Information and Software Technology 53 (2011) 1370–1381

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A relaxable service selection algorithm for QoS-based web service composition

Chia-Feng Lin [a], Ruey-Kai Sheu [b,*], Yue-Shan Chang [c], Shyan-Ming Yuan [a]

[a] Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
[b] Department of Computer Science, Tunghai University, Taichung, Taiwan
[c] Department of Computer Science and Information Engineering, National Taipei University, Taipei, Taiwan

ARTICLE INFO

Article history:
Received 2 February 2010
Received in revised form 30 May 2011
Accepted 29 June 2011
Available online 13 July 2011

Keywords:
Web service
Quality of service
Service composition
Service selection

ABSTRACT

Context: Web services are emerging technologies that enable application to application communication and reuse of autonomous services over Web. Composition of web services is a concept of integrating individual web services to conduct complex business transactions based on functionality and performance constraints

Objective: To satisfy user requirements, technologies of Quality of service (QoS)-based web service composition (QWSC) are widely used to build complex applications by discovering the best-fit web services in term of QoS.

Method: In this paper, a QoS-based service selection (RQSS) algorithm is proposed to help composite web application development by discovering feasible web services based on functionalities and QoS criteria of user requirements. The RQSS recommends prospective service candidates to users by relaxing QoS constraints if no suitable or available web service could exactly fulfill user requirements.

Results: A generic framework is implemented to demonstrate the feasibility and performance of RQSS by adapting WS-BPEL standards, and can be reused for QoS-based web composition applications.

Conclusion: The experimental results show that the RQSS algorithm indeed performs well and increases the system availability and reliability.

1. Introduction

Web services are considered as self-contained, self-describing, modular applications that can be published, located, and invoked across the web [10]. An increasing amount of companies and organizations only implement their core business and outsource other services over Internet. If no single web service can satisfy the functionality of user needs, there should be a possibility to combine existing services together in order to fulfill the user requirements. If the implementation of a web service's business logic involves the invocation of other web services, it is necessary to combine the functionality of several web services. The process of developing such composite service is called service composition. Composite services are recursively defined as an aggregation of elementary and composite services.

With the increasing number of web services providing similar functionalities, the QoS is becoming an important criterion of selection of the best available service. There are two major problems in using QoS for service discovery, and they are the specification and storage of the QoS information, and the specification of the customer's requirements [31]. Some researchers used the linear integer programming (LIP) model to solve the service selection problem by maximizing the utility value which is a weighted sum of user-defined QoS attributes [4,11,32]. LIP-based approaches can be used for service matching, ranking and selection. However, the computation complexity and cost of the LIP solution will increase exponentially with the growth of the size of web services. Some studies tried to extend the multi-dimension multi-choice 0–1 knapsack problem (MMKP) with heuristic algorithms to get near-optimal solutions for service selection problems [8,21,31]. Although the heuristic algorithms improve the efficiency of service selection, they might fail if no feasible service exists. Recently, some researchers proposed the similarity-based web service discovery mechanisms by calculating the semantic similarity, and tried to maximize the number of candidate services to reduce the possibility of system failure [15,18]. However, the similarity-based solution might be not feasible for engineering or manufacturing industries because the accuracy of engineering application semantics should be very precise. In addition, the research in [32] was proposing a QoS-aware fault tolerant middleware. They aim at advancing the current state-of-the-art in fault tolerance technologies for dependable service composition.

Obviously, most of existing approaches were endeavoring to look for suitable services for composing the business logic of application. They either did not aim at the possible failure of service selection or did make too many constraints to find efficiently

* Corresponding author.
E-mail addresses: cflin@cs.nctu.edu.tw (C.-F. Lin), rickysheu@thu.edu.tw (R.-K. Sheu), ysc@mail.ntpu.edu.tw (Y.-S. Chang), smyuan@cs.nctu.edu.tw (S.-M. Yuan).

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.06.010

suitable services. Undoubtedly, stringent constraints might close to infeasibility solution. It is desirable to minimize the frequency of failure in finding a feasible solution, if one exists. Such problem can be viewed as a maximum feasible subsystem problem (*MaxFS*)[1] which is a NP-hard problem.[2] The *MaxFS* is that wishes to find a feasible subsystem containing a maximum number of inequalities. There are so far various heuristic approaches [36–39] have been proposed to find the approximation solution. Recently, some researches have shown that the relaxation method [36,39] is an admirable heuristic to approximate the solution of *MaxFS*.

Most of approaches to service selection do not handle the situation when there is no feasible solution to fulfill QoS constraints set by users. Also, some approaches may fail in finding the feasible solution if ones exist. To improve these problems, in the paper, we propose a relaxable QoS-based service selection algorithm (RQSS). The term of ''relaxable'' means that RQSS will release or relax the degree of constraints heuristically. The idea of proposed algorithm is to find a solution with smaller amount of constraints violation if there is no a feasible solution to fulfill the QoS constraints. Also, if no feasible can be found, a recommended solution is presented to the user by relaxing the QoS constraints. In other words, some QoS constraints have to be relaxed to find such a solution.

From the point of view of users, there may be some QoS constraints which cannot be relaxed. Therefore, users can specify each QoS constraint as either a relaxable constraints or a non-relaxable constraint. According to the relaxability of QoS constraints, the relaxable QoS constraints are relaxed to find a solution which can fulfill the non-relaxable QoS constraints if no feasible solution exists. Note that it is expected that the amount of relaxation of QoS constraints is as small as possible. Moreover, the proposed selection algorithm is a heuristic algorithm that is not an exhaustive search. We expect the algorithm can lower the rate of failure in finding a feasible solution when the heuristic is used.

We adopt the idea of a heuristic algorithm proposed in [17], which is to find the feasible solution for MMKP, to design the relaxable QoS-based service selection algorithm. In addition, a generic framework is proposed to simulate the feasibility and performance of RQSS. The results of the simulation experiments show that the idea of relaxing QoS criteria in RQSS performs well with lower failure rate especially when no web service could exactly match users' QoS constraints.

The remainder of the paper is organised as follows. Section 2 surveys some related work including web service standards, web service composition techniques, and other QoS constraint web service composition research; and briefly introduce the concept and notations used in the paper. Section 3 describes the model of proposed RQSS QoS web service composition, including basic definitions, constraints and functions. Section 4 depicts the design of relaxable QoS-based service selection algorithm, its complexity analysis, and QoS-based Web Service Composition Framework. Section 5 presents the evaluation setup, experiment results and analysis. In addition, we also make a comparison with other similar approaches and give a discussion. And finally, the conclusions and future work are given in Section 6.

2. Background

2.1. Related work

It is well-known that there are many industrial standards have been developed for service oriented computing and web service composition. The WS-Policy standard[3] can be used to specify policies expressing non-functional requirements for Web Services (WS), while WS-QoS is for specifying provided and required Web Services QoS [40]. And the Web Services flow specification languages like WS-BPEL[4] (Web service Business Process Execution Language) provides a programming-language like constructs as well as graph-based links that represent additional ordering constraints on the constructs, while the WSCI[5] is to describe the messages exchanging order between services. In addition, semantic annotations have been widely discussed in the Semantic Web community where preconditions and the effects of services are explicitly declared in the Resource Description Format (RDF).

Web service composition is a very complex and challenging task beyond human capability. Many researches tried to simplify the integration of web services by defining several standard interfaces, and proposed many models and approaches for static or dynamic service composition [22,23,26–28]. Dustdar et al. suggested six major issues that have a large impact on service composition and many researches took them into consideration seriously [6]. They are coordination, transaction, context, conversation modeling, execution monitoring, and infrastructure of web services. Besides, Dustdar et al. categorized five composition strategies which is used by most researches [6]. These approaches are static/dynamic mechanism, model driven, declarative, automated/manual, and context based web service discovery and composition.

Alonso et al. summarized six different dimensions of service composition models which make different assumptions of what a component is and what it is not [1]. For instance, an orchestration model defines abstractions and languages to define the order in which and the conditions under which web services are invoked. Most of these models focus on the matching, selection and execution of web services to meet functional requirements. As for non-functional constraints or Quality of Service (QoS) of a composite web service, such as latency, throughput, reliability, availability, cost, etc. are out of the major scopes of their researches.

Alrifai and Risse [33] proposed a solution that combined global optimization with local selection techniques to benefit from the advantages of both worlds. The proposed solution consists of two steps: first, they used mixed integer programming (MIP) to find the optimal decomposition of global QoS constraints into local constraints. Second, they used distributed local selection to find the best web services that satisfy these local constraints. But they do not take possible failure in finding a feasible solution into consideration.

Canfora et al. [34] proposed a QoS-aware composite service binding approach based on Genetic Algorithms (GAs). The main advantage in the use of GAs is the possibility to apply the approach in presence of arbitrary, non-linear QoS aggregation formulae, whereas traditional approaches, such as linear integer programming, require linearization. Obviously, they also do not take possible failure in finding a feasible solution into consideration.

Yu et al. [21] modeled web services selection with end-to-end QoS constraints in two ways. The first model is the combinatorial model that defines the problem as a multidimension multichoice knapsack problem (MMKP). The second model is the graph model that defines the problem as a multiconstrained optimal path (MCOP) problem. Based on both models, they specified a user-defined utility function of some system parameters to optimize application-specific objectives. In addition, Yu et al. also proposed heuristic algorithms to find near-optimal solutions in polynomial time which is more suitable for making runtime decisions. Therefore, they also do not take possible failure in finding a feasible solution into consideration.

[1] MaxFS problem: Given an infeasible linear system $AX \geqslant b$, find a Maximum Feasible Subsystem, i.e., a feasible subsystem containing a maximum number of inequalities.

[2] http://risorse.dei.polimi.it/maxfs/.

[3] http://www.w3.org/TR/ws-policy/.

[4] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[5] http://www.w3.org/TR/wsci/.

Pastrana et al. [35] presented a methodology for Web Service composition and coordination based on connectors which are defined by Web Service client and automatically generated by the COMPOSITOR tool authors have developed. Connectors use contracts to express the non-functional requirements and the behavior desired by the client of a service, such as QoS (Quality of Service) features. The work is mainly aiming at proposing a framework for QoS-enabled and self-adaptive connectors for Web Services composition and coordination. Adaption [5,25,35] is also an important feature for web service composition. But, they do not find out the feasible solution for composition.

Ko et al. [19] addressed web service composition planning from the aspect of QoS and proposed an efficient QoS-oriented web service composition algorithm. This algorithm is characterized as a hybrid meta-heuristics that combines tabu search and simulated annealing and designed not only to find constraint–compliant composition plans but also to reduce the computational burden required for searching the plans. The algorithm takes a composition schema, six QoS permissible values, and their weights as the inputs and generates a composition plan that satisfies the client's QoS requirement as the output. In addition, under a given composition plan, the algorithm creates a better neighbor plan by modifying a restricted number of decision variables. Obviously, it does not deal with finding feasible solution while possible failure.

Liu et al. [29] proposed a QoS-based dynamic web service composition approach. In the approach, they also model web service selection using multi-choice knapsack problem, and divide the service selection problems into two classes to facilitate the service selection process. The mathematical models are established for each class respectively. Accordingly, the heuristic selection algorithms are proposed to solve the models.

In summary, most of existing approaches were endeavoring to look for suitable services for composing the business logic of application. These approaches do not handle the situation when there is no feasible solution to fulfill QoS constraints. It is an important issue for finding a solution with smaller amount of constraints violation if there is no a feasible solution to fulfill the QoS constraints. Therefore, to get higher system reliability and availability, a relaxable QoS-based service selection algorithm is proposed in the paper.

2.2. Preliminary

Web service interactions can be described in two ways: executable business processes, and abstract business processes. Executable business processes model actual behavior of a participant in a business interaction. WS-BPEL is meant to be used to model the behavior of both executable and abstract processes. WS-BPEL provides a language for the specification of executable and abstract business processes [13]. Building composite web services are similar to the construction of a workflow for executable business processes [7]. For the emulation of RQSS, the WS-BPEL is used to describe the control flow of abstract processes. In order to clearly identify the concept, we first define the abstract process.

Definition 1 (*Abstract Process*). An abstract process *AP* can be represented by a DAG, so it can be defined as a tuple of (G_{AP}, AS_{AP}). That is $AP = (G_{AP}, AS_{AP})$, where

- $G_{AP} = (V, E)$ is a directed and acyclic graph, where $V = \{v_1, \ldots, v_n\}$ represents a set of activities. Each vertex in V is a tuple $v = (AT_v, A_v)$, where
  - AT_v is the activity type of the vertex where $AT_V \in$ {ProcessStart, ProcessEnd, IFStart, IFEnd, FlowStart, FlowEnd, LoopStart, AbstractService}.
  - A_v is the activity associated with the vertex.

$E \subset V \times V$ is a set of directed edges indicating the transition among two activities.

- $AS_{AP} = \{as_1, \ldots, as_n\}$ is for a set of abstract services involved in the abstract process where n is the number of the abstract services.

In an *AP*, there are five basic patterns of relationship are used to integrate individual services for business processes construction, and they are "sequence", "if", "flow", "while" and "repeatUntil". The five relations will be used in an abstract process to construct the fundamental control flow structures of business workflows, including "sequential structure", "parallel structure", "conditional structure" and "loop structure". All user requirements will be described by abstract services of an abstract process using the WS-BPEL. In addition, a *loopNumber* attribute is defined as the upper bound for the loop number of iterations. It determines the expected maximum number of iteration used to unfold the loop when calculating the QoS of loop structures. By using the basic notations of patterns, an abstract process specification can be represented by a directed acyclic graph (DAG).

An abstract process could be treated as a set of activities which consist of a start point, an end point, and other intermediate activities which are represented by the *ProcessStart*, *ProcessEnd* and other activity notations, respectively. The abstract process can orchestrate the set of abstract services in a precedence order. Accordingly, the sequences of activities are connected by the directed edges based on the order of them and conduct a DAG state diagram. The directed edges indicate the precedence relationship between two activities. Since "if", "flow", "repeatUntil" and "while" activities are all the non-loop structures, each of them can be represented by an entry point and an exit point. The *IFStart* notation is used to connect with the branches of possible conditions of an "if" activity, and an *IFEnd* notation is used for the convergence of the activity. For a "flow" activity, a *FlowStart* notation is used to connect each of the activities involved in it and these activities will be converge on a *FlowEnd* activity eventually. For a "repeatUntil" or a "while" activities, a *LoopStart* activity and a *LoopEnd* activity can be used to represent the activities which need to be repeated several times.

In addition, an abstract service describes the functionalities but not the execution details of a web service. The functionalities offered by a web service are usually performed by its provided operations. Thus, the definition of an abstract service should include the name, required operations, inputs, and outputs.

Definition 2 (*Abstract Service*). An abstract AS is a tuple of (N, OT, I, O), where

- N is the name of the abstract service.
- OT is the type of required operation in terms of ontology concepts.
- $I = \{I_1, \ldots, I_m\}$ is a set of inputs of required operations, where m is the number of inputs. $I_i = (IN, IT)$, where $i = 1, \ldots, m$. IN is the name of input. IT is the input type in terms of ontology concepts.
- $O = \{O_1, \ldots, O_n\}$ is a set of outputs of the required operations, where n is the number of outputs. $O_i = (ON, OT)$, where $i = 1, \ldots, n$, and ON is the name of output. OT is the output type in terms of ontology concepts.

The functional characteristics of an abstract service contain the inputs, outputs and the classification type representing an ontology concept. The operation type stands for the classification type of a web service operation. A WSDL example of an abstract service in the abstract process specification is shown in Table 1.

Table 1
A WSDL example of an abstract service.

```
<abstractService name="example" operationType="USWeatherForecast">
  < inputs>
    < input name="input" inputType="USAddress">
  </inputs>
  < outputs>
    < output name="output" outputType="FiveDayWeatherInformation">
  </outputs>
</abstractService>
```

USWeatherForecast, *USAddress*, and *FiveDayWeatherInformation* are operation types which will be used for ontology mapping or matching during the service selection process. The mapping and matching of ontology concepts are out of the scope of this paper.

Definition 3 (*Selection Plan*). A selection plan SP of an abstract process *AP* is a set of (as_i, op_i) pairs. That is $SP = \{(as_1, op_1), \ldots, (as_n, op_n)\}$. For each pair (as_i, op_i) in *SP*, the service operation op_i provides the required function of an abstract service as_i, where $i = 1, \ldots, n$.

The selection plan describes a combination of operations offered by concrete services for an abstract process. It can be viewed as a service assignment for the abstract process.

3. The RQSS QoS model

This section articulates the QoS model that is the basis of the RQSS algorithm. Here describe the QoS criteria, aggregation functions, and constraints are described.

3.1. QoS criteria

QoS criteria are used to differentiate the web services providing the same functionality during the service selection process of QWSC. Each operation of a web service will be associated with a set of QoS criteria. To reduce the complexity, the RQSS uses the generic QoS criteria as the basis for further discussions which are also used in [4,11,16].

- Execution Time *ET*(*op*): The execution time of the service operation *op* is a time interval between when the *op* is invoked and when the *op* finished.
- Reliability *Rel*(*op*): The reliability of the service operation *op* is the probability that a request is correctly responded within the expected time.
- Availability *A*(*op*): The availability of the service operation *op* is the probability that the operation is accessible during a period of time.
- Reputation *Rep*(*op*): The reputation of the service operation*op* is the measurement of trustworthiness. It is an integer range from 1 to 10. The higher the value, the better the reputation is.
- Price *P*(*op*): The price of the service operation*op* is the cost which a service requester has to pay to the service provider of *op*.

3.2. QoS aggregation functions

To calculate the quality of a web service, aggregation functions are defined in Table 2 for each QoS criterion. P_i is the probability that an abstract service as_i will be executed for a conditional structure. The probability can be evaluated from statistical execution logs. If an abstract process has never been executed before, the probability will be assumed to be uniform distribution ($p_i = 1/n$).

Table 2
Aggregation functions for QoS criteria.

	Sequence	Conditional	Parallel
Execution Time	$\sum_{i=1}^{n} ET(op^i)$	$\sum_{i=1}^{n} ET(op^i) \times p_i$	$\mathrm{Max}_{i=1}^{n} ET(op^i)$
Reliability	$\prod_{i=1}^{n} \mathrm{Rel}(op^i)$	$\sum_{i=1}^{n} \mathrm{Rel}(op^i) \times p_i$	$\prod_{i=1}^{n} \mathrm{Rel}(op^i)$
Availability	$\prod_{i=1}^{n} A(op^i)$	$\sum_{i=1}^{n} A(op^i) \times p_i$	$\prod_{i=1}^{n} A(op^i)$
Reputation	$\frac{1}{n}\sum_{i=1}^{n} \mathrm{Rep}(op^i)$	$\frac{1}{n}\sum_{i=1}^{n} \mathrm{Rep}(op^i)$	$\frac{1}{n}\sum_{i=1}^{n} \mathrm{Rep}(op^i)$
Price	$\sum_{i=1}^{n} P(op^i)$	$\sum_{i=1}^{n} P(op^i) \times p_i$	$\sum_{i=1}^{n} P(op^i)$

As defined above, an abstract process is a set of abstract services. The service selection would be the key for the construction of an abstract process from composite services.

3.3. QoS constraints

The QoS constraints represent the minimum thresholds which an abstract process has to reach to. For an abstract process, a QoS constraint can be identified by one of the QoS criteria which are "Execution Time", "Reliability", "Availability", "Reputation" and "Price". Besides, a relaxability of QoS constraints is introduced to specify whether a QoS constraint is relaxable or not. During the process of service selection, relaxable QoS constraints could be relaxed when no service can fulfill the requirements of QoS constraints.

Definition 4 (*QoS constraint*). A QoS constraint is a tuple *QC* = (*N*, *V*, *R*) where

- *N* is the name of the QoS criterion associated with *QC*.
- *V* is the value of the QoS criterion associated with *QC* desired to be satisfied. It is a real number.
- *R* is the relaxability of the QoS constraint which indicates whether the QoS constraint is relaxable or not. It is a Boolean value.

4. The relaxable QoS-based service selection algorithm

The service selection process of a QWSC takes abstract processes as inputs, and then comes out a set of qualified service operations supported by composite web services. That is, for an abstract process *AP* containing *n* abstract services, where $AS_{AP} = \{as_1, \ldots, as_n\}$, there will be a set of qualified service operation groups $OP = \{OP_1, \ldots, OP_n\}$ associated with each as_i, where.

- Each abstract service $as_i (i = 1, \ldots, n)$ has l_i candidates of service operations. $OP_i = \{op_{i1}, \ldots, op_{il_i}\}$ is a service operation group of as_i.
- Each candidate operation $op_{ij} (j = 1, \ldots, l_i) \in OP_i$ is associated with a QoS vector ($ET(op_{ij})$, $Rel(op_{ij})$, $A(op_{ij})$, $Rep(op_{ij})$, $P(op_{ij})$).
- The QoS constraints of the *AP* defined by users are $QC = \{qc^{ET}, qc^{Rel}, qc^{A}, qc^{Rep}, qc^{P}\}$.

Based on the above analysis, a service selection process could be formulated as Eq. (1).

$$
\begin{aligned}
&\text{Find } X = (x_1, \ldots, x_n), \quad \forall x_i \in \{1, \ldots, l_i\}, \ \ i = 1, \ldots, n \text{ such that} \\
&\mathrm{ET}(\{(as_1, op_{1x_1}), \ldots, (as_n, op_{nx_n})\}, AP) \leqslant qc^{ET}, \\
&\mathrm{Rel}(\{(as_1, op_{1x_1}), \ldots, (as_n, op_{nx_n})\}, AP) \geqslant qc^{Rel}, \\
&\mathrm{A}(\{(as_1, op_{1x_1}), \ldots, (as_n, op_{nx_n})\}, AP) \geqslant qc^{A}, \\
&\mathrm{Rep}(\{(as_1, op_{1x_1}), \ldots, (as_n, op_{nx_n})\}, AP) \geqslant qc^{Rep}, \\
&\mathrm{P}(\{(as_1, op_{1x_1}), \ldots, (as_n, op_{nx_n})\}, AP) \geqslant qc^{Rep},
\end{aligned}
\tag{1}
$$

X is a vector representing the feasible solution containing the indices of selected service operations for every abstract service in AP. It can be viewed as a selection plan of AP which can satisfy all the QoS constraints. It is expected to find a selection plan which can satisfy the QoS constraints based on a set of QoS criteria of an abstract process. Accordingly, Eq. (1) can be used to model the service selection problem for a QWSC solution. In the following section, the RQSS algorithm is proposed for solving the service selection problem for QWSC.

4.1. The design of RQSS algorithms

The original motivation of RQSS is to find out a feasible solution which minimizes the differences between selected web service operations with user-defined requirements when no web service exactly matching the QoS constraints. In other words, by adding user-defined relaxable QoS constraints to RQSS, some constraints could be relaxed to find alternative services if none of the services can match the QoS criteria. It is not users' intention to release the QoS constraints, so the QoS relaxing process can only be triggered when no feasible solution could be found. And, it is expected that the degrees of relaxation of QoS constraints are as small as better.

Similar to the algorithm of multi-dimension multi-choice 0–1 knapsack problem (MMKP) proposed in [17,21], the proposed algorithm assigns a value to every candidate item by evaluating the number of resource consumptions. In RQSS, the value of resource consumption is assigned to each item based on the degree of the constraint violation. Based on the value, the RQSS tries to find an initial solution by selecting an item from each group with the lowest value. If no feasible initial solution could be found, the one with the second lowest value will be selected from each group recursively. However, to increase the service availability or reduce the rate of failure, a new item is practically considered to be selected only if it has lower value and less number of violated constraints. It is one of the major purposes of the RQSS algorithm to balance the resource consumption and service availability.

To simplify the algorithm design, based on the Eq. (1), all of the QoS constraints inequalities are transformed into less than-or-equal inequalities before performing the RQSS algorithm. The values of reliability, availability and reputation of each service operation and the values of the constraints on these QoS criteria are set to the reciprocals. Hence, all of the values of QoS constraints would be the upper bounds for all QoS criteria. Let $QC^{nr} = \{qc_1, \ldots, qc_r\}$ be the set of non-relaxable QoS constraints set by the user. The details of the RQSS algorithm are shown in Tables 3–5.

There are three major steps in the RQSS algorithm. The first step of RQSS is to sort the service operations for each service operation group and service operation groups based on the normalized quality value (nqv). The normalized quality value is calculated for each service operation op_{ij} in the each service operation group OP_i using the following equation:

$$nqv_{ij} = \sqrt{\left(\frac{ET(op_{ij})}{qc^{ET}}\right)^2 + \left(\frac{Rel(op_{ij})}{qc^{Rel}}\right)^2 + \left(\frac{A(op_{ij})}{qc^{A}}\right)^2 + \left(\frac{Rep(op_{ij})}{qc^{Rep}}\right)^2 + \left(\frac{P(op_{ij})}{qc^{P}}\right)^2} \quad (2)$$

We use the normalized quality value as a heuristic to select service operations. For service operations op_{i1} and op_{i2}, the heuristic implies that the probability of service operation op_{i1} may be greater than the one of op_{i2} if $nqv_{i1} > nqv_{i2}$. Therefore, the candidate service operations in each service operation group OP_i are sorted based on the normalized quality value. All OP_i of abstract services are also sorted based on their lowest normalized quality values of the service operations.

Table 3
Relaxable QoS-based service selection algorithm.

Algorithm RQSS:
// n: the total number of abstract services of a abstract process
// l_i: the number of candidate service operations in the service operation group i
// nqv_{ij}: normalized quality value of candidate service operation j in the service
// operation group i
// X_{old}: old solution vector used to record the solution which satisfies all of QoS
// constraints in the step 3
// X^{nr}_{old}: old solution vector used to record the solution which satisfies non-relaxable
// QoS constraints in step 3
// QC^{nr}: the set of non-relaxable QoS constraints
1. Service operation sorting
   1.1. Calculate the normalized quality value for every candidate service operation $\forall i = 1, \ldots, n$, $j = 1, \ldots, l_i$, calculate nqv_{ij}.
   1.2. Sort the candidate service operations in non-descending order in each service group according to nqv_{ij}
   1.3. Sort the service operation groups in non-descending order according to the normalized quality values of the service operations with the lowest normalized quality value and go to step 2.
2. Naïve service selection
   2.1. Perform the NSS algorithm
   2.2. if no feasible solution can be found in the step 2.1 **then** go to step 3
3. Service selection with QoS constraints relaxation
   3.1. Perform the SS_QCR algorithm
   3.2. **if** no feasible solution could be found in step 3.1 **then if** $|QC^{nr}| = 0$ **then** return the solution X_{old} **else** return the solution X^{nr}_{old}.

Table 4
Naïve service selection algorithm.

Algorithm NSS:
// n: the total number of abstract services of an abstract process
// l_i: the number of candidate service operations in the candidate service
// operation group i
// *isFeasible*(X): a function which returns true if the solution X satisfies the
// constraints and false otherwise
// X: current solution vector
// *min_cso*: the minimum number of candidate service operations among all
// service operation groups
$X \leftarrow$ the service operation of the lowest normalized quality value from each group
if *isFeasible*(X) **then**
  A feasible solution is found and return X
endif
$j \leftarrow 2$
repeat
  for $i \leftarrow 1$ **to** n **do**
    Choose the service operation j from service operation group i
    Update solution X
    if *isFeasible*(X) **then**
      a feasible solution is found and return X
    endif
  next
  $j \leftarrow j + 1$
until j > *min_cso* // all the service operations in a group have been examined

In the second step, a naïve service selection (NSS) is performed to select the service operations with lower normalized quality values. As depicted in Table 4, the solution is initialized by selecting the service operation with the lowest normalized quality value from each group. Then, the service selection process is applied repeatedly to every operation group based on their order. Once all the operations are examined and selected, i.e. no feasible solution is found; the naïve service selection procedure stops, and RQSS goes to the third step.

In the final step of RQSS, a heuristic is proposed to find a feasible solution by the average violated quality value ($avqv$), as shown in Table 5. The $avgv$ represents the degree of the constraint violation

Table 5
Service Selection Algorithm with QoS Constraints Relaxation.

Algorithm SS_QCR:

// n: the total number of abstract services in AP
// l_i: the number of candidate service operations in the candidate service operation group i
// *isFeasible*(X): a function which returns true if the solution X satisfies the constraints and false otherwise
// *select*(X,i,j): a function used to switch the selection of service from
// operation group i to service
// operation $j$based on X and update the value of solution X
// X_{old}: old solution vector, V_{old}: the set of QoS constraints that is
// violated by X_{old}
// X^{nr}_{old}: old solution vector used to record the solution which satisfies
// non-relaxable QoS constraints
// V^{nr}_{old}: non-relaxable QoS constraints which are violated by X_{old}
// $avqv_{old}$: average violated quality value of X_{old}
// $avqv^{nr}_{old}$: average violated quality value of X^{nr}_{old} for non-relaxable QoS constraints
// X_{new}: new solution vector, V_{new}: the set of QoS constraints which are
// violated byX_{new}
// X^{nr}_{new}: new solution vector used to record the solution which
// stratifies non-relaxable
// QoS constraints
// V^{nr}_{new}: non-relaxable QoS constraints which are violated by X_{new}
// $avqv_{new}$: average violated quality value of X_{new}
// $avqv^{nr}_{new}$: average violated quality value of X^{nr}_{new} for non-relaxable
// QoS constraints
// $avqv^{r}_{old}$: average violated quality value of X^{nr}_{old} for relaxable QoS constraints
// $avqv^{r}_{new}$: average violated quality value of X^{nr}_{new} for relaxable QoS constraints
// $iteration_{\max}$: maximum number of iterations
Initialize X_{old}, X_{new}, X^{nr}_{old} and X^{nr}_{new} by selecting the service operation with the lowest normalized quality value from each group
Identify V_{old} and V^{nr}_{old}, and then compute $avqv_{old}$ and $avqv^{nr}_{old}$
$avqv^{r}_{old} \leftarrow avqv_{old} - avqv^{nr}_{old}$
$iteration \leftarrow 1$
$stop \leftarrow$ false
repeat
stop ← true// check whether the $avqv$ and $avqv^{nr}$ are reduced in an iteration
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** l_i **do**
    cur_sel ← current selection of service operation group i based on X_{new}
    if $j \neq$ cur_sel **then**
      select(X_{new},i,j)
      if *isFeasible*(X_{new}) **then**
        Substitute X_{old} with X_{new} and return the feasible solution X_{old}
      endif
      Identify V_{new} and compute$avqv_{new}$
      if $avqv_{new} < avqv_{old}$ **then**
        Substitute X_{old} and $avqv_{old}$with X_{new} and $avqv_{new}$
        $stop \leftarrow$ false
      endif
      cur_sel ← current selection of service operation group i based on X^{nr}_{new}
      endif
      if $j \neq$ cur_sel **and** $|QC^{nr}| \neq 0$ **then**
        select(X^{nr}_{new}, i, j)
        Identify V^{nr}_{new} and $QC^{nr} - V^{nr}_{new}$, and then compute $avqv^{nr}_{new}$ and $avqv^{r}_{new}$
        if $avqv^{nr}_{new} < avqv^{nr}_{old}$ **then**
          Substitute X^{nr}_{old}, $avqv^{nr}_{old}$ and $avqv^{r}_{\text{old}}$ with X^{nr}_{new}, $avqv^{nr}_{new}$ and$avqv^{r}_{new}$
          $stop \leftarrow$ false
        endif
        if $avqv^{nr}_{new} = 0$ **and** $avqv^{r}_{\text{new}} < avqv^{r}_{old}$ **then**
          Substitute X^{nr}_{old}, $avqv^{nr}_{old}$ and $avqv^{r}_{\text{old}}$ with X^{nr}_{new}, $avqv^{nr}_{new}$ and $avqv^{r}_{new}$
          $stop \leftarrow$ false
        endif
      endif
    next
  next
 $iteration \leftarrow iteration + 1$
until $iteration > iteration_{\max}$ **or** $stop$

of a solution. The reduction of the average violated quality value for all QoS constraints is applied here to find a feasible solution or a solution that satisfies non-relaxable constraints if no feasible one could be found, respectively. Two solutions, X_{old} and X^{nr}_{old}, will be generated to record the solution fulfilling all of QoS constraints and the one of satisfying non-relaxable QoS constraints, respectively. Similar to the NSS algorithm, in this step, a solution is also initialized by selecting the service operation with the lowest normalized quality value from each group. It then goes throughout all the groups to evaluate either a feasible solution exists or $avqv$ of X_{old} and $avqv^{nr}$ of X^{nr}_{old} can be reduced. If no feasible solution exists, it switches to next group and performs the same examination process recursively. The average violated quality value of X_{old} is calculated using Eq. (3),

$$avqv = \frac{1}{\lambda}\sum_{i=1}^{\lambda}\frac{Q_i(\{(as_1, op_{1x_1}), \ldots, (as_n, op_{nx_n})\}, AP)}{qc_i} \tag{3}$$

where λ represents the total number of the QoS constraints violated, i represents the indexes of the QoS constraints violated by X_{old} and Q_i is the QoS aggregation function of the corresponding QoS criterion.

The average violated quality value of X_{old} is denoted as avqv_{old}. In this step, the average violated quality values are used as a heuristic to select service operations. Given that there are two non-feasible solutions, and they are X_1 and X_2. The concept of the heuristic is: if $avqv_1 > avqv_2$, the probability that X_1 becomes a feasible solution may be greater than the one for X_2. That is, it would be better to create a new feasible solution by choosing the one of lower average violated quality value. After identifying X_{old}, V_{old} and calculating the average violated quality value avqv_{old}, we can create selection plans to find a feasible solution. We denote the new solution as X_{new}. If $avqv_{new} < avqv_{old}$, then set $X_{old} = X_{new}$, $avqv_{old} = avqv_{new}$. In the first service operation group, the above expression will be performed for all service operations, and the selected service operation will be changed to the one of the lowest normalized quality value eventually. At the same time, the idea can be also applied to find the solution which satisfies the non-relaxable QoS constraints. There is also an initial solution X^{nr}_{old} which is set equal to X_{old}. The only difference is that the average violated quality value should be calculated based on non-relaxable QoS constraints and relaxable QoS constraints. The average violated quality value of X^{nr}_{old} for non-relaxable QoS constraints and relaxable QoS constraints are calculated using the following formulas:

$$avqv^{nr} = \frac{1}{\tau}\sum_{i=1}^{\tau}\frac{Q_i(op_{1x_1}, \ldots, op_{nx_n}, AP)}{qc_i} \tag{4}$$

$$avqv^{r} = \frac{1}{|QC| - \tau}\sum_{\text{j}=1}^{|QC|-\tau}\frac{Q_{\text{j}}(op_{1x_1}, \ldots, op_{nx_n}, AP)}{qc_{\text{j}}} \tag{5}$$

where τ is the number of the non-relaxable QoS constraints violated, i represents the indexes of the non-relaxable QoS constraints that are violated by X^{nr}_{old} and j represents the indexes of the relaxable QoS constraints that are violated by X^{nr}_{old}.

As mentioned above, if there is no feasible solution, it is expected to find a solution to fulfill the non-relaxable QoS constraints by relaxing the relaxable QoS constraints. The average violated quality value for non-relaxable QoS constraints is used to match the idea. The proposed algorithm could find a solution that can fulfill the non-relaxable QoS constraints by reducing the average violated quality value during the selection process. Moreover, if a solution satisfying the non-relaxable QoS constraints is found, an additional examination will be performed to check whether the

$avqv^r$ can be reduced or not. The purpose of the additional examination is used to minimize the total number of the relaxations for finding solutions for the non-relaxable QoS constraints.

If a solution that satisfies the non-relaxable QoS constraints is found, an extra examination for minimizing the amount of the relaxation is performed to check whether $avqv^r$ can be reduced or not. The final step is an iterative process performed when all service operation groups are evaluated. The process will stop at the maximum iteration or no more reduction of $avqv_{old}$ and $avqv^{nr}_{old}$ happened after some iterative runs. If no feasible solution could be found in this step (X_{old} is infeasible), the final solution X^{nr}_{old} will be recommended. In the step 3.2, if the $|QC^{nr}| = 0$, it represents no QoS constraints are involved in the service selection algorithm. We can simply return X_{old} to the caller procedure, otherwise the RQSS will return X^{nr}_{old} to the caller procedure. Obviously, the final solution can be considered as a better solution than others because that not only does it satisfy all non-relaxable QoS constraints but it also has the lower average violated quality value for relaxable QoS constraints.

4.2. Complexity analysis of RQSS

To simplify the complexity analysis, it is assumed that each abstract service has s candidate service operations. Let n be the total number of abstract services in an abstract process, m be the number of QoS criteria and q^{nr} be the number of non-relaxable QoS constraints. For the first step of RQSS, the complexity of computing the normalized quality values of all candidate service operations is $O(s.n.m)$. The complexity of sorting service operations of every service operation group is $O(s.n.\log s)$. The complexity of performing the sorting of service operation groups is $O(n.\log n)$. Consequently, the computational complexity of the first step of RQSS is $O(s.n.\log s)$. Similar to the first step, the computational complexity of the second step of RQSS is also $O(s.n.m)$. For the third step, the complexity of calculating the average violated quality value is $O(m.n)$. Finally, there are $n.(s-1)$ candidate service operations to be explored in one iteration to evaluate whether a new solution exists or not. If the maximum number of iterations is l, which is used in the SS_QCR in the Table 5, the total complexity of the third step is $O(l.m.n^2.s)$. According to above analysis, the complexity of the RQSS algorithm is $O(l.m.n^2.s)$.

4.3. The QoS-based web service composition framework

For the reliability and availability simulation, the test data and algorithms are implemented based on the framework shown in Fig. 1. An abstract process is used to describe the functional requirement of a process of web services at an abstract level. The functionality of required service of an abstract process is represented by an abstract service. The abstract services and the service advertisements published in the service registry are described using ontology. The available and feasible services are then discovered effectively by a semantic search [9,12,24]. Moreover, based on a set of QoS criteria, QoS constraints can also be specified to represent the non-functional properties of the whole abstract process to be satisfied. According to abstract process and the QoS constraints, selection algorithms can select a set of operations offered by concrete web services, which provide the required functionalities fulfilling the QoS constraints. If no feasible solution can be identified by the selection algorithm, the framework will recommend alternative solutions based on user pre-defined relaxable constraints. It is expected that the recommended solution for service selection has small amount of relaxation for relaxable QoS constraints.

As shown in Fig. 1, users can use the *GUI Console* (GC) to build the abstract processes and set the QoS constraints according to their functional requirement and non-functional requirement, respectively. To reduce the development effort, users can also reuse the existing abstract processes loaded from *Abstract Process Repository* (APR). The *Service Selector* (SS) is responsible for selecting web services according to the abstract process specification and QoS constraints forwarded from GC. During the service selection process, *Service Broker*searches for candidate services from *Service Registry* (SR) which is an extension to UDDI registry which record the published service information as well as the QoS information [2]. These services published in SR are described by ontology stored in the *Ontology Repository*. When a feasible solution is found in APR, *Feasible Solution Manager* stores this feasible solution into *Feasible Solution Repository* (FSR). Users can reuse existing

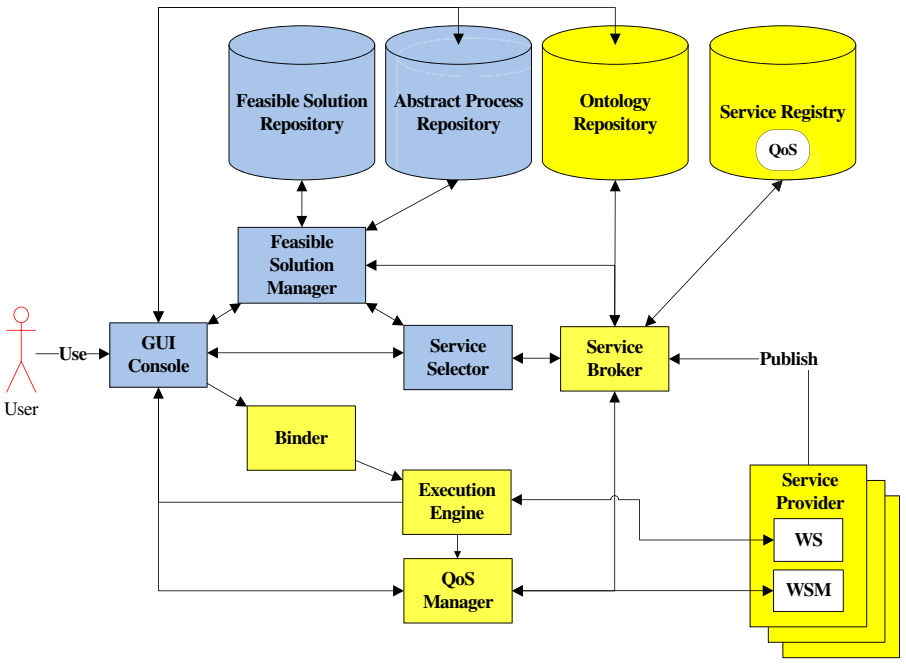

Fig. 1. A generic framework for generic QoS-based web service composition.

abstract processes from the FSR easily. After the feasible solution is identified, users can then composite the selected services by define the flow of input and output data binding to each services. Based on WSBPEL [13], the *Binder* generates the executable process specification according to the abstract process and binding information. The executable process specification is then forwarded to the *Execution Engine* to construct and execute the process instance. Furthermore, a*QoS Manager* (QM) is responsible for managing and monitoring QoS information of web services published in service registry [3]. QM collaborates with *WSM* [14,20] which is a web service management process to collect QoS information such as of the invoked service.

5. Experimental analysis

5.1. The evaluation setup

In this section, we compare the performance of proposed RQSS, WFlow [21] and RWSCS_KP [8]. WFlow and RWSCS_KP that are both well-known heuristic algorithms for QWSC only consider finding a feasible solution. They did not take the rate of failure in finding a feasible solution. We try to analyze the rate of failure in finding a feasible solution for QWSC and the average violated quality value if no feasible solution is found for these algorithms.

Especially, in the WFlow [21] approach, Yu et al. divided the service selection problem into two categories. One is to composite services with a sequential flow structure; the other is to composite services with a general flow structure. In the latter, it may contain complex structure between function nodes, such as loop. In order to simplify the problem, the WFlow removed the loop operations by unfolding the cycle. Based on the categorization, WFlow considering different objective function proposed two distinct algorithms; there are WFlow_EU and WFlow_HP respectively. In the experiments, we compared our approach only with the WFlow_EU (short for WFlow).

We utilized Java programming language to implement these algorithms and ran them on an Intel Pentium (R) D 3.4 Ghz, 2 GB RAM desktop PC with 100 MB/s Ethernet card, Window XP and JDK 6.0. In the simulation, test pattern generation is the same as what is introduced in [30]. The value of each service is not directly proportional to the QoS consumption. The random function of integer i gives an integer from 0 to i, following the uniform distribution. To generate the test instances, we first randomly generate the abstract processes, each containing two or more control flow patterns. For simplicity, we assume that each abstract service contains the same number of candidate service operations. For each candidate of abstract service, five QoS parameters ($q1$, $q2$, $q3$, $q4$, $q5$): execution time, reliability, availability, reputation and price are considered. Each quality value is randomly generated with a uniform distribution in a range, as shown in Table 6.

These parameters are exploited in some researches [32,35] and refer them to set the values. In the simulation, we assume that all services can be completed within 100 ms. And most of services are robustness enough. Therefore the reliability and availability of services are ranging from 0.95 to 0.99999. We think that is reasonable for most of web services. In addition, the reputation is between [1,10] according to how popular of the service is and the price is assumed that is between [1] based on the access cost of the service. These assumptions are all reasonable in various applications.

Moreover, the QoS constraint on each QoS criteria is generated using the Eq. (6). The concept of generating the QoS constraints refers to the test pattern generation for QWSC presented in [30]. The constraint factor is defined to increase the difficulty to find a feasible solution.

$$QC_i = n * Q_i^{\max} * CF, i = 1, \ldots, 5 \quad (6)$$

Table 6
The range of the value for each QoS parameter.

QoS parameter	Range
Execution Time	[1, 100]
Reliability	[0.95, 0.99999]
Availability	[0.95, 0.99999]
Reputation	[1, 10]
Price	[1, 100]

where n is the number of abstract services in an abstract process, $Q_i^{\max}$ is the maximum value of a quality parameter Qi of a service and CF is the constraint factor which is used to adjust the strength of a QoS constraint.

The constraint factor CF is a real number between $[0,1]$. It can be used to represent the strength of a QoS constraint. The lower the constraint factor value is, the higher the strength of the QoS constraint is. When CF = 1, there is no constraint on QoS requirement. CF = 0.5 means there is 0.5 probability that a random generated service in a group satisfies the average value. The default value of CF in this paper is 0.5. As shown in Table 7, the generated constraints are the upper bounds for the QoS constraints. In the generation of the QoS constraints, all the QoS metrics are assumed as the additive metrics. Since the reliability and availability are multiplicative metric, they can be transformed into additive parameters using logarithmic function. It should be noted that RQSS algorithm does not transform the reliability and availability into additive parameters. Therefore, the generated constraints on reliability and availability need to be transformed back into multiplicative metrics, i.e. $\left(\frac{1}{0.95}\right)^{n*CF}$.

5.2. Experimental results

WFlow [21] and RWSCS_KP [8] are implemented and compared with RQSS here because both of them are well-known heuristic algorithms for QWSC. Only the parts of finding a feasible solution in WFlow and RWSCS_KP are considered for justice.

From the proposed approach and algorithm mentioned in Section 4, there are three aspects related to the will be illustrated in the experiments, as follows:

1. The failure rate is reduced in finding a feasible solution while the abstract service increasing.
2. The strength of QoS constraints is increasing while the constraint factor value is decreasing.
3. The lower the *avgv* value is, the better the performance of the algorithm is.

5.2.1. Comparison of failure rate in finding a feasible solution

Fig. 2 shows the experiment results of examining impact factors for RQSS, WFlow and RWSCS_KP respectively. The analysis of the impact focuses on the variances of the number of abstract services in an abstract process, the number of candidate service operations and the strength of the QoS constraints (constraint factor). Since the number of abstract services will affect the failure of finding a feasible solution. The more the abstract services are, the lower

Table 7
The generation of the QoS constraints.

QoS parameter	QoS constraint generation
Execution Time	$n * 100 * CF$
Reliability	$n * \log\left(\frac{1}{0.95}\right) * CF$
Availability	$n * \log\left(\frac{1}{0.95}\right) * CF$
Reputation	$n*10*CF$
Price	$n*100*CF$

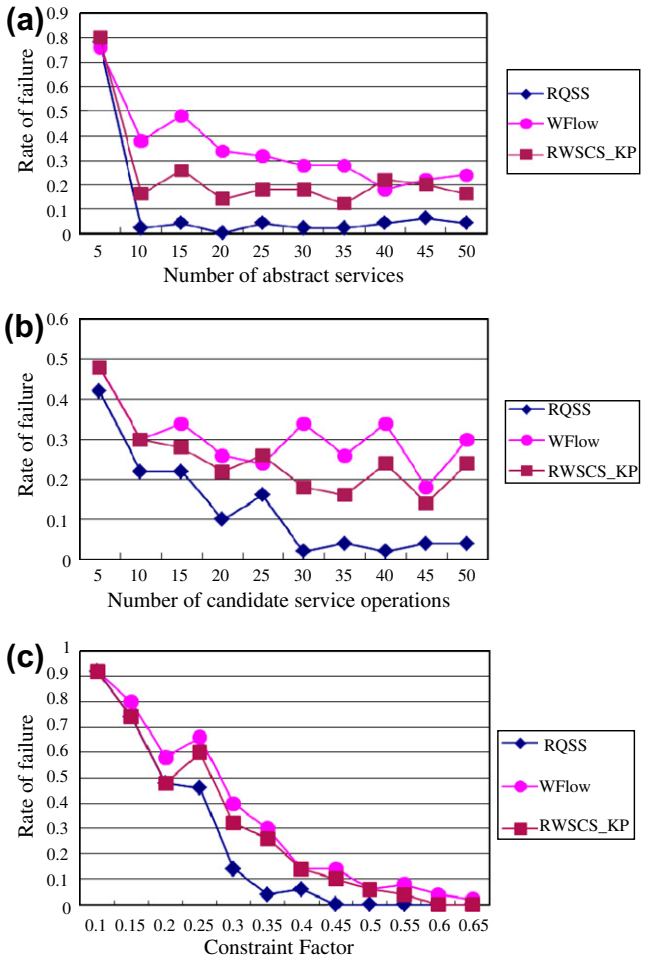

Fig. 2. Comparison of failure rate.

failure rate in finding a feasible solution is. Similarly, the number of candidate service operations is also affect the failure rate. In the simulation, we vary the two factors to analyze failure rate in finding a feasible solution. In addition, since the constraint factor is used to increase the difficulty to find a feasible solution, in the simulation, we also analyze the influence of constraint factor regarding to failure rate in finding feasible solution.

5.2.1.1. Impact of the number of abstract services. In the simulation, the numbers of abstract services are set from 5 to 50 to analyze the influence over the three algorithms. There are 50 test cases implemented and examined per number of abstract services. For each test case, there are 50 candidate service operations for each abstract service and the constraint factor is fixed to 0.4. As depicted in Fig. 2a, RQSS has lower failure rate in finding a feasible solution than the other two algorithms for almost of the test cases. When the number of abstract service is equal to 5, all algorithms have unacceptable high failure rates to find feasible solutions. The reason is that the number of abstract services is so few that the constraints relatively become very severe. When the number of abstract services is more than twenty, the failure rates of all algorithms behave stably. It can be said that the effects are few on the failure rate when the number of abstract services is more than a constant value.

5.2.1.2. Impact of the number of candidate service operations. Similar to the previous analysis, the numbers of candidate service operations aer also set from 5 to 50, and the number of abstract services and constraint factor are fixed as 50 and 0.4, respectively. As depicted in Fig. 2b, the failure rates are higher along with the decreasing number of candidate service operations. It is natural that the more choices the less failures will happen. And, the constraints are relatively severer when the number of candidate service operations decreases.

5.2.1.3. Impact of constraint factor. To measure the influence of constraint factor, the values are set from 0.1 to 0.65 and 50 test cases are generated for each. There are 50 abstract services and 50 candidate service operations for each abstract service. Fig. 2c shows that when the constraint factor is less than 0.1, no feasible solution will be discovered for all algorithms. When the constraint factor is greater than 0.65, every algorithm can find feasible solution.

5.2.2. Comparison of average violated quality value

The average violated quality value (*avqv*) represents the degree of the constraint violation of a solution. To evaluate the performance of our proposed QoS constraint relaxation technique, we compare the average violated quality value of RQSS with WFlow and RWSCS_KP. We only consider the situations where all the algorithms fail in finding a feasible solution and use the final solutions found in these algorithms to calculate the average violated quality values. As the previous analysis, we analyze the impact of varying the number of abstract services in the abstract process, varying the number of candidate service operations and varying the constraint factor. The numbers of non-relaxable QoS constraints for RQSS are set from 1 to 3 respectively for each test case. But, the items of non-relaxable QoS constraints are randomly selected. Note that the value of *avqv* is greater than 1 in all cases and the lower the value is, the better the performance of the algorithm is. The factors for analyzing the impact are the same as previous subsection.

5.2.2.1. Impact of the number of abstract services. Again, the numbers of abstract services are set from 5 to 50 and there are 50 test cases for each. The number of candidate service operations for every abstract service is set as 50 and the constraint factor is fixed as 0.4. Fig. 3a shows that RQSS performs better than the other two

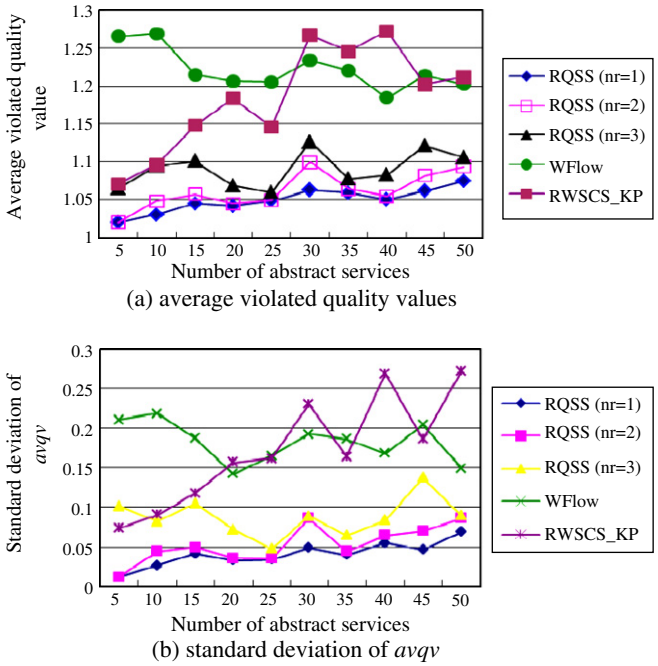

Fig. 3. Impact of the number of abstract services for *avqv*.

algorithms for every test case. Fig. 3b shows the standard deviation of *avqv* for 50 test cases per number of abstract services. All of standard deviations of *avqv* in RQSS are less than 0.15. In addition, the behaviors of both WFLow and RWSCS_KP are too unstable to be accepted. It is because the objective of the two algorithms is to find a feasible solution but not suggest alternative solutions when no feasible one can be discovered. This result highlights the importance of relax of QoS constraints.

5.2.2.2. Impact of the number of candidate service operations. The variances of the number of candidate service operations are set from 5 to 50. The number of abstract services is fixed as 50 and the constraint factor is set as 0.4. As depicted in Fig. 4a, RQSS has lower rate of failure in finding a feasible solution than other two algorithms for all of the tested number of candidate service operations. Obviously, the number of candidate service operations or the number of the non-relaxable QoS constraints increase, the *avqv* value of RQSS decreases. The more the number of candidates, the higher the possibility of finding a feasible solution is. It is because that the constraints are severer along with the decreasing number of candidate service operations and the relaxable QoS constraints are fewer. Thus, the number of constraints to be relaxed will substantially affect the *avqv* value. Fig. 4b shows that all of standard deviations of *avqv* in RQSS are less than 0.3. Similar to the analysis of impact of the number of abstract services, WFLow and RWSCS_KP also have more unstable *avqv*.

5.2.2.3. Impact of constraint factor. The constraint factors are from 0.1 to 0.45 to analyze the effect for each algorithm. Fifty test cases are generated for each constraint factor. There are 50 abstract services and 50 candidate service operations for each case. Fig. 5a shows that RQSS performs better than the other two algorithms when the constraint factor is greater than 0.2. RWSCS_KP can even perform better than RQSS when the constraint factor and the number of non-relaxable QoS constraints are small. It is in the situation when both the values of constraint factor and number of relaxable constraints are small enough. RQSS has to substantially release relaxable QoS constraints causing the high *avqv* value. Note that when the constraint factor is below 0.1, the constraint is too severe to find the solution that satisfies non-relaxable QoS constraints.

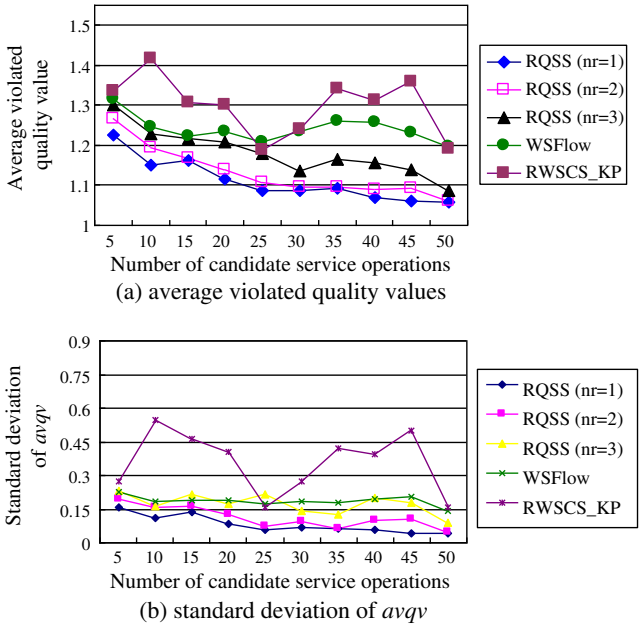

Fig. 4. Impact of the number of candidate service operations for *avqv*.

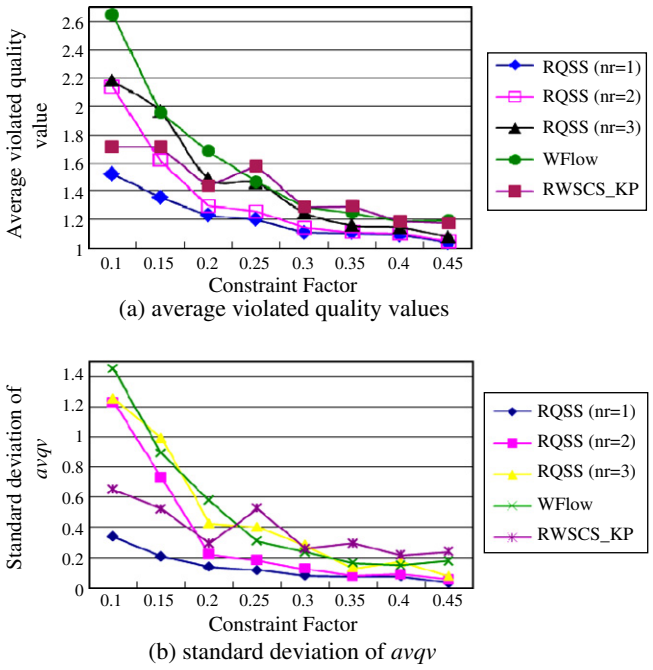

Fig. 5. Impact of constraint factor for *avqv*.

When the constraint factor is greater than 0.45, there should exists at least one feasible solution in RQSS. Therefore, we exclude these situations.

Fig. 5b shows that when the constraint factor is greater than 0.2, the standard deviation of *avqv* is less than 0.4. When the constraint factor is less than 0.2 and the numbers of non-relaxable QoS constraints are 2 and 3, the standard deviation of *avqv* is larger. The reason is also that the constraints are much severer in some test cases. RQSS has to substantially relax relaxable QoS constraints and thus the *avqv* of those test cases are much higher.

5.3. Discussion

From the experimental results, we conclude that, since the RQSS adopt heuristic approach to design the relaxable QoS-based service selection algorithm, it leads to significantly better QoS of composite service executions with lower the rate of failure in finding a feasible solution. Even if there is no a feasible solution to fulfill the QoS constraints, the RQSS can also find a solution with smaller amount of constraints violation. Table 8 shows the comparison with the WFlow and RWSCS_KP in terms of time complexity, failure rate in finding feasible solution, availability, and adaptation respectively. Even if the time complexity of the RQSS is similar to the others due to adopting similar heuristic approach, the

Table 8
Comparison with existing methods.

	WFlow	RWSCS_KP	Proposed RQSS
Time complexity	$O(n^2(s-1)^2m)$.	$O(n^2(s-l)^2)m)$	$O(l.m.n^2.s)$
Failure rate in finding a feasible solution	Middle	Middle	Low
Availability	Middle	Middle	High
Adaption	Good	Good	Good

n is the total number of abstract services in an abstract process.
m is the number of QoS criteria.
l is the maximum number of iterations.
s is the number of candidate service operations in an abstract service.

complexity of RQSS is possibly superior to the WFlow and RWSCS_KP while the *s* is far greater than *l*. It will be possible when the number of alternative web service increasing.

In addition, the failure rate and the availability can be proved that the RQSS is superior to the others from the simulation result in previous subsection. It is the main contribution of the proposed algorithm. The other factor in the comparison is the adaptation. Self-adapting [35] is an important feature of dynamic software architecture, especially for large scale web service composition. Systems which can monitor and adapt to the changes in their environment are known as self-adaptive, self-healing, or self-managing systems. Since WFlow, RWSCS_KP, and RQSS are all adopting heuristic approach to find feasible solution in each composition of web service, they can be adaptive to the changes in their environment as well as possible. Obviously, the only shortcoming is that the RQSS needs the user intervention to specify which QoS constraints are relaxable before the service selection.

In summary, we have stated that the heuristic algorithm is similar to the MMKP [17,21] in Section 4.1, which have shown that their algorithms are near-optimal solution for web service composition. In addition, based on the comparison, it is obvious that the RQSS is also a near-optimal solution for web service composition with lower failure rate and high availability.

6. Conclusions and future work

It is not easy to implement a development environment for users to reduce the complexity application building based on the web service composition technology. Most researches or studies do not deal with the situation of no suitable or feasible solution can be found during the service composition process. To get higher system reliability and availability, a relaxable QoS-based service selection algorithm, RQSS, is proposed in this paper. Not only does the RQSS help users to discover web services fulfilling the functional requirements and non-functional QoS constraints, but also it recommends solutions which could satisfy the non-relaxable QoS constraints by relaxing the relaxable QoS constraints. Besides, a generic framework is also designed for the evaluation of system performance for service selection algorithms. The experiment results reveal that the failure rate of finding a feasible solution in RQSS is much lower than those of WFlow and RWSCS_KP approaches. RQSS also has the lower average violated values for almost of the test cases, especially when no feasible solutions could be discovered. That is, RQSS performs well not only because of the low computation complexity of itself but also the idea of relax of QoS constraints.

In the next step, we plan to complete the framework for particular domain applications. We will also try to extend the RQSS algorithm for the case of contingencies and dynamic service composition which are currently out of the scope of our study.

References

[1] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services Concepts, Architectures and Applications, Springer-Verlag, Berlin, Heidelberg, 2004.

[2] A. ShaikhAli, O.F. Rana, R.A. Ali, D.W. Walker, UDDIe: an extended registry for web services, in: Proceedings of the Symposium on Applications and the Internet Workshops, January, 2003, pp. 85–89.

[3] C. Zhou, L.T. Chia, B.S. Lee, QoS-aware and federated enhancement for UDDI, International journal of Web Services Research 1 (2) (2004) 58–85.

[4] D. Ardagna, B. Pernici, Adaptive service composition in flexible processes, IEEE Transactions on Software Engineering 33 (6) (2007) 369–384.

[5] Quan Z. Sheng, Boualem Benatallah, Zakaria Maamar, Anne H.H. Ngu, Configurable composition and adaptive provisioning of web services. IEEE Transactions on Services Computing 2 (1) (2009), 34–49.

[6] S. Dustdar, W. Schreiner, A survey on web services composition, International Journal of Web and Grid Services 1 (1) (2005) 1–30.

[7] F. Casati, M. Sayal, M.-C. Shan, Developing e-services for composing e-services, in: Proceedings of 13th International Conference on Advanced Information Systems Engineering, Interlaken, Switzerland, June, 2001, pp. 171–186.

[8] H. Cao1, X. Feng, Y. Sun1, Z. Zhang, Q. Wu, A service selection model with multiple QoS constraints on the MMKP, in: Proceeding of the IFIP International Conference on Network and Parallel Computing Workshops, September, 2007, pp. 584–589.

[9] J. Cardoso, A. Sheth, Semantic e-workflow composition, Journal of Intelligent Information Systems 21 (3) (2003) 191–225.

[10] Jinghai Rao, Xiaomeng Su, A survey of automated web service composition methods, in: Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, San Diego, California, USA, July, 2004, pp. 43–54.

[11] L. Zeng, B. Benatallah, QoS-aware middleware for web services composition, IEEE Transactions on Software Engineering 30 (5) (2004) 311–327.

[12] M. Paolucci, T. Kawamura, T.R. Payne, K. Sycara, Importing the semantic web in UDDI, in: Proceedings of Workshop on Web Services, E-Business and Semantic Web, 2002, pp.225–236.

[13] Organization for the Advancement of Structured Information Systems (OSAIS), Web Services Business Process Execution Language (WSBPEL). <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel>.

[14] Organization for the Advancement of Structured Information Systems (OSAIS) and Web Services Distributed Management (WSDM). <http://www.oasis-open.org/specs/index.php#wsdmv1.1>.

[15] P. Plebani, B. Pernici, URBE: web service retrieval based on similarity evaluation, IEEE Transaction on Knowledge and Data Engineering 21 (11) (2009) 1629–1642.

[16] R. Aggarwal, K. Verma, J. Miller, W. Milnor, Constraint driven web service composition in METEOR-S, in Proceedings of the IEEE International Conference on Services Computing, 2004, pp. 23–30.

[17] R. Para-Hernández, N.J. Dimopoulos, A new heuristic for solving the multichoice multidimensional Knapsack problem, IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans 35 (5) (2005) 708–717.

[18] D. Stefan, G. Alessio, D. John, Exploiting metrics for similarity-based semantic web service discovery, in: Proceedings of International Conference on Web Services, July, 2009, pp. 327–334.

[19] J.M. Ko, C.O. Kim, I. Kwon, Quality-of-service oriented web service composition algorithm and planning architecture, The Journal of Systems and Software 81 (2008) 2079–2090.

[20] S.M. Yuan, Chia-Feng Lin, Ruey-Shyang Wu, Kuan-Yu Chen, Distributed systems management for enterprise web services environment, in: Proceedings of International Conference on New Trends in Information and Service Science, 2009, pp. 384–389.

[21] T. Yu, Y. Zhang, K.J. Lin, Efficient algorithms for web services selection with End-to-End QoS constraints, ACM Transactions on the Web 1 (1) (2007). Article 6.

[22] The Workflow Management Coalition, "The Workflow Reference Model", Document Number TC00-1003, January, 1995.

[23] UDDI.org. <http://www.uddi.org>.

[24] V. Suraci1, S. Mignanti, A. Aiuto, Context-aware semantic service discovery, in: Proceedings of the Third International Conference on Semantics, Knowledge and Grid, October, 2007, pp. 499–502.

[25] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein O. Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, Ulrich Scholz, MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments, Software Engineering for Self-adaptive Systems (2009) 164–182.

[26] W3C WS-Policy Framework Ver. 1.2. <http://www.w3.org/Submission/WS-Policy/>.

[27] World-Wide Web Consortium (W3C), Simple Object Access Protocol (SOAP). <http://www.w3.org/2000/xp/Group/>.

[28] World-Wide Web Consortium (W3C), Web Services Description Language (WSDL). <http://www.w3c.org/TR/wsdl>.

[29] X. Liu, G. Huang, H. Mei, Discovering homogeneous web service community in the user-centric web environment, IEEE Transactions on Services Computing 2 (2) (2009) 167–181.

[30] Y.S. Luo, Y. Qi, L.F. Shen, D. Hou, C. Sapa, Y. Chen, An improved heuristic for QoS-aware service composition framework, in: Proceeding of IEEE International Conference on High Performance Computing and Communications, September, 2008, pp. 360–367.

[31] Ziqian Xu, Patrick martin, Wendy Powley, Farhana Zulkernine, Reputation-enhanced QoS-based web service discovery, in: Proceedings of International Conference on Web Services, July, 2007, pp.249–256.

[32] Zibin Zheng, Michael R. Lyu, A QoS-aware fault tolerant middleware for dependable service composition, in: Proceeding of IEEE/IFIP International Conference on Dependable Systems & Networks, 2009, pp. 239–248.

[33] M. Alrifai, T. Risse, Combining global optimization with local selection for efficient QoS-aware service composition. In: International Conference on World Wide Web 2009 Proceedings, pp. 881–890.

[34] G. Canfora, M. Di Penta, R. Esposito, M.L. Villani, A framework for QoS-aware binding and re-binding of composite Web services, Journal of Systems and Software 81 (10) (2008) 1754–1769.

[35] J.L. Pastrana, E. Pimentel, M. Katrib, QoS-enabled and self-adaptive connectors for Web Services composition and coordination, Computer Languages, Systems & Structures 37 (1) (2011) 2–23.

[36] E. Amaldi, M. Bruglieri, G. Casale, A two-phase relaxation-based heuristic for the maximum feasible subsystem problem, Computers & Operations Research 35 (5) (2008) 1465–1482.
[37] E. Amaldi, V. Kann, The complexity and approximability of finding maximum feasible subsystems of linear relations, Theoretical Computer Science 147 (1–2) (1995) 181–210.
[38] John W. Chinneck, Fast heuristics for the maximum feasible subsystem problem, INFORMS Journal on Computing 13 (3) (2001) 210–223. Summer.
[39] Edoardo Amaldi, Pietro Belotti, Raphael Hauser, Randomized relaxation methods for the maximum feasible subsystem problem, Lecture Notes in Computer Science 3509/2005 (2005), 249–264.
[40] T. Rajendran, P. Balasubramanie, Resmi Cherian, An efficient WS-QoS broker based architecture for web services selection, International Journal of Computer Applications 1 (9) (2010) 79–84.