SOCA (2010) 4:157–168
DOI 10.1007/s11761-010-0063-6

SPECIAL ISSUE PAPER

The design and implementation of service process reconfiguration with end-to-end QoS constraints in SOA

Kwei-Jay Lin · Jing Zhang · Yanlong Zhai · Bin Xu

Received: 1 December 2009 / Revised: 19 May 2010 / Accepted: 31 May 2010 / Published online: 20 June 2010

Abstract Service processes in SOA are composed dynamically by services from different service providers. At run-time, some services may become faulty and cause a service process to violate its end-to-end quality of service (QoS) constraints. We propose an effective approach for replacing only faulty services and some of their neighboring services to maintain the original end-to-end QoS constraints. We use an iterative algorithm to search for a reconfiguration region that has replaceable services to meet the original QoS constraint for the region. Services in reconfiguration regions may be replaced using one-to-one, one-to-many, or many-to-one service mappings. By replacing only services in reconfiguration regions rather than the whole service process, reconfiguration overheads are lowered and service disruptions may be reduced. We have implemented the Adaptation Manager in the Llama ESB middleware. Performance study shows that our approach may efficiently repair service processes.

This research was supported in part by Tsinghua National Laboratory for Information Science and Technology (TNList), and China National High-Tech Project (863) under grant No. 2009AA01Z120 and 2007AA010306. Yanlong Zhai was supported by a visiting fellowship from the China Scholarship Council.

K.-J. Lin (✉) · J. Zhang
Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697-2625, USA
e-mail: klin@uci.edu

Y. Zhai
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

B. Xu
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Keywords Service process · SOA reconfiguration · Quality of service · End-to-end constraint · Service accountability

1 Introduction

Service-oriented architecture (SOA) provides a flexible and dynamic paradigm for integrating distributed services into business processes [1,2]. SOA allows enterprises to, statically or dynamically, compose service processes by integrating services deployed by different service operators using some process description language, such as WS-BPEL [3]. As SOA becomes more popular, there are now many Web services with rich functionalities available. Services from different providers can be used to perform similar functionalities. One thus can compose a service process by selecting services from competing service providers. Enterprises and their clients are free to pick and choose services from service providers that best suit their needs.

With SOA now adopted in real-time enterprises (such as Fedex, UPS, and Twitter), it has become increasingly important for enterprises to build service processes that not only provide accurate results, but also deliver a desirable quality of service (QoS), including but not limited to response time, availability, and security. Customers at different application domains may have different concerns. For example, some may have very strict end-to-end response time requirements, while others may be more concerned about the end-to-end security. Some may have a tight budget constraint, while others only care about the reliability of the service process.

In [4,5], we have studied the pre-run-time service process composition problem with end-to-end QoS constraints. We presented algorithms to select services and their service levels for a business process according to a user's functional and

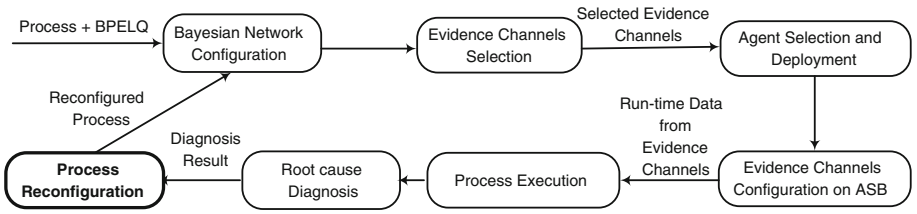

Fig. 1 System deployment flow for accountable SOA

QoS needs. Other projects for service selection in SOA have also been reported in literature [6,7].

However, the QoS delivered by individual services at run-time may not meet their pre-defined QoS levels. Many factors, such as host overload, network congestion, unexpectedly large number of requests, can affect the QoS delivered by services. When that happens, a service process must be repaired immediately in order to continue serving the business process functions effectively. Providing a QoS-consistent SOA system is a big challenge. In [2], self-healing has been identified as one of the leading edge challenges in SOA.

In the event of QoS violation at run-time, it is not desirable to always stop and recompose the entire process. Recomposition is time-consuming since the optimal service selection problem is NP-hard [5]. Moreover, most enterprises would like to have a stable business environment and minimize the number of system shutdown and reconfiguration, so that customers may not experience unexpected service migration too often.

We have recently proposed a solution [8] to reconfigure a service process in order to handle multiple faulty services. A reconfiguration region is constructed for every faulty service. The algorithm tries to include a small number of services in each region, as long as they have enough flexibility and ability to deliver the original QoS in the region. This method can help reduce the overhead of repairing a complete faulty service process since fewer services are involved in reconfiguration. In this paper, we extend the work in [8] by considering more function replacement models, allowing one-to-many and many-to-one service replacements, in addition to the original one-to-one replacement model. We have also designed and implemented the Adaptation Manager in the Llama middleware [9]. We also report the performance study result in this paper.

The rest of this paper is organized as follows. Section 2 presents the background of this work. The service process QoS model is defined in Sect. 3. Our region-based process reconfiguration is presented in Sect. 4. Section 5 describes the system architecture and the main components of the system. Section 6 shows the performance study for region-based reconfiguration with and without the new replacement models. Section 7 compares our work with related work. The paper is concluded in Sect. 8.

2 Background on SOA management

SOA is designed to facilitate simple service integration and dynamic service process composition using services from various service providers. In our previous work [4,5], we have built a QBroker that makes Web service selections based on service client's functional and QoS requirements. We have also developed the SOA management framework to check the accountability of individual services regarding a process's performance issues [10]. Accountability allows a service computing environment to be traceable, measurable, and more dependable. An SOA middleware, the inteLLigent Accountability Middleware Architecture (Llama), has been developed to support accountable SOA [9,11]. Among other components, Llama includes an Accountability Authority (AA) and many accountability Agents that collaborate with one another to perform run-time process monitoring and fault diagnosis. Llama enterprise service bus (ESB) has a built-in support for dynamic service replacement by re-routing service messages to new services.

Figure 1 shows the deployment flow of a service process on Llama. Given a service process composed by QBroker, which defines the QoS constraints for each individual service in a business process QoS specification (called BPELQ), AA will configure a Bayesian network model for the process flow, by incorporating historical knowledge on service performance and dependency. AA also uses the Evidence Channel Selection algorithm to find the best locations for collecting performance data about the process in order to conduct the Bayesian analysis. Accountability Agents for monitoring the service process are then selected and deployed by AA. Evidence channels will collect performance data on services during the executions of processes. If an Agent detects any abnormal behavior from an evidence channel, it will trigger AA fault diagnosis. AA uses its diagnosis engine to identify a list of likely faulty services and ask Agents for fault confirmation. Once confirmed, AA will initiate a process reconfiguration to replace those faulty services.

The study reported in this paper is on the part of service process reconfiguration (the leftmost step in Fig. 1) after faulty services are identified. The problem was first studied in [12], which presents two repair algorithms. The first algorithm (CSPB) generates, for each service along a service process path, a *secondary* path from its predecessor service

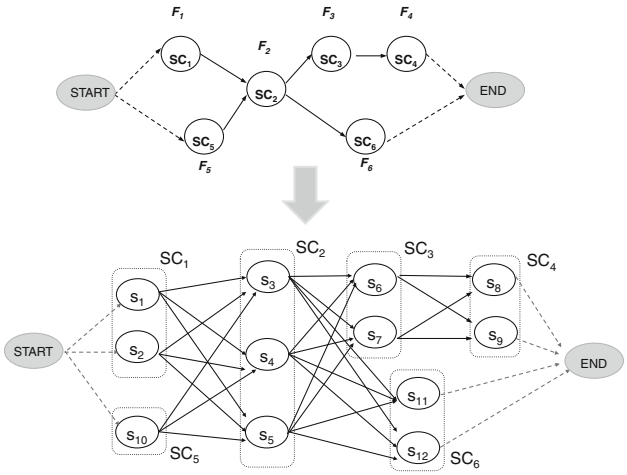

Fig. 2 Service composition model

to the end of the service process (without going through the service). At run-time, a service process can automatically switch to its secondary path whenever a service becomes faulty. The second algorithm (CSPR) builds a *replacement* path for each service, by finding a complete path from the start of service process to an alternate service (which can replace the faulty service), and then to the end of the service process. Neither of the two proposed algorithms works when there are multiple faulty services. To solve this problem, we propose another reconfiguration solution in this paper that can handle multiple faulty services in a service process.

3 Service process QoS model

3.1 Service process model

In our SOA model, every service is classified by their functionalities into service classes. Every service in a service class has the same functionality, input and output types, but may deliver a different QoS.

The top of Fig. 2 shows a function graph. In the function graph, every node represents a function that maps to a service class SC_i. Four possible flow paths, $[F_1, F_2, F_3, F_4]$, $[F_1, F_2, F_6]$, $[F_5, F_2, F_3, F_4]$, $[F_5, F_2, F_6]$, can be used to complete the process. Each service class s_i may have several service candidates. For example, services s_3, s_4, s_5 are included in service class SC_2. We may select one service candidate for each service class in any of the flow paths to compose a service process.

Functions may be replaced in a number of ways. Some functions may simply be replaced by another function. Other functions may be expanded into several functions. For example, a travel planning function may be divided into functions for flight booking, hotel reservation, and car rental. Finally, several existing functions may be replaced by a new individual function. For example, many trip components may

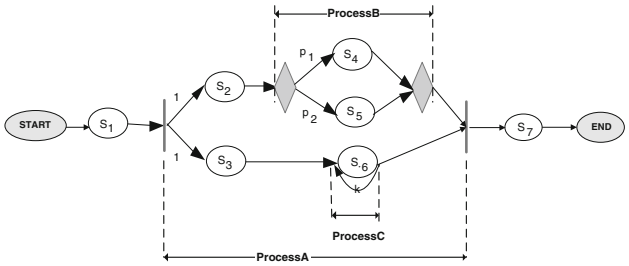

Fig. 3 Service process example

be replaced by a packaged deal that is more convenient yet less expensive. In summary, function replacements can be done in one-to-one, one-to-many, and many-to-one models (Fig. 3).

3.2 QoS model

Because of the dynamic and unpredictable nature of business applications and distributed systems, delivering quality services to meet user demands is a big challenge. Without a careful management of service quality, critical business applications may suffer detrimental performance degradation and result in functional errors and/or financial losses.

Service processes in SOA are composed from simple services connected by different flow structures. End-to-end QoS of a business process is dependent on the process structure. Figure 4 shows a service process example with three subprocesses (A, B, and C). s_2 is followed by either s_4 or s_5 with a probability of P_1 and P_2 respectively. s_6 may be executed for at most k times. Suppose t_i is the response time of service s_i, the end-to-end response time T is defined as follows.

$$T = t_1 + \max(t_2 + \max(t_4, t_5),\ t_3 + k * t_6) + t_7$$

To derive the end-to-end constraint of response time, three types of process structure are considered:

– Parallel: The response time of every path in a parallel structure should be less than the constraint. For example, in Fig. 4, suppose the response time constraint for process A is T_A, then $t_2 + t_{\text{ProcessB}} < T_A$ and $t_3 + t_{\text{ProcessC}} < T_A$;

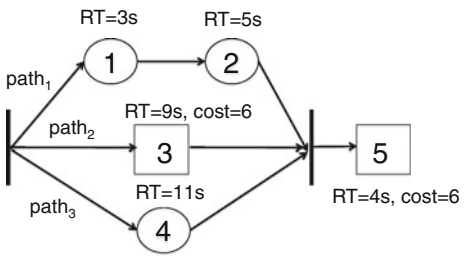

Fig. 4 Flow structure example

– Conditional: The response time of every path in the conditional structure needs to satisfy the constraint.
  For example, suppose the response time constraint for process B is T_B, then $t_4 < T_B$ and $t_5 < T_B$;
– Loop: The number of loop k is added to the node as a weight. For example, suppose the response time constraint for process C is T_C, then $k * t_6 < T_C$.

4 Partial process reconfiguration

Suppose s_3 in Fig. 4 suffers from a QoS problem. We want to find a replacement service for s_3 and ensure that existing users of the service process will receive the same, if not better, QoS. If we cannot find a substitutive service that meets the QoS criteria of s_3, we will try to replace both s_3 and s_6 if s_6 has some candidates for replacement. Replacing both services together gives us more flexibility as long as the replaced services can meet the combined QoS constraints for both. In this way, we can monotonically expand the reconfiguration region to include more services until we find a large enough region with an acceptable set of replaced services. But if we include too many services for replacement, it may not make sense to continue "repairing" the process. In that case, we may as well recompose a complete process from scratch to meet the end-to-end QoS constraints.

We now present a region-based service reconfiguration approach.

4.1 Reconfiguration algorithm

Algorithm 1 shows the reconfiguration algorithm. Given p faulty services and a reconfiguration threshold c ($0 < c < 1$) on the maximum percentage of services to be repaired, the algorithm first tries to find a replacement for every faulty service that meets the original QoS (lines 4–7).

After that, the algorithm starts a loop (lines 9–19) to repeatedly expand all not-yet-repaired regions. Inside the loop, the algorithm tries to expand and to recompose the regions in faulty region set R. It first calls the ExpandRegion algorithm (Algorithm 2 in Sec. 4.2) to include some immediate neighbors into region r_i (line 11). It then tries to recompose each reconfiguration region r_i after finding its QoS constraints.

The recomposition steps (line 14) will be presented in Sect. 4.3.

The algorithm continues until either all regions have a satisfactory replacement or the total number of services in all regions are too big (line 9). If all repair regions together contain too many services (more than $c * s$, s is the process size and c is a threshold factor), the reconfiguration algorithm will be stopped. Instead, the whole service process will be recomposed (line 21).

Algorithm 1 Service Process Reconfiguration Algorithm

Input: faulty services $S_f = \{s_1, \ldots, s_p\}$, threshold c, process size s
Output: replaced subprocesses $R_f = \{r_i\}$
1: $\forall i$, set region $r_i = \{s_i\}$; region set $R = \{r_1, \ldots, r_p\}$, $R_f = \emptyset, d = 0$
2: **for all** r_i in R **do**
3: find the QoS constraints for r_i
4: select another service r_i' for r_i that meets the QoS of r_i
5: **if** success on service selection for r_i' **then**
6: remove r_i from R and add r_i' to R_f
7: **end if**
8: **end for**
9: **while** $R \neq \emptyset$ and $\sum_{r_i \in R_f} |r_i| < c * s$ **do**
10: $d = d + 1$
11: call ExpandRegion(R, d) (Algorithm 2)
12: **for all** r_i in R **do**
13: find the original QoS constraints for r_i
14: recompose r_i to meet QoS (see Sect. 4.3)
15: **if** success on r_i recomposition **then**
16: remove r_i from R and add it to R_f
17: **end if**
18: **end for**
19: **end while**
20: **if** $R \neq \emptyset$ **then**
21: recompose the complete process and set it as R_f
22: **end if**
23: return R_f

Algorithm 2 ExpandRegion Algorithm

Input: Regions with failed services $R = \{r_i\}$, distance d
Output: Expanded regions $R = \{r_i\}$
Require: $\mathcal{D}[i][j]$: distance matrix between any s_i and s_j.
1: **for all** r_i in R **do**
2: find $H_i = \{s_i | s_i \notin r_i, \exists j, s_j \in r_i$ and $\mathcal{D}[i][j] = 1\}$
3: **for all** s_j in H_i **do**
4: **if** s_j is not an end node of a flow structure **then**
5: add s_j into r_i
6: **else**
7: let s_x be the faulty service in r_i
8: identify s_k which is the other structural end node of s_j
9: **if** $s_k \in H_i$ or $s_k \in r_i$ or ($\mathcal{D}[j][x] \leq d$ and $\mathcal{D}[k][x] \leq d$) **then**
10: add all nodes in the structure between (and including) s_j and s_k into r_i
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: **for all** r_i in R **do**
16: merge two regions if they have one or more common services
17: **end for**
18: return R

4.2 Identifying reconfiguration regions

In this section, the region expansion algorithm is presented to identify the sub-processes that should be reconfigured to repair faulty services. We would like to identify a small reconfiguration region to reduce the number of services that will be replaced.

The region expansion algorithm is shown in Algorithm 2. In the algorithm, the service process is represented by a

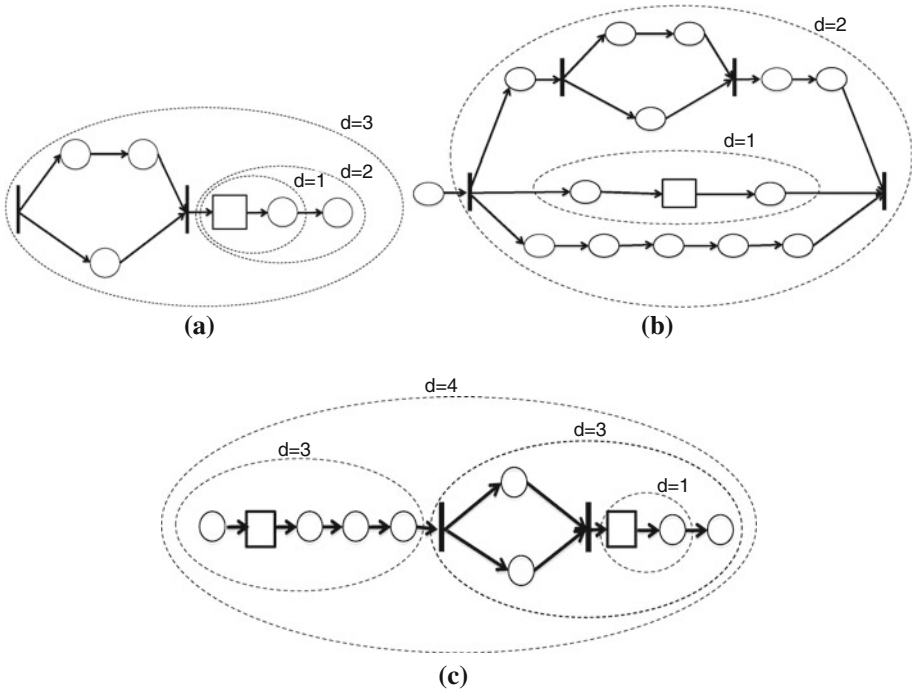

Fig. 5 Reconfiguration region example

directed acyclic graph (DAG). The DAG graph includes both service nodes and control flow nodes, such as AND join and split. A distance matrix is used to record the distance between any two nodes in the DAG.

The input for the algorithm includes the distance bound d and the set of unsatisfied reconfiguration regions that include at least one faulty service. For a faulty service s_x, the algorithm looks for an output region r_i that extends from s_x to some nodes that are connected to s_x with a distance less than d. If a node is within the distance to s_x but is an end node of a structure that is not completely within distance d of the faulty node, it will not be included in the region (line 9). On the other hand, when r_i includes both end nodes of a structure, all nodes in all paths of the structure are automatically included in the region even if their distance to s_x is larger than d (line 10). Finally, if any two regions have an overlap, the algorithm merges the two into one (line 16).

In this structure-based region expansion algorithm, the complete parallel and conditional structures must be considered, i.e. we will not add one end node of a structure into a reconfiguration region with the other. If one of the branches is included in the region, all the other branches in the structure should also be included in the region. For example, in Fig. 5, if service 5 fails and only one path of the structure is involved in the region, say service 3. A reconfiguration may pick a new service 3 that takes 2 seconds and a new service 5 that takes 7 seconds. As a result, the response time for path 3 (service 4 and service 5) will be 18 seconds, causing a response time violation.

On the other hand, if a faulty service is in a structure, before including nodes outside of the structure, all nodes in the structure need to be included in the region. For example, in Fig. 5, if service 3 fails, service 5 should not be included in the region, until all services 1, 2, 3, 4 are in it.

Figure 6 shows examples of producing the reconfiguration regions for faulty services. In all figures, square nodes denote faulty services. By increasing the value of the distance bound, different sub-processes are included in the region. In Fig. 6a, when the distance bound is two, the parallel sub-process is not included in the region. This is because the distance between the faulty service and the start node of the parallel sub-process is larger than two. In order to keep the completeness of a structure, none of the nodes in the parallel structure will be included in the region. Figure 6b shows that there is one failed service in a parallel structure. When the distance is one, two services are included in the region. When the distance bound is increased to two, the whole parallel structure is included in the region. This is because both end nodes of the parallel structure are included in the region. As a result, all nodes of that structure will be included. Figure 6c shows a service process that has two faulty services. When the distance bound increases to four, the two regions are merged into one.

4.3 Recomposition for sub-process

In the Llama architecture (to be discussed in Sect. 5), a Service Repository is used to store all service candidates for every function and all possible function replacement

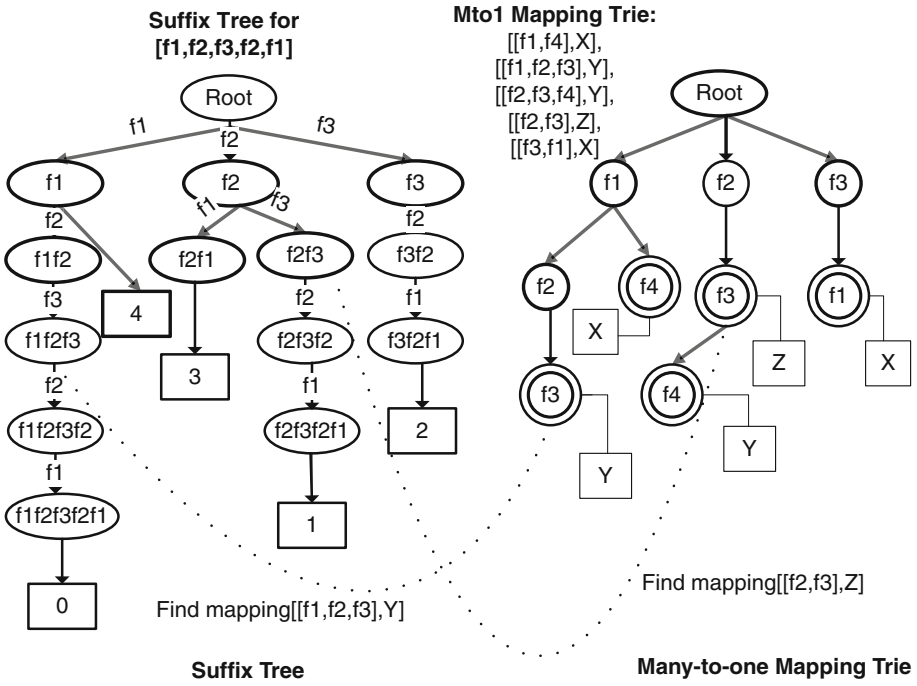

Fig. 6 Many-to-one mapping search example

mappings. A function can be replaced by another function (one-to-one) or a sequence of several functions (one-to-many). It is also possible to replace a sequence of functions by one single function (many-to-one). We assume service owners and domain experts have provided such a mapping database in the Service Repository.

In our approach, region-based recomposition (line 14 of Algorithm 1) is conducted in three steps:

1. Identify the function plan. By identifying the service functions of the service nodes in a reconfiguration region, the function plan for the region will be identified.
2. Expand the function plan to another with more paths. The original function plan may be expanded to another by adding all replacement functions, including one-to-one, one-to-many, and many-to-one mappings. Searching for one-to-one and one-to-many mappings in Service Repository is easy. But finding many-to-one mapping is more complex. We present a solution in Algorithms 3 and 4.
3. Select services for the new function plan. An executable process can be composed by selecting a service for every function node in the expanded function plan.

Algorithm 3 shows how to identify many-to-one mappings and add to the new function plan in the repair region. For each many-to-one mapping $M = (F, f')$, the function list $F = [f_1, \ldots f_k]$ is a sequential flow of f_1 to f_k in the repair region and can be mapped to a function f'. (We discuss only sequential flows in this paper. More general flow structures to be matched using graph isomorphism algorithms will be reported in the future.) All many-to-one mappings are stored as a trie structure [13] in the Service Repository. The list of F is record by the path of the trie, and the replacement function f' is stored on the terminal node of the corresponding path (Fig. 7).

Algorithm 3 Many-to-one Graph Expansion Algorithm

Input: graph of the original functional plan P
Output: expanded functional graph P_{new}

1: set Stack $ST = \emptyset$, FuncList $L = emptylist$, STreeSet $Y = \emptyset$, $P_{new} = P$
2: set $ST = ST.push(P.start\_node)$
3: **for all** $u \in P$ **do**
4: set $u.notvisited = true$
5: **end for**
6: **while** $ST \neq \emptyset$ **do**
7: set $u = ST.pop()$, $u.notvisited = false$
8: **if** u is a structure node & $L \neq emptylist$ **then**
9: $T = build\_STree(L)$
10: $Y.add(T)$
11: $L = emptylist$
12: **else**
13: $L.append(u)$
14: **end if**
15: **for all** $v \in u.next$ **do**
16: **if** $v.notvisited = true$ **then**
17: $ST.push(v)$
18: **end if**
19: **end for**
20: **end while**
21: $R = Search\_Mto1(Y)$ (see Algorithm 4)
22: **for all** $m_i = [F, f'] \in R$ **do**
23: add a new path for f' in P_{new} as a replacement path of F
24: **end for**
25: return P_{new}

Algorithm 4 Many-to-one Mapping Algorithm: *Search_Mto1()*

Input: suffix tree set $Y = \{T_1, \ldots, T_p\}$
Output: many_to_one mapping set $R = \{m_1, \ldots, m_q\}$.
Require: MT: many_to_one mapping trie.
1: set $R = \emptyset$
2: **for all** $T_i \in Y$ **do**
3: set $r = T_i.root, r.tnode = MT.root$
4: set Queue $Q = \emptyset$, $Q.enqueue(r)$
5: **while** $Q \neq \emptyset$ **do**
6: set $Snode = Q.dequeue(), Tnode = Snode.tnode$
7: **if** $Tnode$ is a terminal node **then**
8: $R.add([Snode.path\_list(), Tnode.mapping])$
9: **end if**
10: **for all** $s \in Snode.children$ **do**
11: **if** $s.in\_edge = t.value$ where $t \in Tnode.children$ **then**
12: set $s.tnode = t$, $Q.enqueue(s)$
13: **end if**
14: **end for**
15: **end while**
16: **end for**
17: return R

In Algorithm 3, we first identify instances of sequential flows from the original functional plan by checking and removing structure nodes (line 6-20). A suffix tree [14] is then built for each sequential flow instance (line 9). Every path in the suffix tree has a leaf node that records the position of the first function node that is matched in the path. The suffix trees are passed to Algorithm 4 to identify all many-to-one mappings (line 21).

In Algorithm 4, each suffix tree is visited using breadth-first search. It looks for a matching node in the trie for every node in the suffix tree. If there is no node in the trie that can match the current node in the suffix tree, it will stop visiting the children of the current suffix tree node. If a terminal node in the trie is visited, then a many-to-one mapping is found. After all possible many-to-one mappings are identified, new paths with the replacement functions will be inserted into the repair region for possible replacement actions.

An example suffix tree and a trie are shown in Fig. 7. The suffix tree is generated for the sequence of functions $[f_1, f_2, f_3, f_2, f_1]$. The trie is for five many-to-one mappings as shown. Using Algorithm 4, two matches will be found, [[f_1, f_2, f_3], Y] and [[f_2, f_3], Z].

Figure 3 shows an example of functional plan expansion. The functional plan initially has only four nodes and one path. After searching for related functions, we find that function F_1 can be mapped to function list [F_5, F_6]; function F_2 can be mapped to function F_7 and function list [F_3, F_4] may be mapped to F_8. By adding the replacement functions, the functional plan is expanded into a plan with eight nodes and four possible paths. Expanding a functional plan increases the likelihood to successfully recompose a process to meet QoS constraints.

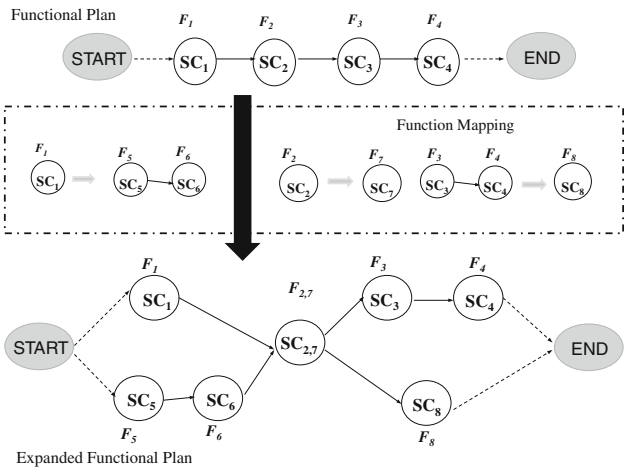

Fig. 7 Expanded functional plan

4.4 Algorithm complexity

Suppose there are n services and p of them are faulty in a process. In Algorithm 1, the first loop will be run p times. Inside the loop, a new service is to be selected according to the QoS constraints, with a linear time complexity on the number of service candidates m. Therefore, the first loop has a complexity of $O(pm)$. In the second loop, the range bound d increases in every round. Suppose for round i, region size is v_i. As discussed in the next paragraph, the complexity of line 11 (Algorithm 2) is $O(pv_i^2)$. On line 13, to calculate the constraint of response time equals to finding the longest path in a DAG, so the complexity is $O(v_i^2 \lg v_i)$. Composition includes function mapping and service candidate selection. Integer programming used for candidate selection has a high complexity that is an exponential function of the region size; we denote the complexity as $IP(x)$. On the other hand, the worst case complexity of function mapping is polynomial. So the complexity of line 14 is $IP(v_i)$, which is the most critical step in Algorithm 1. The maximum size for v_i is cn and the number of regions with the size cn is no greater than $min(p, 1/c)$. So the complexity of Algorithm 1 is $O(min(p, 1/c)IP(cn))$. In comparison, the complexity of recomposing the whole process is $O(IP(n))$. Therefore, our region-based service reconfiguration approach is more efficient than recomposing the whole process if there is a small number of faulty services p and a small threshold c.

In Algorithm 2, the first loop (lines 1-14) goes through all regions and can be run for at most p times. For each region, nodes outside of the region will be checked (line 3), so the inner loop may be run at most n times. The complexity of calculating all regions is $O(np)$ (lines 1–14). The complexity of regions union (lines 15–17) can be done $O(pn^2)$ (or possibly $O(pn \lg n)$ if we use a good data structure). Therefore, the complexity of Algorithm 2 is at most $O(pn^2)$.

In Algorithm 3, suppose there are f nodes in the functional plan. The complexity of initialization (lines 1–5) is $O(f)$. To search for sequential flows (lines 6–20), every node in the functional plan is visited only once; so the complexity of identifying sequential flows is $O(f)$. The complexity of building a suffix tree for a flow with k nodes is $O(k^2)$. The sum of all flows' node number is less then f, so the complexity of building suffix trees (line 9) is $O(f^2)$.

In Algorithm 4, suppose the number of nodes in tree t_i is n_i. The complexity of Algorithm 4 is $O(\sum_i n_i)$. Since the suffix tree size for a k-node-flow is $O(k^2)$, the complexity of many-to-one mapping search (line 21 in Algorithm 3) is $O(f^2)$. Suppose x many-to-one mappings are found, the complexity of adding new paths (lines 22–24) is $O(x)$. In summary, the worst case complexity of functional plan expansion by many-to-one mapping (Algorithms 3 and 4) is $O(f^2 + x)$.

5 System support for reconfiguration

As introduced in Sect. 2, we have implemented the Llama middleware to support accountable SOA. In this section, the design of the Adaptation Manager in Llama is presented.

In our project, a service process is specified in a language called BPELQ. BPELQ defines both the structural flow and QoS information for the service process. In BPELQ, a node can be a service node or a control flow node, such as AND-joint or -split. For service nodes, their QoS attributes and service classes are defined.

Figure 8 shows the interactions for service process reconfiguration in the Llama system. Four main components are involved in the reconfiguration: Accountability Authority (AA), Adaptation Manager, QBroker and Llama ESB.

AA is responsible for identifying services that cause a performance problem in a service process. Since only part of the service process is monitored at run-time, AA performs cause diagnosis using Bayesian network diagnosis [10] or Dependency Matrix–based diagnosis [15]. The diagnosis results are then sent to the Adaption Manager for process reconfiguration.

Adaptation Manager is in charge of reconfiguration region identification and reconfiguration execution decision. Adaptation Manager receives the reconfiguration request (with BPELQ and faulty services) from AA and identifies reconfiguration regions and calculates the QoS constraints for the regions. The regions and their constraints are sent to QBroker for selecting individual services for each function.

When Adaptation Manager receives the results from QBroker, it will decide the reconfiguration strategy for the whole process:

- If no multiple services are replaced by a single service, the changes in the reconfiguration can be implemented using the Routing Table on ESB.
- If multiple services are to be replaced by a single service, the reconfiguration information will be used to generate a new BPEL for the whole process and re-deploy it on BPEL engine.

As discussed in Sect. 2, QBroker provides process planning, service selection as well as BPEL generation for a service process. For reconfiguration, QBroker recomposes sub-processes and generates new BPEL files.

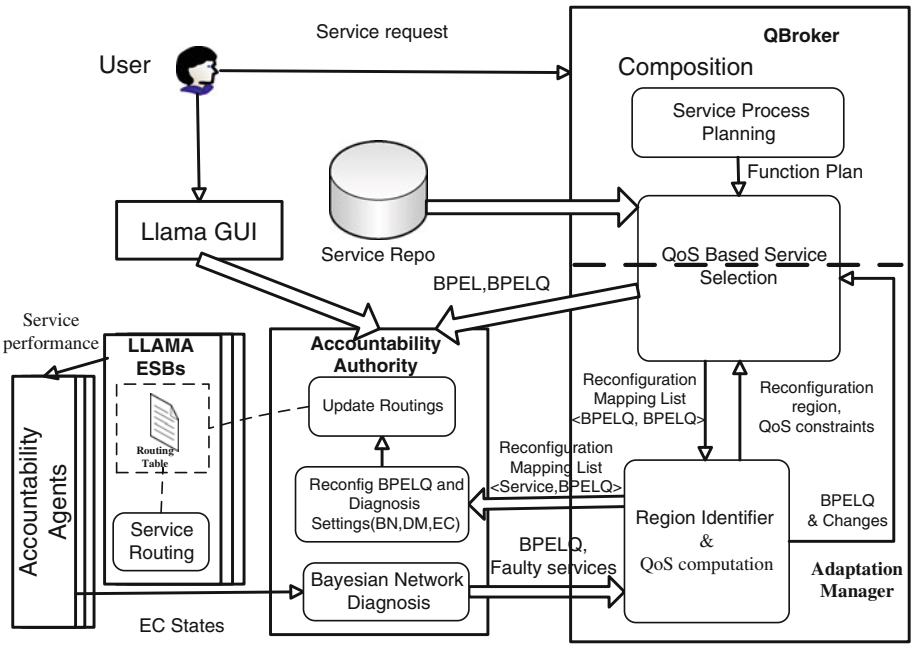

Fig. 8 Reconfiguration in Llama

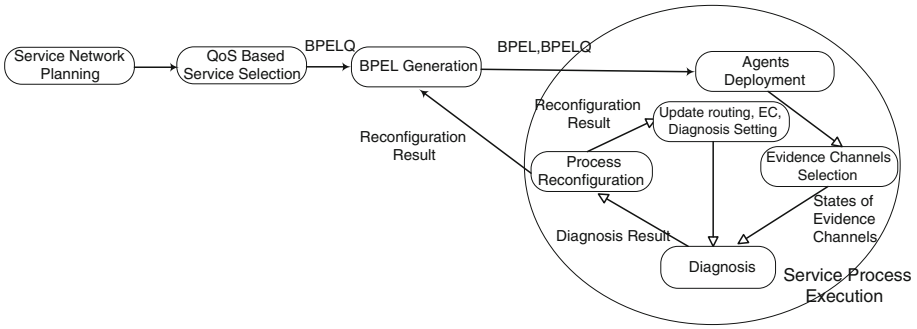

Fig. 9 Workflow for reconfiguration

For one-to-one and one-to-many mappings, the reconfiguration results are stored in the Routing Table on Llama ESB (Fig. 8). Every element in Routing Table records the process ID, service information (ID in process and URL) and a replacement URL for the service. During a service invocation on Llama, the communication interceptor is invoked to check whether the service is in the Routing Table or not. If the service is in the Routing Table, the replacement URL will be invoked instead of the original service. In this way, the process reconfiguration can be easily realized on the Llama ESB.

The reconfiguration workflow is shown in Fig. 9. Reconfiguration in Llama has two workflow loops, a big loop and a small loop. The big loop includes five steps: diagnosis, process reconfiguration, BPEL generation, agents deployment and evidence channel selection. The small loop includes three steps: diagnosis, process reconfiguration and updating routing table, evidence channel and diagnosis settings. If the reconfiguration result includes mapping multiple services to a single service, the big loop reconfiguration will be invoked; otherwise, the small loop reconfiguration will be applied.

6 Performance study

To study the performance of our proposed reconfiguration algorithms, we use a business process with 49 nodes and 8 parallel structures for simulation. Figure 10 shows the process structure. A function repository with 120 functions was also created. For every function in the repository, 1–3 input and output data types and 1–10 service candidates with four QoS values were randomly assigned. About 30% of the functions in the repository can be substituted by other functions or function sets. The simulation steps are as follows.

1. Compose an executable service process by selecting services from the repository in order to meet the end-to-end QoS constraints [4];
2. Randomly select 3 services in the process to be faulty services;
3. Use our algorithms to reconfigure the process to meet the QoS constraints.

Table 1 shows the comparison of reconfiguration performance among region-based repair with function plan expansion (noted with 'Y'), without function plan expansion (noted with 'N'), and the total process recomposition time without repair. Three sets of performance data are shown: the longest distance in reconfiguration regions (showing the number of rounds on region expansion), the total number of services in reconfiguration regions, the response time (in *ms*) for reconfiguration.

From Table 1, we can see that the response time for region-based repairs are all much shorter than the recomposition response time. As discussed in Sec. 4.4, the region-based repair complexity is a function of the size of the largest region. In our simulation, the region size (number of services) is much less than half of the process size. Most of the region sizes are less than 1/3 of the whole service process, which makes the repair to be much faster than recomposing the whole process.

Comparison between repairs with functional plan expansion and without functional plan expansion shows that reconfigurations with functional plan expansion involve a smaller number of functions than without functional plan expansion. Moreover, for some test cases, like test case 2, reconfiguration with functional plan expansion can successfully find a process within 166 ms by only changing 8 nodes, when the other two strategies fail. This is because functional plan expansion can bring more possible paths and more service

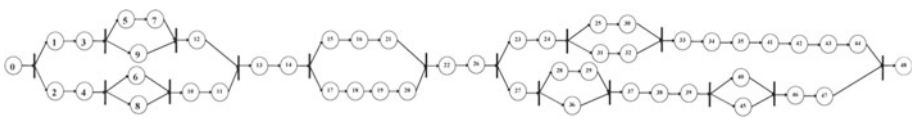

Fig. 10 The process flow for experiments

Table 1 Reconfiguration performance

Test case	1	2	3	4	5	6	7	8	9	10
Max region distance (Y)	4	3	8	Fail	1	3	4	3	3	5
Max region distance (N)	5	Fail	8	Fail	1	4	7	3	3	6
Services affected (Y)	11	8	11	Fail	6	13	19	11	14	10
Services affected (N)	24	Fail	19	Fail	6	16	28	11	14	18
Reconfiguration time (Y)	156	166	366	NA	51	141	169	73	83	206
Reconfiguration time (N)	147	NA	124	NA	51	95	188	68	81	124
Recomposition time	2,809	NA	3,617	NA	1,509	2,688	3,278	1,986	2,301	3,398

candidates to reconfiguration. For example, when a function with only one service candidate fails in the process and no other service can replace this service, recomposition without functional plan expansion will fail. But functional plan expansion may find other functions that can substitute this function and select proper service candidates for those functions to compose a new executable process. However, since the action of functional plan expansion needs time, in some test cases ,the response time for the reconfiguration with functional plan expansion is a little longer than that of reconfiguration without functional plan expansion.

In summary, the response time for both repair strategies (with or without functional plan expansion) is similar, but the number of affected functions is significantly smaller when using functional plan expansion. Also, in some test cases, functional plan expansion can help find a solution when the other repair strategies fail. Considering all above observations, the overall performance of region-based repair with functional plan expansion is the most desirable reconfiguration strategy.

7 Related work

SOA systems are often executed in dynamic environments. Therefore, service process adaptation from faulty performances is an important area of research. The issue of meeting end-to-end QoS constraints on adaptive service processes has been one of the most challenging.

Many projects have studied the adaptation in SOA but without considering QoS. For example, Verma et al. [16] introduce a suite of stochastic optimization–based methods, including both centralized and decentralized methods for adapting business process modeled as Markov decision processes. Both exogenous events and inter-service constraints have been taken into account when performing the adaptation. In [17], alternate plans are pre-specified at the logical level, the physical level, and the run-time level. Depending on the type of changes in the environment, alternative plans from these three levels are selected. While capable of adapting to different events, certain pre-specified plans may not be feasible, making the approach inefficient. And there is no guarantee on the optimality of resulting service processes.

He et al. [18] present an approach to the adaptation of Web service composition based on workflow patterns. This approach measures the value of changed information (VOC), and the cost that updated services may potentially introduce in the business process. The adaptation will be performed within a certain scope defined by workflow patterns when it is expected to pay off. Again end-to-end QoS constraints have not been considered. In [19], Mei et al. present a dependable architecture and reflective middleware, called PKUAS, to support their dependable adaptation. Our Llama middleware is also used to monitor run-time information, investigate faults and route service after reconfiguration. Compared to their research, we focus on multiple end-to-end QoS and adopt integer programming for our reconfiguration, while they did not address QoS.

Much research effort has been conducted on service process composition under QoS constraints. In addition to our work on service composition with end-to-end QoS constraints [4,5], Zeng et al. [6] use a quality-driven approach to select component services for a composite service. They consider multiple QoS attributes, take into account of global constraints, and use the integer linear programming method to solve the service selection problem. Li et al. [20] propose a correlation model–based approach for multi-QoS constrained Web Services selection. The correlation model is established to reduce the search space. Based on the correlation model, a heuristic algorithm is proposed to find a feasible solution for multi-QoS constrained Web services selection. In our work, these composition approaches can be utilized in the sub-process composition part. But all these work have not expanded functional plans by the relationships among functions to introduce more feasible solutions, as we discussed in this paper.

Zeng et al. [21] present another composition framework, in which composition schema are generated incrementally by a rule inference mechanism based on a set of domain-specific business rules enriched with contextual information. A quality-driven composition schema selection strategy is proposed

based on the execution quality and schema quality. Compared to our work, their work focuses on composition schema evolution and selection, whereas our work in this paper mainly addresses service recovery and QoS-aware reconfiguration.

Recent work on SOA management has started to study process adaptation with QoS constraints. The MAIS project [7] has studied how to select an optimal executable business process considering end-to-end QoS constraints. MAIS performs adaptive composition by negotiating with service providers when no feasible solution can be found and triggering re-optimization when the execution plan is probably suboptimal or experiences QoS violation at run-time. The main difference between re-optimization in MAIS and our work is that besides end-to-end QoS constraints, their work focuses more on globally optimal exception plan, while we emphasize more on the efficiency of reconfiguration.

Rouvoy et al. build a middleware, called MUSIC [22], to support self-adaption in ubiquitous and service-oriented environments. Similar to our project, planning-based model is utilized for process reconfiguration in MUSIC. However, MUSIC only considers weighted sum of multiple QoS but not multiple end-to-end QoS constraints for process composition and does not control reconfiguration size during adaptation. For the above two reasons, our approach is more efficient and practical than MUSIC for process reconfiguration in SOA. On the other hand, MUCIS may trigger adaptation due to context changes, which is one of our future research topics.

Vanhatalo et al. [23] introduce a Refined Process Structure Tree (RPST) to re-organize a Business Process Modeling Notation (BPMN), for the purpose of translating BPMN to BPEL. RPST can be also applied in other use cases, like identifying sub-process. BPMN might be used in our region extension part to achieve similar result with our Algorithm 2. For example, the current sub-process' parent in RPST can be identified as the extension result for the current sub-process. But in this way, the extension speed might be much quicker than our current approach. The idea of RPST could be modified and utilized in our future work.

Finally, Athanasopoulos et al. [24,25] present a formal framework to classify services into classes and substitute services by others in the same class. The idea of substituting service in their research is focused on how to classify services and fundamental object-oriented design of services. Their concern is on the service level while our study is on the process level. Their work of substitution classification can be used as a component of our overall recomposition framework.

8 Conclusion

SOA systems should recover from dynamic service faults as soon and as efficiently as possible. It is undesirable to abort and to recompose a large service process if there are only a few services at fault. This paper presents an efficient services reconfiguration solution for SOA with end-to-end QoS constraints. Due to the high computational complexity of services composition when there are a large number of services in a process, a region-based reconfiguration algorithm can be used to greatly reduce the recomposition complexity. Our experimental study confirms that most of the recovery can be successfully achieved by reconfiguring only a small number of services in a service process. We believe this is a promising approach to make SOA systems adaptive to QoS faults.

References

1. Bichler M, Lin K (2006) Service-oriented computing. IEEE Comput 39(3):99–101
2. Papazoglou MP, Traverso P, Dustdar S, Leymann F (2007) Service-oriented computing: state of the art and research challenges. IEEE Comput 40:38–45
3. OASIS (2007) Web services business process execution language version 2.0. OASIS Stand 11. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
4. Yu T, Lin K (2005) Service selection algorithms for web services with end-to-end QoS constraints. Inform Syst E-Bus Manage 3(2):103–126
5. Yu T, Zhang Y, Lin K (2007) Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans Web 6
6. Zeng L, Benatallah B, Ngu A, Dumas M, Kalagnanam J, Chang H (2004) QoS-aware middleware for web services composition. IEEE Trans Softw Eng 30(5):311–327
7. Ardagna D, Pernici B (2007) Adaptive service composition in flexible processes. IEEE Trans Softw Eng 33(6):369–384
8. Lin K, Zhang J, Zhai Y (2009) An efficient approach for service process reconfiguration in SOA with end-to-end QoS constraints. In: Proceedings of IEEE international conference on e-commerce technology(CEC)
9. Panahi M, Lin K, Zhang Y, Chang S, Zhang J, Varela L (2008) The Llama middleware support for accountable service-oriented architecture. In: Proceedings of 6th international conference service-oriented computing (ICSOC' 08), pp 180–194
10. Zhang Y, Lin K, Hsu J (2007) Accountability monitoring and reasoning in service-oriented architectures. J Serv Oriented Comput Appl 1(1):35–50
11. Lin K, Panahi M, Zhang Y, Zhang J, Chang S (2009) Building accountability middleware to support dependable SOA. IEEE Internet Comput 13:16–25
12. Yu T, Lin K (2005) Adaptive algorithms for finding replacement services in autonomic distributed business processes. In: Proceedings of the 7th international symposium on autonomous decentralized systems (ISADS2005), pp 427–434
13. Knuth D (1997) In: The art of computer programming, 2nd edn, vol 3: sorting and searching. Addison-Wesley, chap 6, sect 3, p 492
14. McCreight EM (1976) A space-economical suffix tree construction algorithm. J ACM (JACM) 23(2):262–272
15. Zhang J, Chang Y, Lin K (2009) A dependency matrix based for framework for QoS diagnosis in SOA. In: Proceedings of IEEE

international conference on service-oriented computing and applications(SOCA)
16. Verma K, Doshi P, Gomadam K, Miller J, Sheth A (2006) Optimal adaptation in web processes with coordination constraints. In: Proceedings of the IEEE international conference on web services (ICWS06), pp 257–264
17. Chafle G, Dasgupta K, Kumar A, Mittal S, Srivastava B (2006) Adaptation in web service composition and execution. In: Proceedings of IEEE international conference on web services (ICWS)
18. He Q, Yan J, Jin H, Yang Y (2008) Adaptation of web service composition based on workflow patterns. In: Proceedings of international conference on service-oriented computing (ICSOC)
19. Mei H, Huang G, Wei-Tek Tsai W (2005) Towards self-healing systems via dependable architecture and reflective middleware. In: IEEE international workshop on object-oriented real-time dependable systems, pp 337–346
20. Li L, Wei J, Huang T (2007) High performance approach for multi-QoS constrained web services selection. In: Proceedings of international conference on service-oriented computing (ICSOC)
21. Zeng L, Ngu A, Benatallah B, Podorozhny R, Lei H (2008) Dynamic composition and optimization of web services. Distrib Parallel Databases 24(1–3):45–72
22. Rouvoy R, Barone P, Ding Y, Eliassen F, Hallsteinsen S, Lorenzo J, Mamelli A, Scholz U (2009) Music: middleware support for self-adaptation in ubiquitous and service-oriented environments. Softw Eng Self Adap Softw Syst LNCS 5525:164–182
23. Vanhatalo J, Völzer H, Koehler J (2009) The refined process structure tree. Data Knowl Eng 68(9):793–818
24. Athanasopoulos D, Zarras AV, Issarny V (2009) ForeverSOA: Towards the maintenance of service oriented software. In: CSMR workshop on software quality and maintenance (SQM'09)
25. Athanasopoulos D, Zarras AV, Issarny V (2009) Service substitution revisited. In: 24th IEEE/ACM international conference on automated software engineering (ASE'09)