2009 International Conference on Frontier of Computer Science and Technology

A Heuristic Approach with Branch Cut to Service Substitution in Service Orchestration[1]

Jinbo Du
School of Software
Shanghai Jiao Tong University
Shanghai, P.R.China
dujinbo@gmail.com

Haopeng Chen
School of Software
Shanghai Jiao Tong University
Shanghai, P.R.China
chen-hp@sjtu.edu.cn

Can Zhang
School of Software
Shanghai Jiao Tong University
Shanghai, P.R.China
zhangcan05@gmail.com

Abstract--With the rapidly growing number of Web services throughout the Internet, Service Oriented Architecture (SOA) enables a multitude of service providers (SP) to provide loosely coupled and inter-operable services at different Quality of Service (QoS) levels. This paper considers the services are published to a QoS-aware registry. The structure of composite service is described as a Service Orchestration that allows atomic services to be brought together into one business process; This paper considers the problem of finding a set of substitution atomic services to make the Service Orchestration re-satisfies the given multi-QoS constraints when one QoS metric went unsatisfied at runtime. This paper leverage hypothesis test to detect possible fault atomic services, and propose heuristic algorithms with different level branch cut to determine the Service Orchestration substitutions. Experiments are given to testify the algorithms are effective and efficient, and the probability cut algorithm reaches a cut/search ratio of 137.04% without loss solutions.

SOA; Web Service; Service Orchestration; QoS-aware Service Substitution

I. INTRODUCTION

There has been significant recent interest in QoS-aware Service Composition[1,2,3,4,5]. The number of Web services grows rapidly throughout the Internet, and several UDDI Business Registries (UBR) established by IBM, SAP, Microsoft have emerged in recent years. However, none of them are pervasively used according to the survey in [6]. The primary reason is that they only considered functional criteria, while Service Oriented Architectures (SOA) enable a multitude of service providers (SP) to provide loosely coupled and inter-operable services at different Quality of Service (QoS) levels. As a result, certain subset of all possible non-functional properties that may affect the quality of the service collectively referred to as QoS[7,8,9] has been taken into account[10].

QoS-aware registry enables inquire services based on QoS metrics. Benefit from this feature, We consider a composite service described as a Service Orchestration that brought atomic services together into one business process, such as BPEL[11], which allows constructs such as sequence, switch, while, flow, and pick and brings atomic services together. We consider the Service Orchestration satisfies a certain QoS constraints, however, while the QoS of atomic services changes at real-time, the Service Orchestration would become unsatisfied, which can be detected by the Orchestration Execution Engine or 3rd part monitors. In this environment, it makes sense to investigate mechanisms to substitute atomic services by inquires from a QoS-aware registry to re-satisfy the QoS constraints. This problem is referred as the QoS-aware Service Substitution.

A traditional approach to solve this problem is to re-select all the atomic services, which transform this problem into an Optimal Service Selection problem. The optimal service selection is clearly an NP-Hard problem[12,13], which generates all possible service selections, and selects the optimal composition. Many researchers have proposed different solutions to solve this problem, and some give sub-optimal solutions in a reasonable asymptotic time complexity. Evolved algorithms by using normalization that computed Multiple QoS metrics into one comparable value, and then the knapsack[2] or genetic algorithm can be leveraged to achieve a sub-optimal solution[16,17]. Another approach is substitute services by using a backup service path generated in the initial selection.

There are clearly problems with the approaches above. The re-selection solution cost too much time and additional unnecessary computation to selection faultless atomic services, and the backup service path would becomes also unsatisfied while runtime.

Furthermore, service substitution is different from service selection while the runtime characteristics are considered. Based on the above, we propose a heuristic approach based on the analysis of runtime QoS data for each atomic services. There are two main problems to tackle in order to solve QoS-aware service substitution problem. The first is determine which atomic services should be substituted and their substitution order; Another problem is how to search a set of substitution atomic services to re-satisfy the given QoS constraints. The dynamic substitution problems such as semantically compatibility and session state maintenance[18] are not within the scope of this paper.

[1] This paper is supported by the National High-Tech Research Development Program of China (863 program) under Grant No. 2007AA01Z139.

978-0-7695-3932-4/09 $26.00 © 2009 IEEE
DOI 10.1109/FCST.2009.44

IEEE computer society

The paper is organized as follows. Section II introduces the related work and assumptions about the problem. Section III introduces the fault detection algorithm in details. Two heuristic services substitution algorithms are described in Section IV. Experimental results are given in Section V. Section VI concludes the paper.

II. RELATED WORK AND ASSUMPTIONS

Although QoS plays an important role in Web service discovery, a widely-accepted standard specifying QoS metrics and measurement has not existed yet. Many researchers have proposed their own QoS definitions [8] [19]. As a result , we are focused on the observable QoS metrics, which can be presented by a numerical value. Without loss of generality, we made the assumption that all QoS metrics are decreasing functions of the quality, and the best value of each QoS metrics is 0.

There are two major facets of related work to support our approach. For each facet, we will make some assumptions to make the paper clear and in focus. Firstly, A QoS-aware registry should be valid in order to inquire substitution services and their QoS data[10]. We made the assumption that we do not need to take functional criteria into consideration and focus on QoS metrics only.

The second facet is the QoS metric evaluation method[14]. Such methods evaluate the QoS of service orchestration from the QoS attributes of its atomic services and from the orchestration structures. We also made the assumption that the probability density function (pdf) and cumulative distribution function (CDF) of all atomic service QoS metrics are known, and all the QoS metrics are independent of each other. With these pdfs and CDFs, the pdf and CDF of the service orchestration QoS metrics are assumed computable. In fact, the two facets of related work are under researching in our team, and we leverages them in our experiments yet.

Let,

- A service orchestration $\mathbf{O}$ be composed of $\mathbf{N}$ atomic services $\mathbf{s_i}$, i = 1, 2, … , $\mathbf{N}$.
- $\mathbf{m}$ be the QoS metrics dimension, where $\mathbf{m} < \mathbf{N}$.
- $\mathbf{Q} = \{\mathbf{A_1}, \mathbf{A_2}, \dots, \mathbf{A_m}\}$ be a random variable stands for the QoS metric vector of a service, where $\mathbf{A_i}$ stands for each dimension QoS attribute with a given probability density function $\mathbf{p}(\mathbf{A_i})$ and a cumulative distribution function $\mathbf{P}(\mathbf{A_i})$.
- $\mathbf{Q_c} = \mathrm{f}(\mathbf{Q_1}, \mathbf{Q_2}, \dots, \mathbf{Q_N})$ be the QoS metric evaluation function for certain service orchestration, and $\mathbf{Q_c}$ be a random variable stands for the QoS metric vector of the service orchestration. Here we made the assumption the probability density function $\mathbf{p}(\mathbf{Q_c})$ and a cumulative distribution function $\mathbf{P}(\mathbf{Q_c})$ is computable.
- $\mathbf{Q_{max}} = \{\mathbf{A_{1max}}, \mathbf{A_{2max}}, \dots, \mathbf{A_{mmax}}\}$ be the QoS constraints vector of the service orchestration. Notice we have made the assumption that each QoS metrics are decreasing function of its quantity.

III. ATOMIC SERVICE FAULT DETECTION

When one QoS metric of the service orchestration is reported unsatisfied to the QoS constraints $\mathbf{Q_{max}}$, we try to detect which service or services went wrong and mark them as Fault Services.

The reasons of a QoS metric reduction of a Service Orchestration can be classified in two groups:

- One or more atomic services are down significantly.
- No atomic services are down significantly, but the QoS metrics of one or more atomic services downgraded slightly, which results the Service Orchestration's QoS metric went out of the given QoS constraints.

The first case is trivial and can be detected directly. Thus, we focus on the second case in the following. The basic idea of the detection for this case leverages hypothesis test. We presents, in what follows, the detection procedures that we develop in four principal phases.

A. QoS metric sample choice

Because all the QoS metrics are assumed observable, and the recent historical QoS metric data are recorded by a monitor or can the Orchestration Execution Engine. We select the most recent data as our samples under the assumption: the size of the sample space should have a reasonable size; On the other hand, the samples should be the latest data after the fault arrival moment.

B. Decision test establishment

To carry out effectively a detection, it is first of all necessary to define an event carrying the fault information, which constitutes the selected samples. The decision requires definitions of both hypothesis and its confidence interval. In the simplest case we can define the hypothesis like this:

$\mathbf{H_0}$: *The certain QoS metric is acceptable.*

$\mathbf{H_1}$: *The certain QoS metric is significantly unsatisfied.*

And the confidence interval:

$\mathbf{E_r}$: *The hypothesis rejection area,*

$\mathbf{E_a}$: *The hypothesis acceptance area.*

As we mentioned in section II, assumptions were made that the probability density function (pdf) and cumulative distribution function (CDF) of all the QoS metrics are known, and all the QoS metrics are independent of each other.

Many measurable QoS metrics yields or approximately yields one of normal distribution, log-normal distribution, Fisher distribution, exponential distribution or student distribution[15]. For each distribution, we select appropriate test method.

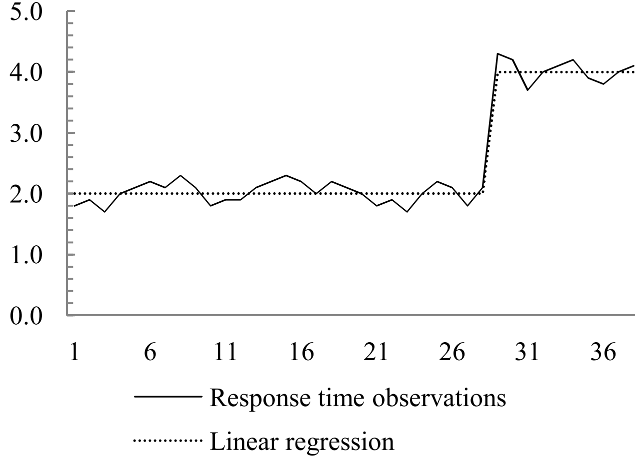

Figure 1. Response time observations, fault occurred in 28th second.

For example, in some cases, response time(RT) yields log-normal distribution[1], thus the log(RT) yields normal distribution. Without loss of generality, we take normal distribution in the following.

α: We should select a Significance level **α** either depend on user pre-defined or by default, the selection of which reflect on the count that how many fault atomic service could be identified. The more fault identified, the higher success probability of the service substitution process, however, it also brings a much expensive computing cost in the worst time when we found no substitution is valid.

Mathematical expectation μ_0:The original QoS metric for a atomic service while the Service Orchestration runs normally, which would be given by the Monitor or the Orchestration Execution Engine.

Then the significance index of a atomic service **s** can be given in formula (1).

$$\mathbf{Significance}(\mathbf{s}) = \frac{\bar{x} - \mu_0}{\frac{S}{\sqrt{n}}} \quad (1)$$

Thus, **Significance**(**s**) yields student distribution, where S stands for the sample variance, and $\bar{x}$ stands for the sample mean, and n stands for the sample space size.

As we assumed in section II, all QoS metrics are decreasing functions of their qualities. Thus, the original hypothesis and checking hypothesis can be described as:

$\mathbf{H_0}$: Significance(s) ≤ α

$\mathbf{H_1}$: Significance(s) > α

Fig. 2 shows the curve of student distribution and its confidence interval. Based on the above, we consider **s** as a Fault Service only when the original hypothesis $\mathbf{H_0}$ are rejected.

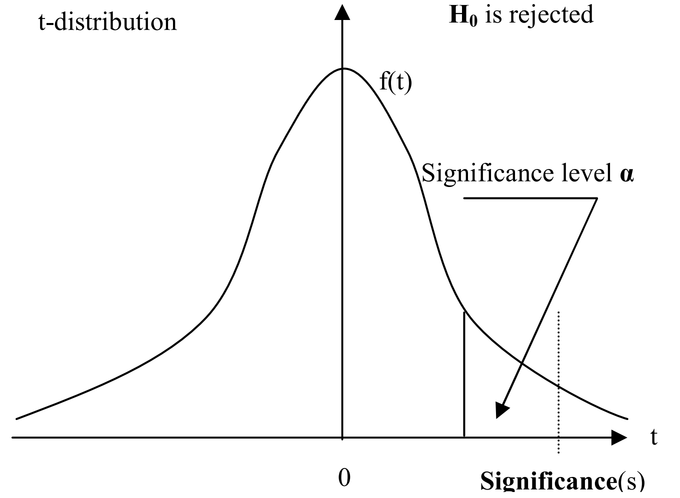

Figure 2. Student distribution and its confidence interval.

C. *Influence Factor calculation*

The **Significance**(**s**) is well described how the atomic service **s** deviated from its normal/original QoS metric, but we cannot order the Fault Services by their significance index. Atomic services with the same QoS metric values would contribute unequally to a Service Orchestration. Let **Influence**(**s, O**) be the influence factor of a service **s** in Service Orchestration **O** when runs normally, and obviously **Influence**(**s, O**) should be also considered as another important parameter.

Influence(**s, O**) of different QoS metric types should be calculated differently, the calculation algorithm is similar to the QoS metric evaluation method[14], which we introduced one of our related works in section II. We only take response time as an example and give the algorithm to compute the influence factor. The algorithm leverages some Service Orchestration statistics data such as the Branch Probability and Average Loop Count, which can be gathered by the Orchestration Execution Engine or 3rd party monitors. There are 4 basic cases in a orchestration, and formula (2) to (5) are given for each case in the following:

- Sequence:

$$\mathbf{Influence}(\mathrm{s,i}) = \sum_{\mathrm{k} \in \mathrm{children(i)}} \mathbf{Influence}(\mathrm{s,k}) \quad (2)$$

- Branch:

$$\mathbf{Influence}(\mathrm{s,i}) = \sum_{\mathrm{k} \in \mathrm{children(i)}} \mathbf{Influence}(s,k) \times \mathbf{BranchProbability}(\mathrm{k}) \quad (3)$$

- Loop:

$$\textbf{Influence}(s, i) = \textbf{AverageLoopCount}(i) \times \sum_{k \in \text{children}(i)} \textbf{Influence}(s, k) \quad (4)$$

• Concurrent:

$$\textbf{Influence}(s, i) = \max_{k \in \text{children}(i)} \textbf{Influence}(s, k) \quad (5)$$

Where i and k are tree nodes in the service orchestration **O**, which is described in a tree structure (see Fig. 3).

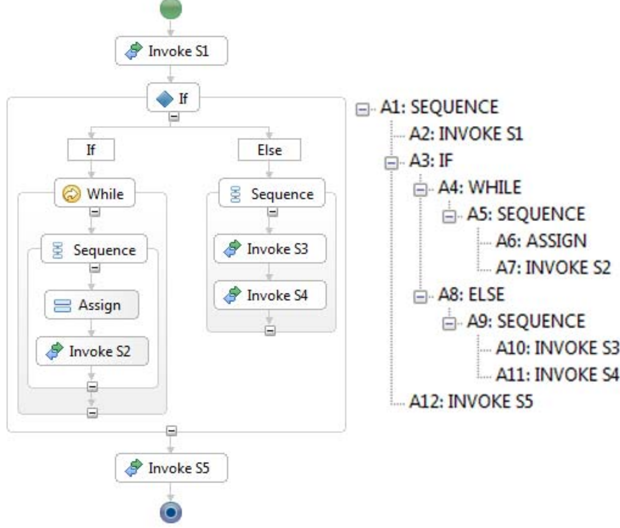

Figure 3. An example of an service orchestration on the left, the corresponding tree structure on the right.

Let's define the Fault Service Influence Effect of a atomic service **s** as formula (6) in the following:

$$\textbf{Effect}(\mathbf{s}) = \textbf{Influence}(\mathbf{s}) \times \textbf{Significance}(\mathbf{s}) \quad (6)$$

Where **Effect(s)** will approximately indicate how significant the QoS metric of fault atomic service contributes to the service orchestration.

D. Fault services set establishment

The results have be calculated from the previous phases and the only work to do is an dynamic adjustment to the Significance Level **α** if there are too many or no fault services are detected. It is reasonable to keep the amount of fault services in a constant level.

The last step we put every Fault Service **s** and its Fault Service Influence Effect **Effect(s)** into a Fault Services Set **F**, which will be used in the next section.

As discussed above, we present the pseudo code of fault service detection in Algorithm 1, Where A(s) stands for the original QoS metric value of a atomic service **s**.

Algorithm 1 Fault Service Detection with Hypothesis Test

```
01: function CalculateInfluence (s, i) returns (Influence)
02:    if i is a leaf node then
03:        if activity(i) is invoke s then
04:            return A(s);
05:        else
06:            return 0;
07:        end if
08:    else
09:        switch activity(i)
10:        case sequence:
11:            return ∑k ∈ children(i) CalculateInfluence(s, k);
12:        case branch:
13:            return ∑k ∈ children(i) CalculateInfluence(s, k) ×
14:                       BranchProbability(k);
15:        case loop:
16:            Inf ← ∑k ∈ children(i) CalculateInfluence(s, k);
17:            return AverageLoopCount(i) × Inf ;
18:        case concurrent:
19:            return maxk ∈ children(i) CalculateInfluence(s, k);
20:        end switch
21:    end if
22:end function
23:
24:function FaultDetection(O) returns (F)
25:    F ← NULL;
26:    for each s   O do
27:        Significance ← HypothesisFunction(s);
28:        if Significance > α then
29:                Influence ← CalculateInfluence(s, O);
30:                Effect ← Significance × Influence;
31:                insert s to F order by Effect descending;
32:        end if
33:    end for each
34:    return F;
35:end function
```

IV. SERVICE SUBSTITUTION SEARCH ALGORITHMS

In this section the service substitution search algorithms will be introduced in details. Notations will be given at the beginning and then the heuristic substitution search algorithm with basic branch cut. After that a probability branch cut algorithm will be introduced as an improvement of the former basic heuristic algorithm.

A. Basic idea and notations

As we discussed in section I, without regard to the runtime characteristics, service substitution algorithm is a partial service selection procedure. However, service substitution process is more strict to the asymptotic time complexity because it should be computed at runtime. As we discussed in the above section, we managed to limit the Fault Services to a constant level while our Fault Service detection phase, and calculated the Fault Service

Influence Effect **Effect(s)** from the runtime historical QoS data as one of the heuristic factors.

Let,

- **F** be the Fault Services set detected in the previous section. The atomic services in **F** are already sorted decreasingly by its Fault Service Influence Effect **Effect(s)**.
- **c** be the count of atomic services in **F**.
- **s ∈ F** be the services in the Fault Services set.
- **B(s)** be the backup services set of service **s**. Notice the backup services will be looked up dynamically in the QoS aware registry.
- **b(s) ∈ B(s)** be a backup service in Set **B(s)**.
- $\mathbf{Q(s) = \{A_1(s), A_2(s), ..., A_m(s)\}}$ be the QoS metrics of atomic service **s**.
- **Z** be the set of all possible service substitutions of the fault atomic services of Service Orchestration **O**. Notice that not all the atomic services in **F** should be substituted until the **O** becomes satisfy QoS constraints $\mathbf{Q_{max}}$.
- **z∈ Z** be a service substitution of 1 to **c** services that support the Service Orchestration **O**. Notice that each **z** is also a Service Orchestration and its QoS could be computed.
- $\mathbf{Q(z) = \{A_1(z), A_2(z), ..., A_m(z)\}}$ be the QoS metrics of a service substitution **z**.
- $\mathbf{N(z) = \{NA_1(z), NA_2(z), ..., NA_m(z)\}}$ be the normalized QoS metrics of **Q(z)**. The normalization process scales each dimension of the values into the range of 0 and 1.

B. Basic heuristic substitution algorithm

As we mentioned in section II, assumptions are made that there is already QoS metric evaluation function as following:

- **extern function** QoSMetric(z) **returns Q(**z);

This function calculates the QoS metrics for Service Orchestration **z** by its structure and all QoS metrics of its atomic services. There are many ways to model and implements this function[14]. Most implementation treats different QoS metrics and different Service Orchestration structure separately or divided into different groups. Notice in order to make a more precise evaluation, the runtime heuristic data such as branches probabilities and average loop count should also be passed to this function, but it doesn't represented in the function definition for the sake of a better readability.

Algorithm 2 Heuristic Service Substitution Search Algorithm with Branch Cut (HSSSABC)

```
01:function ComputeDistance(T) returns (T)
02:    for each z ∈ T do
03:        N(z) ← normalize(Q(z));
04:    end for each
05:    T ← sort T with |N(z), space(0, normalize(Qmax))|
06:    return T;
07:end function
08:
09:function HeuristicSubstitution (z, i) returns (z)
10:    if i > c then return NULL;
11:    s ← F[i];
12:    T ← NULL;
13:    Q ← (∞)m;
14:    for each b ∈ B(s) do
15:        if exist v∈B(s) satisfy all Ai(b) > Ai(v) then continue;
16:        z ← substitute s with b in z;
17:        add z to T;
18:        if exist Ai(b) ≥ Ai then continue;
19:        Q(z) ← QoSMetric(z);
20:        if Q(z) satisfy Qmax then return z;
21:        for each Ai ∈ Q
22:            if Ai(z) ≥ Aimax and Ai < Ai(b) then Ai ← Ai(b);
23:        end for each
24:    end for each
25:    T ←ComputeDistance(T)
26:    for each z ∈ T
27:        z ← HeuristicSubstitution(z, i + 1);
28:        if z ≠ NULL return z;
29:    end for each
30:    return NULL;
31:end function
```

Algorithm 2 shows the detailed steps of the heuristic with dominated branch cut and bound cut. The initial invocation is HeuristicSubstitution(O, 1). Notice that all set are one-based indexing in this paper. Let T be a local temporary set of unsatisfied substitutions. In line 5, space(A, B) stands for the cuboid space enclosed by planes through point A or B and parallel to coordinate surfaces. |A, S| stands for the distance between point A and the space S. Code line 0 to line 7 defines a utility function ComputeDistance(T) used to sort those unsatisfied substitutions by the distance between the QoS metrics of each substitutions and the satisfied solution space. Notice that the normalization procedure could be implemented in variable approaches.

Fig. 4 shows the solution space and the satisfied solution(feasible solution) space of a 2 dimension QoS metrics constrained problem. Each axis represents a QoS metric. The solution space is the area delimited by the dashed lines, which indicate the lower and upper bounds for Q1 and Q2 of the Service Orchestration. The satisfied solution space is represented by the dotted area. It is the portion of the solution space delimited by the lower bound and the QoS constraints (Qmax). The function ComputeDistance(T) calculate for each substitution in set T the distance to the solution, and sort the result in an increasing order as a heuristic factor. As shown in Figure 4, z1 is closer to the satisfied solution space than z2 and z3, substitute other Fault Services in z1 would have a higher successful probability than z2 and z3. As shown in

the algorithm 2, It sorts set T in line 25, and then invoke itself recursively in line 27.

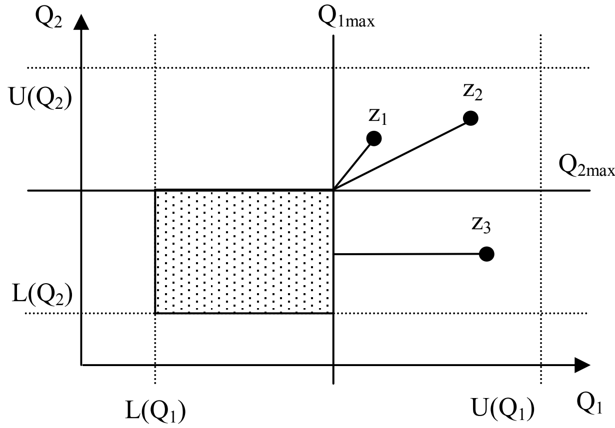

Figure 4. A conceptual representation of the solution space and of the satisfied solution. L(Q1) denotes the lower bound of the Q1 in the solution space, while U(Q1) denotes the upper bound, so are L(Q2) and U(Q2). z1, z2, z3 denotes substitutions in the solution space.

From line 14 to line 24, we try to substitute the (i)th fault service in z, and check whether there is a satisfied substitution.

There are another two optimizations:

- Dominated Branch Cut

The idea of Dominated Branch Cut is easy to comprehend. It is shown in line 15 in the Algorithm. For a given backup service, of which every dimension of QoS metrics are lower than another backup service, the given backup service and all its subsequent substitutions should be cut.

- Bound Cut

Bound Cut is designed to cut redundant invocations to function QoSMetric(z) by set a Bound Cut Threshold **Q**. The bound is initialized with the m-dimension vector $(\infty)_m$ in line 13 and reduced in line 21 to line 23, and it represents the unsatisfied QoS metric in every QoS dimension. That's to say, if a QoS of a backup service has one or more dimension greater than Q, the substitution of this backup service must be an unsatisfied solution and it should be skip. This optimization is shown in line 18.

Now we use the proposed algorithm to compute the solution for the Service Orchestration in Fig.3 using the data listed in Table I. We take the following as known: the average loop count of the WHILE activity in Fig.3 is 3.0, and the branch probabilities of the IF-ELSE activities are both 0.5.

In order to show how the substitution algorithm works, we skip the Fault Services detection phase and assume S1, S5, S2 are Fault Services, which have already been detected and ordered by their Influence Effects. S3, S4 run normally and no need to be substituted. We also assume that there are only three QoS metrics are took into account. Let $\mathbf{Q}_{\mathbf{max}}$ = (3.0, 4.0, 2.0).

TABLE I. QOS METRICS OF ATOM SERVICES AND ITS BACKUP SERVICES IN A SERVICE ORCHESTRATION. THE SERVICE ORCHESTRATION IS SHOWN IN FIG. 3

Services	QoS	Backups	QoS
S1	(1, 1, 1)	S1-B1	(0.4532,0.8799,0.4474)
		S1-B2	(0.7102,0.3011,0.6500)
		S1-B3	(0.6767,0.2232,0.6220)
S2	(1, 1, 1)	S2-B1	(0.1710,0.6405,0.4474)
		S2-B2	(0.2584,0.6322,0.2812)
		S2-B3	(0.6518,0.5516,0.3812)
S3	(1, 1, 1)	-	-
S4	(1, 1, 1)	-	-
S5	(1, 1, 1)	S5-B1	(0.9875,0.8826,0.0261)
		S5-B2	(0.3793,0.2538,0.8554)
		S5-B3	(0.3317,0.6754,0.4680)

The steps in this example are given below:

- Initial State: z = (S1, S2, S3, S4, S5), **Q**(z) = (3.5, 4.0, 3.5). Notice each dimension of Q(z) would be unequal because different QoS metrics are calculated differently by function QoSMetric(z). Bound Cut Threshold **Q** = (∞, ∞, ∞).
- Step I: z_1 = (S1-B1, S2, S3, S4, S5), Q(z_1) = (2.9532, 4.8799, 2.9474), **Q** = (∞,0.8799, 0.4474).
- Step II: Service S1-B2 meet the Dominated Branch Cut condition. Substitution (S1-B2, S2, S3, S4, S5) and all its subsequent substitutions are cut.
- Step III: z_2 = (S1-B3, S2, S3, S4, S5). Because the 3rd QoS attribute of S1-B3 is greater than the corresponding attribute in **Q** (0.6220 > 0.4474), Bound Cut meet, no need to calculate Q(z_2) at present.
- Step IV: Compute distances to the solution space for both z_1 and z_2. ComputeDistance(z_1) = 1.4211; ComputeDistance(z_2) = 1.4190. Sort solutions in this order: (z_2, z_1) and start to substitute S5 recursively.
- Step V: z_3 = (S1-B3, S2, S3, S4, S5-B1), **Q**(z_3) = (3.1642, 4.1058, 2.1481), **Q** = (0.9875, 0.8826, 0.0261).
- Step VI: z_4 = (S1-B3, S2, S3, S4, S5-B2). Bound Cut meet, no need to calculate **Q**(z_4) at present.
- Step VII: z_5 = (S1-B3, S2, S3, S4, S5-B3). Bound Cut meet, no need to calculate **Q**(z_5) at present.
- Step VIII: Compute distance for z_3, z_4, z_5 and get the order (z_5, z_4, z_3).
- Step IX: z_6 = (S1-B3, S2-B1, S3, S4, S5-B1), **Q**(z_6) = (2.7497, 3.7463, 1.8718). where **Q**(z_6) < $\mathbf{Q}_{\mathbf{max}}$. Solution is found after six substitutions.

C. *Probability Branch Cut improvement*

As we mentioned in section II, the assumption was made that the following function existed:

• **extern function** ProbabilityQoSMetric(z)
returns p(z), P(z);

With a certain Service Orchestration z, and with the probability density function (pdf) and cumulative distribution function (CDF) of each atomic services, the function compute the pdf p(z) and CDF P(z) of the Service Orchestration z.

Thus, Let

$$P = \prod_{i=0}^{m} \int_{0}^{Aimax} p(z) \qquad (7)$$

In formula (7), P denotes the probability of the event: z satisfy $\mathbf{Q_{max}}$.

According to the principle that extremely small probability did not occur in a time, we set a Branch Cut Threshold $\boldsymbol{\beta}$ to a proper small positive value. If $P < \beta$, we consider the a substitution z is not valuable for further substitutions and the search branch should be cut.

Based on the idea above, we improved the function ComputeDistance(T) in Algorithm 2 to the function ComputeDistanceAndCutBranch(T), which presents in Algorithm 3. Notice in the computing of Service Orchestration's pdf and CDF, only not-yet-substituted atomic services QoS metrics are treated as random variables. The other atomic services QoS values are treated as constants or one-point distribution as well.

Algorithm 3: Compute Distance And Cut Branch

```
01:function ComputeDistanceAndCutBranch(T) returns (T)
02:     for each z ∈ T do
03:         p(z), P(z) ← ProbabilityQoSMetric(z);
04:         P ← ∏_{i=0}^{m} ∫_0^{Aimax} p(z) ;
05:         if P < β then
06:             remove z from T;
07:         else
08:             N(z) ← normalize();
09:         end if
10:     end for each
11:     T ← sort T with |N(z), space(0, normalize(Qmax))|
12:     return T;
13:end function
```

D. *Adaptive the Service*

After the substitution, if solution were found, we got a new Service Orchestration **z**. The next phase is to adaptive the parameters and session state, for which some solutions are given by other researchers and it is out of the scope of this paper.

V. Simulation Experiments

We implemented the algorithm to conduct experiments aimed at evaluating effective and efficient of the two substitution algorithms.

Our experiments still leverage the Service Orchestration given in Fig.3. Three QoS metrics are consider in our QoS metrics vector: The Response Time, the Price of Service and Execution Time. We generate all the QoS attributes randomly. Then all the QoS attribute yield uniform distribution in (0, 1). The normalization function we choose was:

$$f(x) = \frac{2}{\pi} \times \tan^{-1}(x) \;, x \in (0, \infty)$$

We use the same loop count for the WHILE activity the same branch probabilities for the IF-ELSE activities in the sample we given in section IV. We still let S1, S5 and S2 be the Fault Services, while we increase the count of backup services for each atomic services to 5.

We first set $\boldsymbol{\beta}$ to 0.02 and $\mathbf{Q_{max}}$ to (3.0, 4.0, 2.5), and run both algorithms 100 times. Table II shows the first 10 results of both The Basic Algorithm (Algorithm 2) and Evolved Algorithm. Notice when no solutions are found, Search + Dominated Cut + Prob. Cut equals to the solution space size.

TABLE II. Part of Simulation Experiment Results

Exp.	Alg.	Result	Search	Dom. Cut	Bound Cut	Prob. Cut
1	Basic	Succeed	35	28	23	0
1	Evolved	Succeed	29	24	20	10
2	Basic	No Sol.	124	31	85	0
2	Evolved	No Sol.	54	31	34	70
3	Basic	Succeed	27	36	15	0
3	Evolved	Succeed	15	33	9	15
4	Basic	Succeed	5	111	1	0
4	Evolved	Succeed	5	111	1	0
5	Basic	Succeed	61	93	25	0
5	Evolved	Succeed	31	93	17	30
6	Basic	Succeed	53	41	25	0
6	Evolved	Succeed	33	36	16	25
7	Basic	Succeed	17	37	9	0
7	Evolved	Succeed	17	37	9	0
8	Basic	Succeed	13	107	5	0
8	Evolved	Succeed	13	107	5	0
9	Basic	Succeed	23	10	9	0
9	Evolved	Succeed	11	7	6	45
10	Basic	No Sol.	68	87	29	0
10	Evolved	No Sol.	41	69	20	45

We then carry out experiments under different QoS constraints $\mathbf{Q_{max}}$ and different Probability Cut Threshold $\boldsymbol{\beta}$. For each pair of QoS constraints and Probability Cut Threshold, we simulated 1000 times with different backup services generated randomly. The results are shown in Table III.

TABLE III. STATISTICS UNDER DIFFERENT B AND QMAX VALUES. ALG. B DENOTES BASIC ALGORITHM AND ALG. E DENOTES EVOLVED ALGORITHM

β	Q_{max}	Has Sol.	Alg.	Search	Prob-Cut	Error
0	2.0 3.0 2.0	No (668)	B	33.63%	0.00%	
			E	12.47%	27.22%	-
		Yes (332)	B	8.89%	0.00%	
			E	4.52%	6.68%	0.00%
	2.5 3.5 2.5	No (19)	B	1.23%	0.00%	
			E	0.82%	0.49%	-
		Yes (981)	B	14.18%	0.00%	
			E	10.92%	4.97%	0.00%
0.02	2.0 3.0 2.0	No (674)	B	34.35%	0.00%	
			E	11.50%	30.09%	-
		Yes (326)	B	8.45%	0.00%	
			E	4.23%	7.17%	0.00%
	2.5 3.5 2.5	No (14)	B	0.95%	0.00%	
			E	0.58%	0.43%	-
		Yes (986)	B	14.80%	0.00%	
			E	11.27%	5.29%	0.00%
0.1	2.0 3.0 2.0	No (701)	B	35.38%	0.00%	
			E	7.83%	40.27%	-
		Yes (299)	B	8.70%	0.00%	
			E	2.95%	10.45%	1.79%
	2.5 3.5 2.5	No (12)	B	0.72%	0.00%	
			E	0.45%	0.34%	-
		Yes (988)	B	14.33%	0.00%	
			E	9.98%	6.96%	0.00%
0.2	2.0 3.0 2.0	No (643)	B	32.75%	0.00%	
			E	3.85%	44.22%	-
		Yes (357)	B	9.24%	0.00%	
			E	2.41%	17.06%	11.80%
	2.5 3.5 2.5	No (22)	B	1.42%	0.00%	
			E	0.57%	1.04%	-
		Yes (978)	B	14.52%	0.00%	
			E	9.37%	9.54%	0.70%

We draw from our experiments that:

Both the algorithms are effective. The Basic algorithm is efficiency when there is a solution. It can found the solution with an average 11.64% searches in the whole solution space, while the Evolved algorithm with probability branch cut searches only 6.96%.

In the worst time (No substitution is valid in the solution space), the Basic algorithm takes an average 17.55% searches of the whole solution space. The left 82.45% are cut by Dominated Cut, which is simple but useful. The Evolved algorithm searches only 4.76% in the solution space in the worst time in our experiments.

The Evolved algorithm is efficiency than the Basic algorithm. The Evolved algorithm searches less than the Basic one by 60.85% in total, especially 72.88% in the worst time. Notice that the Evolved algorithm searches no more than 12.47% of the solution space in average no matter what QoS constraints and Probability Cut Threshold are given, or whether there is a solution or not.

The Evolved algorithm can go wrong and fail to find existing solutions when the Probability Cut Threshold **β** is too high. In the experiment, there are average 6.25% errors when **β** is set to 20%, but the probability cut rate gains slightly. Notice that when **β** is set to zero, no solution will be lost, and there are still a high cut rate (the cut/search ratio is 137.04%).

When the QoS constraints are loose, the satisfied service substitutions are prone to be turn up, and the Evolved Algorithm cuts less. In practice, because the ProbabilityQoSMetric(z) function consumed additional time to the Basic algorithm, the real execute time of the two algorithms become closer, and sometimes the Evolved Algorithm costs more time than the Basic one. In this condition, it is better to select the Basic Algorithm.

VI. CONCLUSIONS AND FUTURE WORK

Service Orchestrations enables inter-operable services composed through dynamic discovery and substituted at runtime without modification of the source code. With the growth of web services and QoS-aware registries, substitute an atomic service in an SOA application or a composite service becomes common.

This paper introduced a heuristic solution to solve the QoS-aware Service Substitution problems. We presented such an effective mechanism that, detect and substitute fault atomic services in an Service Orchestration to re-satisfy its QoS constraints efficiently. The approach composed in two principal phases: detecting the fault atomic services by using hypothesis test and the searching procedure for proper substitutions. In the searching phase, this paper presented a heuristic algorithm that searches along the most possible solution, and then introduced an evolved algorithm with Probability Branch Cut, which in the experiments reported closer to the former algorithm in most times when the satisfied service substitutions are prone to be turn up, but when the satisfied solutions are rare or even no satisfied solution exists, the latter algorithm will be much more effective while there would be a sacrifice of the correctness in a tunable probability.

We are also currently working on an interesting extension of the work reported here. The probability cut process would be simplified by approximate calculation.

In addition, In order to solve the problem in a strict time limit, The substitution process could be designed into multiple passes, and each passes could be parallel computed in a service substitution cloud, this computing cloud can be generalized as a part of the SOA infrastructures.

REFERENCES

[1] Hiroshi Wada, Paskorn Champrasert, Junichi Suzuki, Katsuya Oba , "Multiobjective Optimization of SLA-aware Service Composition", Proceedings of the 2008 IEEE Congress on Services - Part I, Jul. 2008

[2] T. Yu and K. J. Lin, "Service Selection Algorithms for Composing Complex Services with Multiple QoS Constraints", Proc. of 3rd Int'l Conf. on Service Oriented Computing, pp. 130–143, Dec. 2005.

[3] Jinghai Rao and Xiaomeng Su, "A Survey of Automated Web Service Composition Methods, Semantic Web Services and Web Process Composition", vol. 3387, 2005.

[4] Yu, T., Zhang, Y., and Lin, K.-J. "Efficient algorithms for Web services selection with end-to-end QoS constraints", ACM Trans. Web 1, 1, Article 6 (May 2007), pp. 26. 2007

[5] [5] Lei Li, Jun Wei, Tao Huang, "High Performance Approach for Multi-QoS Constrained Web Services Selection" , Proceedings of the 5th international conference on Service-Oriented Computing, Sep. 2007 - Sep. 2007, Vienna, Austria, pp. 283 – 294

[6] Eyhab Al-Masri,Qusay H. Mahmoud. "Investigating web services on the world wide web", Proceeding of the 17th international conference on World Wide Web, Apr 2008

[7] V. Cardellini, E. Casalicchio, V. Grassi, and L. P.Francesco, "Flow-based service selection for web service composition supporting multiple qos classes", ICWS 2007. IEEE Intl. Conf. Web Services, pp.743–750, July 9-13 2007.

[8] Menasce D.A., "QoS Issues in Web Services", IEEE Internet Computing, November-December 2002, vol. 6, pp. 72-75.

[9] K. Lee, J. Jeon, W. Lee, S.-H. Jeong, S.-W. Park, "QoS for Web Services: Requirements and Possible Approaches", http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/ W3C Working Group, 2003.

[10] Guang Yang, Haopeng Chen, "An Extensible Computing Model for Reputation Evaluation Based on Objective and Automatic Feedbacks", International Conference on Advanced Language Processing and Web Information Technology, 2008

[11] "Web Service - Business Process Execution Language (WS BPEL)" , Version 2.0 - OASIS Committee Draft, 17th May, 2006.

[12] Andreas Mielke, "Elements for response-time statistics in ERP transaction systems", Performance Evaluation, vol. 63, pp. 635-653, Jul. 2006

[13] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. "An Approach for QoS-aware Service Composition based on Genetic Algorithms". In Genetic and Evolutionary Computation Conference, June 2005.

[14] Daniel A. Menasc´e, Emiliano Casalicchio, Vinod Dubey, "A Heuristic Approach to Optimal Service Selection in Service Oriented Architectures", WOSP'08, June 24–26, 2008, Princeton, New Jersey, USA.

[15] R. H. Mokhneche, H. Maaref, and V. Vigneron, "Fault Detection Techniques Analysis and Development of its Procedural Phases" , proceedings of the 13th European Signal Processing Conference (EUSIPCO 2005)

[16] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. "An Approach for QoS-aware Service Composition based on Genetic Algorithms". In Genetic and Evolutionary Computation Conference, June 2005.

[17] M. C. Jaeger and G. Mühl. "QoS-based Selection of Services: The Implementation of a Genetic Algorithm In Conference on Communication in Distributed Systems", Workshop on Service-Oriented Architectures and Service-Oriented Computing, March 2007.

[18] Manel Fredj, Nikolaos Georgantas, Valerie Issarny, Apostolos Zarras. "Dynamic Service Substitution in Service-Oriented Architectures". In Proceedings of the IEEE International Conference on Service Computing (SCC), pp. 3443-3448, 2008.

[19] A. Mani and A. Nagarajan, "Understanding Quality of Service for Web Services", http://www-106.ibm.com/developerworks/library/wsquality .html, January 2002.

[20] H. L. Vu, M. Hauswirth, and K. Aberer. "QoS-Based Service Selection and Ranking with Trust and Reputation Management". Technical Report IC2005029, Swiss Federal Institute of Technology at Lausanne (EPFL), Switzerland, June 2005.

[21] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, "Heuristics for QoS-aware Web Service Composition," Proc. Int'l Conf. on Web Services, Sept. 2006.

[22] C. Peltz. "Web services orchestration and choreography". IEEE Computer, 36(10):46–52, 2003.

[23] Antonio Bucchiarone, Stefania Gnesi, "A Survey on Services Composition Languages and Models", Elsevier Science B. V. 2006