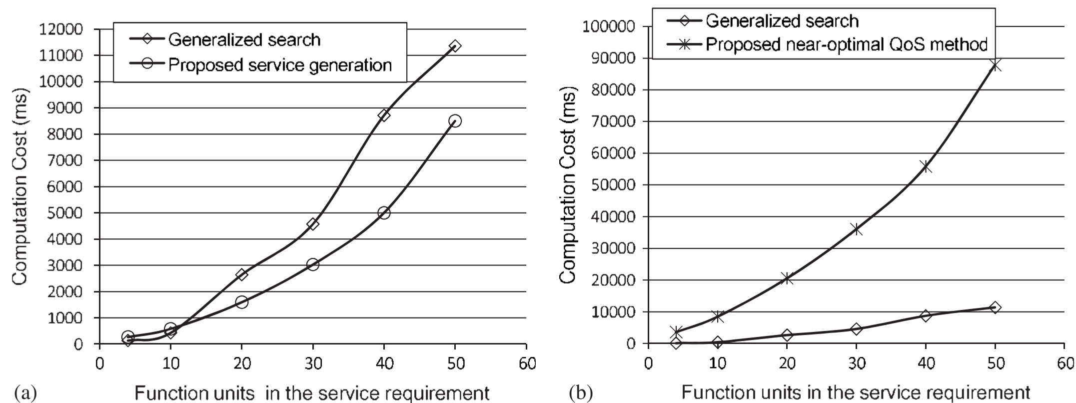

Figure 11. Evaluation results on computation cost.

units in the service requirement were used one by one to search for the reusable services for composition. Our proposed general service generation and the near-optimal QoS-based service generation methods were both evaluated. Based on 30 service requirements, the average experimental results are shown in Figure 11, in which the computation cost included two parts: service generation time cost and the time used to produce the composition plan scripts. The computation cost of the three methods all increase with the number of functional units contained in the service requirement. As shown in Figure 11, for the service requirement with a few functional units, computation costs of our proposed service generation process and the generalized service search are both small, and our method uses a little more time. When the functional unit number becomes larger than 10, our proposed service generation method always shows a better performance than the generalized search method (with average 40% advantage), which is mainly because of the two-level index strategy used in our approach. As exhaustive searching is employed, our proposed near-optimal QoS method usually results in worse performance than the generalized search method, as shown in Figure 11. Although the branch and bound algorithm is employed for searching branch pruning, the average computation cost is still more than 50 s to generate a near-optimal service for the service requirement containing 40 functional units. The near-optimal QoS service generation method is usually applicable for simple service requirements. Given a service requirement containing less than 10 functional units, the near-optimal QoS service can be generated in not more than 8 s, which is usually an acceptable computation cost.

7.3. Evaluating the success rate

In the iterative service generation process, service generation success rate is mainly decided by the service reuse scheme. To evaluate the success rate, we compared our proposed general service generation method with the service-level reuse method adopted in most of existing works [13–15]. In service-level reuse, all existing services can only be entirely reused in the black-box way. In the experiments, the atomic services and the composition plans in the service library were incrementally added to the functionality-based index for service generation. As 100 composition plans and 300 atomic services included in service library, they were added with the proportion of 1:3. Then, totally 50 service requirements were uniformly extracted from existing composition plans and

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

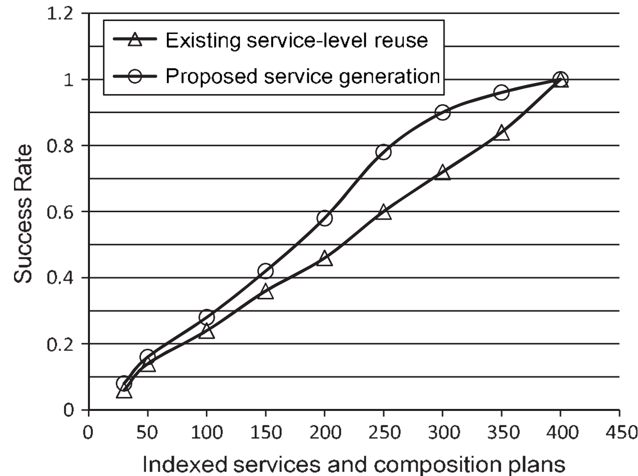

Figure 12. Evaluation results on success rate.

all achievable with the service library. In the experiments, a composition process was claimed successful if and only if the composition plan was generated, no matter whether it was published successfully. The average evaluation results are shown in Figure 12. We can see that the success rates noticeably increase with the number of indexed services. When the number of indexed services is small (<100), the success rates of service-level reuse and our proposed method are both very low (<0.25). In the process, our proposed service generation method always shows higher success rate. When the number of indexed services is between 150 and 350, our proposed service generation method produces the success rate averagely 15% higher than the service-level reuse method. Finally, the two success rates both reach the value around 1. The evaluation results demonstrate the superiority of our proposed method in service generation success rate.

7.4. Evaluating the QoS of generated services

The QoS evaluations for our proposed three methods were performed against the generalized search-based and service-level reuse scheme, which is the basic idea of most existing service reuse approach [13, 14]. In the experiments, to simplify the quality score calculation, all functional service requirements were built in sequential structure. As we have said, in the service library, the minimum time cost ($minq_1$) and the minimum price ($minq_2$) both equaled 2. In QoS threshold method, for a service requirement containing n functional units, the average time cost, i.e. $30n$, and the average price, i.e. $100n$, were empirically selected as the time cost and price threshold values, respectively. Similar to (7), for a generated service s with quality vector $q(s) = \langle q_{tc}(s), q_{pr}(s)\rangle$, the quality score $0.5\times(minq_1/q_{tc}(s)+minq_2/q_{pr}(s))$ is used as the quality metric. The experiments were performed based on service requirements containing 10, 50, and 100 functional units, for each of which, 10 service requirements were successfully processed. The average quality scores of generated services are shown in Figure 13. When the number of functional units contained in service requirements increases, the quality scores in all methods proportionally decrease. The generalized search and service-level reuse scheme always generates services having approximately the same quality scores with our general service generation method, and the QoS threshold-based method

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

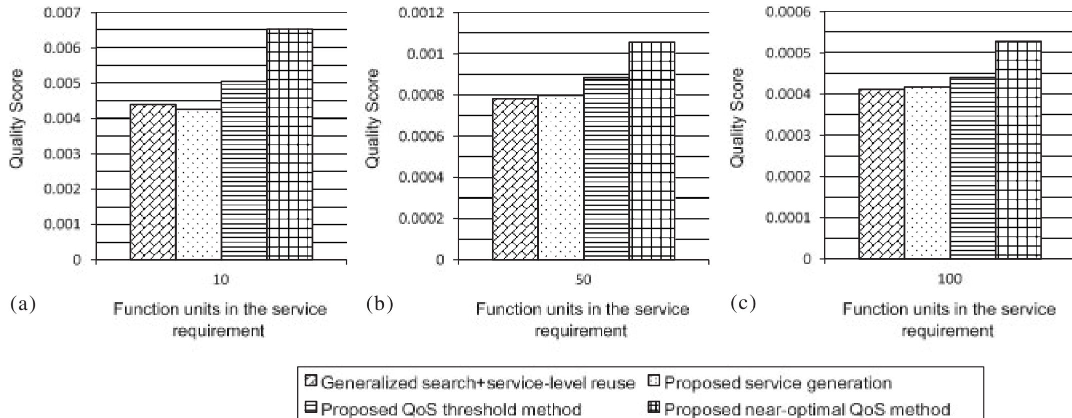

Figure 13. Evaluation results on QoS properties of generated services.

usually generates services with higher quality scores than the former two methods, averagely about 10% advantage. When the functional unit number becomes larger (i.e. equals 100), the former three methods produce services with similar quality scores. The near-optimal QoS method always produces services with highest quality scores, averagely about 20% higher than the QoS threshold method. The evaluation results demonstrate the effectiveness of our proposed QoS-aware methods.

8. CONCLUSIONS

In this paper, we addressed the service reuse issues in ubiquitous computing and proposed a new service reuse approach for service evolution process of the active services paradigm. With this approach, the user-needed service is evolved from the existing services having functionality overlapping with the service requirement, in which the reusable parts of existing services are extracted and directly utilized, and the missing functionalities are then further implemented. A functionality-based index was built to accelerate the service reuse process. QoS properties of ubiquitous services were also considered in the paper and two types of QoS-aware service generation methods have been introduced. Experimental results demonstrated the superiority of our methods in terms of computation cost, success rate, and the quality of generated services.

REFERENCES

1. Mark W. The computer for the 21st century. *Scientific American* 1991; **256**(3):94–104.
2. Zhang YX, Fang CH. *Active Services*: *Concepts*, *Architecture and Implementation*. Thomson Press: Singapore, 2005.
3. Zhang S, Goddard S. xSADL: an architecture description language to specify component-based systems. *International Conference on Information Technology*: *Coding and Computing*, Las Vegas, U.S.A., April 2005; 443–448.
4. Mei H, Chen F, Feng YD, Yang J. ABC: an architecture based component oriented approach to software development. *Journal of Software* 2003; **14**(14):721–732.
5. Morel B, Alexander P. SPARTACAS: automating component reuse and adaptation. *IEEE Transactions on Software Engineering* 2004; **30**(9):587–600.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

6. Penix J, Alexander P. Toward automated component adaptation. *International Conference on Software Engineering and Knowledge Engieering*, Madrid, Spain, June 1997; 535–542.
7. Zaremski MA, Wing JM. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(4):333–369.
8. Kinley A, Wilson D. Learning to improve case adaptation by introspective reasoning and CBR. *Case-based Reasoning*: *Experiences*, *Lessons*, *and Future Directions*. AAAI Press, MIT Press: New York, Cambridge, 1998.
9. Smyth B, Keane MT. Experiments on adaptation-guided retrieval in case-based design. *Technical Report TCD-CS-94-17*, Trinity College, Dublin, December 1994.
10. W3C Technical Reports and Publications Page. Available from: http://www.w3.org/TR/ [20 June 2008].
11. Milanovic N, Malek M. Current solutions for web service composition. *IEEE Internet Computing* 2004; **8**(6): 51–59.
12. BPEL Specification Page. http://www.ibm.com/developerworks/library/specification/ws-bpel/ [20 June 2008].
13. Shahram G, Craig A, Christos P *et al.* Proteus: a system for dynamically composing and intelligently executing web services. *International Conference on Web Service* (*ICWS*), Las Vegas, U.S.A., June 2003; 17–21.
14. Reyhan A, Hande Z. A graph-based web service composition technique using ontological information. *International Conference on Web Service* (*ICWS*), Salt Lake City, U.S.A., July 2007; 1154–1155.
15. Seog CO, Byung WO, Eric JL, Dong WL. BF*: Web services discovery and composition as graph search problem. *IEEE International Conference on e-Technology*, *e-Commerce and e-Service* (*EEE*), Hong Kong, China, March 2005; 784–786.
16. What is a composite service? Available from: http://www-128.ibm.com/developerworks/blogs/page /woolf?entry=composite_services [20 June 2008].
17. Zeng LZ, Boualem B, Anne HH, Marlon D *et al.* QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering* 2004; **30**(5):311–327.
18. Hamadi R, Benatallah B. A Petri net-based model for web service composition. *Australasian Database Conference*, Adelaide, Australian, February 2003; 191–200.
19. Menasc DA. Composing web services: a QoS view. *IEEE Internet Computing* 2004; **8**(6):88–90.
20. Corrmen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms* (2nd edn). MIT Press: Cambridge, U.K., 2002; 253–272.
21. Bentley JL. K-D trees for semi-dynamic point sets. *ACM Symposium on Computational Geometry*, Berkeley, U.S.A., June 1990; 187–197.
22. Yuan J, Duan LY, Tian Q, Xu C. Fast and robust short video clip search using an index structure. *ACM Multimedia Information Retrieval*, New York, U.S.A., October 2004; 61–68.
23. Guttman A. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD International Conference on Management of Data*, Boston, U.S.A., June 1984; 47–57.
24. Katayama N, Satoh S. The SR-tree: an index structure for high-dimensional nearest neighbor queries. *ACM SIGMOD International Conference on Management of Data*, AZ, U.S.A., May 1997; 369–380.
25. Lv Q, Josephson W, Wang Z, Charikar M, Li K. Multi-Probe LSH: efficient indexing for high-dimensional similarity search. *International Conference on Very Large Data Bases*, Vienna, Austria, September 2007; 950–961.
26. Weber R, Schek H, Blott S. A quantitative analysis and performance study for similarity search methods in high dimensional spaces. *International Conference on Very Large Data Bases*, Auckland, New Zealand, August 1998; 194–205.
27. Indyk P, Motwani R. Approximate nearest neighbors: towards removing the curse of dimensionality. *ACM Symposium on Theory of Computing*, Dallas, U.S.A., May 1998; 604–613.
28. Datar M, Immorlica N, Indyk P, Mirrokni VS. Locality-sensitive hashing scheme based on p-stable distributions. *ACM Symposium on Computational Geometry*, Hong kong, China, June 2000.
29. Cai R, Zhang C, Zhang L, Ma WY. Scalable music recommendation by search. *ACM Multimedia*, Augsburg, Germany, September 2007; 1065–1074.
30. Hwang CL, Yoon KS. *Multiple Attribute Decision Making*: *A State-of-the-Art Survey*. Springer: Berlin, Heidelberg, 1981.
31. Land AH, Doig AG. An automatic method for solving discrete programming problems. *Econometirca* 1960; **28**:497–520.
32. UDDI Online Community Page. Available from: http://uddi.xml.org/ [20 June 2008].
33. ActiveBPEL Introduction Page. Available from: http://www.activevos.com/community-open-source.php [20 June 2008].

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac

AUTHORS' BIOGRAPHIES

Peng-Wei Tian received his BS degree in Computer Science and Technology from Northeastern University, Shenyang, China in July 2004. From September 2004 to now, he has been a PhD student in Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests focus on software service reuse, service evolution, active service modeling, and QoS issues in the ubiquitous computing environment. He is the co-author of 10 technical papers and 4 Chinese patents.

Yao-Xue Zhang received his BS degree from Northwest Institute of Telecommunication Engineering, China, and received his PhD degree in computer networking from Tohoku University, Japan in 1989. Then, he joined Department of Computer Science, Tsinghua University, China. He was a visiting professor of Massachusetts Institute of Technology (MIT) and University of Aizu, in 1995 and 1998, respectively. Currently, he is a fellow of the Chinese Academy of Engineering and a professor in computer science and technology in Tsinghua University, China. Additionally, he serves as an editorial board member of four international journals. His major research areas include computer networking, operating systems, ubiquitous/pervasive computing, transparent computing, and active services. He has published over 170 technical papers in international journals and conferences, as well 8 monographs and textbooks.

Yue-Zhi Zhou obtained his PhD degree in Computer Science and Technology from Tsinghua University, China in 2004 and is now working as an associate professor at the same department. He worked as a visiting scientist at the Computer Science Department in Carnegie Mellon University in 2005. His research interests include ubiquitous/pervasive computing, distributed system, mobile device and systems. He has published over 30 technical papers in international journals or conferences. He received the IEEE Best Paper Award in the 21st IEEE AINA International Conference in 2007. He is a member of IEEE and ACM.

Laurence Tianruo Yang focuses on the research fields of high performance computing and networking, embedded systems, ubiquitous/pervasive computing and intelligence. He has published around 300 papers (including around 80 international journal papers such as IEEE and ACM Transactions) in refereed journals, conference proceedings, and book chapters in these areas. He has been involved in more than 100 conferences and workshops as a program/general/steering conference chair and more than 300 conference and workshops as a program committee member. He served as the vice-chair of IEEE Technical Committee of Supercomputing Applications (TCSA) until 2004; currently is the chair of IEEE Technical Committee of Scalable Computing (TCSC), the chair of IEEE Task force on Ubiquitous Computing and Intelligence. In addition, he is the editors-in-chief of eight international journals and few book series. He is also serving as an editor for around 20 international journals.

Copyright © 2009 John Wiley & Sons, Ltd.

Ming Zhong was born in Shaoxing, Zhejiang Province, People's Republic of China in 1983. He received the BS degree in Computer Science and Technology from Tsinghua University, Beijing, China in 2005. At present, he is a PhD candidate in Department of Computer Science and Technology in Tsinghua University. His research interests are in the area of program mining, component organizing, and searching, and he has been involved in various projects on CBSD.

Lin-Kai Weng was born in 1983. He has received his BS degree in Compute Science and Technology from Tsinghua University, Beijing, China in 2006. Then he became an PhD student in the same department and University. His main research interest focus on the personalized recommendation in the social community.

Li Wei received his BS degree in Computer Science and Technology from Tsinghua University, Beijing, China in 2003, and he is currently a PhD Candidate in Department of Computer Science and Technology of Tsinghua University, Beijing, China. His research interests include computing network, transparent computing, and virtual storage.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/dac