11-2014

Automated runtime recovery for QoS-based service composition

Tian Huat TAN

Manman CHEN

Étienne ANDRÉ

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

See next page for additional authors

Citation

TAN, Tian Huat; CHEN, Manman; ANDRÉ, Étienne; SUN, Jun; LIU, Yang; and DONG, Jin Song. Automated runtime recovery for QoS-based service composition. (2014). *Proceedings of the 23rd International World Wide Web Conference, Seoul, South Korea, 2014 April 7-11*. 563-574.
Available at: https://ink.library.smu.edu.sg/sis_research/4994

Automated Runtime Recovery for QoS-based Service Composition

Tian Huat Tan[3] Manman Chen[1] Étienne André[2] Jun Sun[3] Yang Liu[4]
Jin Song Dong[1]
[3]SUTD (Singapore), [1]NUS (Singapore), [2]Université Paris 13 (France), [4]NTU (Singapore)
{tianhuat_tan, sunjun}@sutd.edu.sg, {chenman, dongjs}@comp.nus.edu.sg
Etienne.Andre@univ-paris13.fr, yangliu@ntu.edu.sg

ABSTRACT

Service composition uses existing service-based applications as components to achieve a business goal. The composite service operates in a highly dynamic environment; hence, it can fail at any time due to the failure of component services. Service composition languages such as BPEL provide a compensation mechanism to rollback the error. But such a compensation mechanism has several issues. For instance, it cannot guarantee the functional properties of the composite service after compensation. In this work, we propose an automated approach based on a genetic algorithm to calculate the recovery plan that could guarantee the satisfaction of functional properties of the composite service after recovery. Given a composite service with large state space, the proposed method does not require exploring the full state space of the composite service; therefore, it allows *efficient* selection of recovery plan. In addition, the selection of recovery plans is based on their quality of service (QoS). A QoS-optimal recovery plan allows *effective* recovery from the state of failure. Our approach has been evaluated on real-world case studies, and has shown promising results.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems; H.3.5 [**Online Information Services**]: Web-based services

Keywords

Web Services; QoS; Service Composition; SOA; Genetic Algorithm

1 Introduction

Service-oriented architecture (SOA) promotes the use of services as building blocks for software applications. This allows enterprise to outsource part of their processes to external services, which produces a lower cost of ownership for the enterprises over time. Services make use of open standards, such as WSDL [11] and SOAP [19], allowing interaction of heterogeneous applications. To utilize a set of services to achieve a business goal, service composition languages such as BPEL (Business Process Execution Language) [5] have been proposed. The service that is composed by service composition is a *composite service*, and services that the composite service makes use of are *component services*.

SOA allows functionalities of the composite services to be distributed to third party service providers; therefore component services are allowed to evolve freely, independently of each other. Component services could behave differently after being modified by service providers, or could fail due to various reasons such as network problems, software bugs, hardware failure, etc. In addition, a composite service (expressed, e.g., in BPEL) uses late binding mechanism, where abstract services are used during design time, and the concrete services would only be decided during runtime. As a result, design-time validation of composite services, such as through testing or static verification, is insufficient. Therefore, runtime monitoring of the functional properties, and being able to recover from properties violations, are essential for the dependability of a composite service.

Composite service languages, such as BPEL, are equipped with constructs to support the *compensation* mechanism. The compensation mechanism is an application-specific way to reverse completed activities. For example, the compensation of making a hotel reservation would be to cancel the reservation. One of the important issues of the current compensation mechanism is that it is uncertain whether the compensation will lead to a system state that could satisfy the functional properties of the composite service.

Existing works [23, 24] address this problem by devising a recovery plan that allows the system to recover from properties violations, based on exploring the state space of the composite service using planning techniques based on SAT-solvers. This approach suffers from several disadvantages. First, the full state space needs to be generated for recovery plans exploration; therefore it might encounter the state explosion problem, especially when dealing with large-scale service composition (see, e.g., [13]). Second, the QoS aspects (e.g., dependability and response time) of the recovery plan are not taken into account explicitly in this approach. An important aspect of a recovery plan is the QoS. A recovery plan with poor QoS is not only ineffective, but also it might result in undesired side effects such as compensation loops, i.e., it leads to failed services and compensate repeatedly. Because a failed service has low dependability, the recovery plan that involves the invocation of the failed services will be filtered away in a selection procedure that is QoS-aware.

In this work, we address this issue by proposing a technique based on genetic algorithms (GA) for searching for a recovery plan. GA are computational methods inspired by the biologic evolution, which have been used to solve a variety of problems (see, e.g., [20]). Traditional GA use fix-length encodings, called chromosomes. However, using chromosomes to encode the recovery plan poses a challenge, as the length of chromosome depends on the size of the state space, which is unknown beforehand. Therefore, an estimation on the chromosome length is necessary. Exact calculation of the length of chromosome by exhaustively exploring all possible states is not feasible, as it obviously leads to the state space explosion. Furthermore, over-approximating the length of chromosome might render GA ineffective, whereas under-approximation might result in the incomplete encoding of recovery plan. In this work, we propose *rGA* (recovery plan GA), to find a near-optimal recovery plan in a large state space. *rGA* addresses the aforementioned problems by adaptively adjusting the length of chromosomes with respect to the size of the state space during the recovery plan searching. Furthermore, *rGA* does not require generating the full state space – it only generates the partial state-space on-the-fly during the exploration. Our contributions are summarized as follows.

1. Novel representation and operations – We propose *rGA*, a novel GA making use of dynamic-length chromosomes to represent the recovery plans, and manipulating them using genetic operators for evolving new recovery plans.

2. On-the-fly state-space exploration – *rGA* does not require the generation of full state space beforehand. State space is generated on-the-fly during recovery plan exploration. Since *rGA* performs guided exploration on the most promising region of the search space for the recovery plans, only partial state space is explored in the end. This improves time and space efficiency.

3. Practical recovery plan generation – *rGA* adopts an enhanced initial population policy, and selects a recovery plan with near-optimal QoS; this enables the effective restoration of correctness for the composite service. Furthermore, *rGA* utilizes runtime information (such as variable value before failure) and the structure of composite service, resulting in a more realistic recovery plan with higher chance of success.

We have evaluated *rGA* with real-world case studies, which demonstrate the effectiveness over existing approaches.

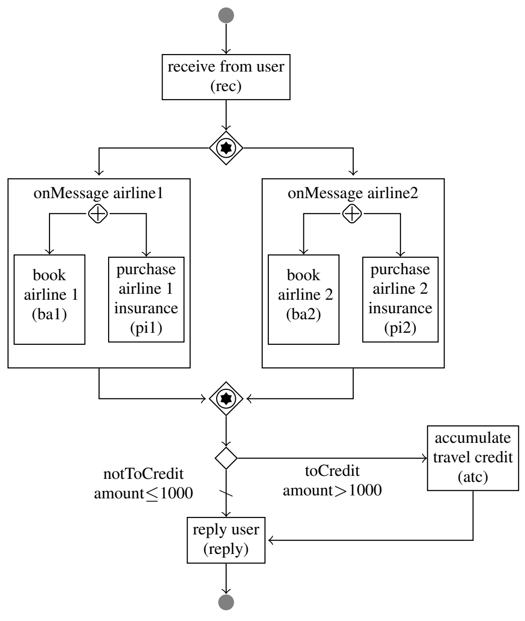
Outline. Section 2 presents a running example. Section 3 introduces the QoS compositional model and the necessary terminology. Section 4 presents *rGA*. Section 5 provides the evaluation of our approach. Section 6 reviews the related work. Finally, Section 7 concludes the paper, and outlines future work.

2 Motivating Example

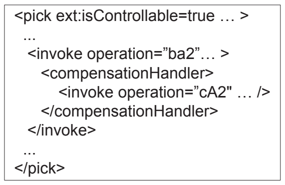
BPEL [5] is a *de-facto* industry standard for implementing composition of existing Web services by specifying an executable workflow using predefined activities. In this work, we assume that composite services are specified using the BPEL language. Basic BPEL activities that communicate with component Web services include `<receive>`, `<invoke>`, and `<reply>`, which are used to receive messages, execute component Web services and return values respectively. In addition, a `<pick>` activity is used to wait for the occurrence of exactly one message from a set of messages.

The control flow of services is defined using activities such as `<sequence>`, `<while>`, and `<if>`, to provide sequential ordering, loop, and conditional structure respectively. BPEL also supports parallel execution of activities by using the `<flow>` activity. A `<scope>` activity is used to contain other activities, and it can be associated with a compensation handler, which specifies activities for compensating the effects of executing the `<scope>` activity. In this work, the `<while>` loops are assumed to be bounded and the loop bound could be estimated using methods like [14].

We consider here a toy example of a Travel Booking Service (TBS), where the goal is to help users to book for the transportation for their travel choice. The workflow of this example is illustrated in Figure 1a. Upon receiving the service request from the user (`rec`), a `<pick>` activity (denoted by ◈) is enabled to wait for exactly one message from two possible messages (`airline1`, `airline2`) provided by the user. If `airline1` message is received, a `<flow>` activity (denoted by ⊕) is invoked: two activities `ba1` and `pi1` are invoked concurrently to book airline 1 and purchase airline 1's insurance respectively. Similar workflow applies when `airline2` message is received. Subsequently, an `<if>` activity (denoted by ◇) is used to check whether the purchased `amount` is larger than 1000. If yes, the `accumulate travel credit` (`atc`) service is invoked to accumulate the travel credit that could be used in the next purchase of the user. In either case, `reply user` (`reply`) is called to return the result of purchases to the user.



(a) Workflow for TBS



```
<pick ext:isControllable=true ... >
  ...
  <invoke operation="ba2"... >
    <compensationHandler>
      <invoke operation="cA2" ... />
    </compensationHandler>
  </invoke>
  ...
</pick>
```

(b) Compensation for book car service

Figure 1: Transport Booking Service (TBS)

Now, let us consider a scenario where the book airline 2 (`ba2`) service is unreachable. Classic recovery strategies may retry it or switch it to an alternating service [7]. We denote such recovery strategy a *point recovery strategy*, as it involves retrying or switching of a particular service. There are cases where such a strategy does not work. For example, `ba2` service could be down, therefore retrying would not work. In addition, there might not exist an alternating service that could be switched directly. In such a case, another important strategy, which we denote as *workflow recovery strategy*, could be used. A flow recovery strategy involves modifying the workflow by backtracking to a previous state, and finding an alternative path for execution. To implement the flow recovery strategy, one needs to devise a *recovery plan* specifying how the compensation should be done, and which alternative path to choose. A good recovery plan also needs to be *QoS-aware*. We give below some of the QoS factors that need to be considered.

1) Cost: What is the cost for compensation, and what is the possible future costs that would be likely to incur in the recovery plan?

2) Dependability: What is the chance of success of the recovery plan?
3) Response time: What is the expected response time of the recovery plan?

These issues will be addressed in the next sections.

3 QoS-aware Compositional Model

In this section, we define the QoS-aware compositional model used in this work. We first give the definition of a composite service.

DEFINITION 1 (COMPOSITE SERVICE). *A composite service $\mathcal{M}$ is a tuple (Var, V_0, P_0), where Var is a finite set of variables, V_0 is an initial valuation that maps each variable to its initial value, and P_0 is the composite service process.*

The semantics of composite service is captured using labeled transition systems (LTSs), as discussed in the following.

3.1 Labeled Transition System

DEFINITION 2 (LABELED TRANSITION SYSTEM (LTS)). *An LTS is a tuple $\mathcal{L} = (S, s_0, \Sigma, \delta)$, where S is a set of states, $s_0 \in S$ is the initial state, Σ is the universal set of actions, and $\delta : S \times \Sigma \times S$ is a transition relation.*

In this work, a *state* s is of the form (V, P), where valuation V is a partial function that maps a variable to its value (in its domain), and process P is a composite service process. Given a composite service (Var, V_0, P_0), a sample valuation V is $(\{var_1 \mapsto 1, var_2 \mapsto \perp\}, P_0)$, where $var_1, var_2 \in Var$. $var_2 \mapsto \perp$ denotes that var_2 is undefined.

In this work, we assume that an error action *Err* (resp. an error state s_{Err}) always exists in Σ (resp. S) of any LTS. The error action *Err* is used to model the error condition (e.g., component service unreachable, functional correctness property violated). The error state s_{Err} is reachable from any state of S via action *Err*, i.e., $\forall s \in S \setminus \{s_{Err}\}, (s, Err, s_{Err}) \in \delta$.

Given an LTS $\mathcal{L} = (S, s_0, \Sigma, \delta)$, we use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in \delta$. Given a state $s \in S$, we denote by $Enable(s)$ the set of states reachable from s by one transition; formally, $Enable(s) = \{s' | s' \in S \wedge a \in \Sigma \wedge s \xrightarrow{a} s' \in \delta\}$. An *execution* π of $\mathcal{L}$ is a finite alternating sequence of states and actions $\langle s_0, a_1, s_1, \ldots, s_{n-1}, a_n, s_n \rangle$, where $\{s_0, \ldots, s_n\} \in S$ and $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$. We denote by $s_0 \xrightarrow{a_1} s_1 \xrightarrow{...} s_{n-1} \xrightarrow{a_n} s_n$ the execution π. The *prefix* of execution π is a fragment of π that starts from state s_0 and ends with a state s_i where $i \leq n$. A *complete execution* is an execution starting in the initial state and ending in a terminal state. A state $s \in S$ is *terminal* if there does not exist a state $s' \in S$ and an action $a \in \Sigma$ such that $s \xrightarrow{a} s' \in \delta$; otherwise, s is *non-terminal*. In addition, we denote the LTS of a BPEL service $\mathcal{M}$ by $L(\mathcal{M})$.

Example: Transport Booking Service

The LTS $L(\text{TBS})$ of the TBS example is shown in Figure 2. The dashed and dotted arrows are not part of the semantics and they will be explained later on. The formal semantics of BPEL activities in this work is based on [16]. For example, consider the conditional activity, $A_{atc} \triangleleft b \triangleright A_{reply}$, that is enabled at state s_9, where the activity A_{atc} is executed when the guard $b = (\texttt{amount} > 1000)$ is evaluated to be true, otherwise the activity A_{reply} is executed. From state $s_9 = (\{\texttt{amount} \mapsto \perp\}, A_{atc} \triangleleft b \triangleright A_{reply})$, it has two possible enabled states, which are $s_{10} = (\{\texttt{amount} \mapsto \perp\}, A_{atc})$, and $s_{11} = (\{\texttt{amount} \mapsto \perp\}, A_{reply})$ respectively. This is denoted in the LTS as a state s_9 with two outgoing transitions to states s_{10}

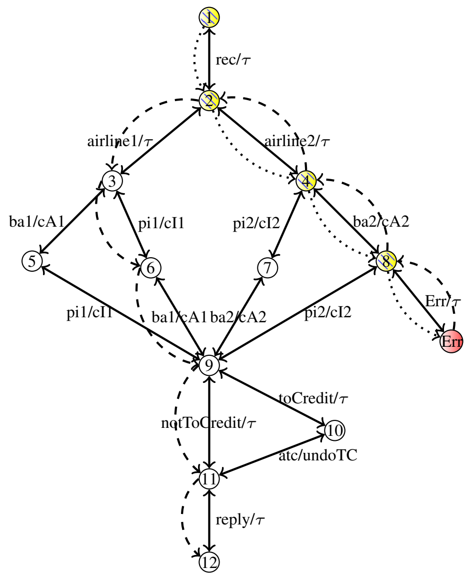


Figure 2: LTS of TBS example

and s_{11}. Noted that if b is defined, b is either false or true; therefore only one branch is explored in the LTS. Similarly, pick activities (`<pick>`) and parallel activities (`<flow>`) are specified using two outgoing arrows to denote all possible execution orders of their child activities. For the sake of readability, the error transitions *Err* from all states (except state s_8) to the error state s_{Err} are not shown in the LTS.

A state is a *migration state* if the state provides alternative choices of execution, i.e., it migrates from the current execution to another one. Migration states include the states where the `<flow>` activity, `<pick>` activity or non-idempotent service invocation is enabled. A service invocation is *idempotent* if any invocation with the same input parameters give the same result. In Figure 2, valid migration states from the state s_8 are states s_4, s_2, and s_1, shown in hatched yellow circles.

3.2 Backward Actions

BPEL supports compensation mechanism [5] as an application-specific way to reverse the activity that has already been completed. The limitation of the default compensation mechanism is that it is difficult to determine the system state after compensation, and therefore it is hard to decide whether it would end up in a system state where the functional properties could be satisfied.

To address this problem, we make an observation that every action of BPEL can make up to two kinds of changes – internal and external changes. Internal changes modify the valuation V of current system state to a different valuation V', while external changes modify the state of component services. External changes could only be made by communication activities, e.g., `<receive>`, `<invoke>`, and `<reply>`, since communication activities are the only activities communicating with component services.

To undo internal changes, the valuation prior to executing for an action is stored as a snapshot valuation; therefore during recovery process, internal changes can be undone by reversing the current valuation V' to the snapshot valuation V automatically. To allow undoing of the external changes, users are required to specify a compensation handler for each communication activity a. For example, in Figure 1b, a compensation handler is specified for `ba2` operation, which compensates the external changes made by `ba2` by invoking the `cA2` operation to cancel the flight that has been booked. As a consequence, for every action a, we have a corresponding *backward action*, a_{bak}, which "goes back" to the state prior to execute action a by reversing the internal changes using

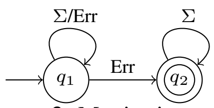


Figure 3: Monitoring automata

the snapshot valuation and external changes with the help of compensation handler. An example of backward action is $s_8 \overset{cA2}{\rightarrow} s_4$ in $L(\text{TBS})$, where the composite service compensates from state $s_8 = (P, V)$ to $s_4 = (P', V')$, by undoing the valuations from V' to snapshot valuation V and at the same time canceling the flight that has been booked. We use τ to denote a backward action that does nothing to compensate. A non-backward action is a *forward action*; an example is $s_4 \overset{ba2}{\rightarrow} s_8$. Given a pair of forward action a_f and backward action a_b, where $s \overset{a_f}{\rightarrow} s'$ and $s' \overset{a_b}{\rightarrow} s$, we combine them as single notation $s \overset{a_f/a_b}{\leftrightarrow} s'$, e.g., $s_4 \overset{ba2/cA2}{\leftrightarrow} s_8$. We use Σ_F and Σ_B to denote the set of all possible forward actions and backward actions respectively.

3.3 Monitoring Automata

In this section, we introduce how the functional properties are represented and verified. The functional properties are represented using deterministic finite automata (DFA), called here *monitoring automata*. Formally:

Definition 3. *A* monitoring automaton $\mathcal{A}$ *is* $(Q, Q_0, \Sigma, \delta, F)$, *where Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, Σ is the universal set of actions, $\delta : Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is a set of accepting states.*

We use Σ^* to denote a set of finite sequences of actions. Given a monitoring automaton $\mathcal{A}$, a sequence of actions $a_1 a_2 \dots a_n \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if there exists a path in $\mathcal{A}$ of the form $q_0 \overset{a_1}{\rightarrow} q_1 \overset{\dots}{\rightarrow} q_{n-1} \overset{a_n}{\rightarrow} q_n$, where $q_0 \in Q_0$, $q_n \in F$, $a_i \in \Sigma$ and $\forall 1 \leq i \leq n, (q_{i-1}, a_i, q_i) \in \delta$. An execution $\pi = s \overset{a_1}{\rightarrow} s_1 \overset{\dots}{\rightarrow} s_{n-1} \overset{a_n}{\rightarrow} s_n$ is *accepted* by $\mathcal{A}$ if the sequence of actions $a_1 a_2 \dots a_n$ is accepted by $\mathcal{A}$; otherwise it is *rejected* by $\mathcal{A}$. We denote the set of accepted sequences of actions as $L(\mathcal{A})$.

Given a functional property P_s, $\mathcal{A}(P_s)$ denotes its monitoring automaton. An execution is accepted by $\mathcal{A}(P_s)$ if it violates the property P_s (otherwise it conforms to P_s). For example given a functional property P_1 "Unreachability of component service can never happen in TBS", where error action `Err` is triggered when the component service is unreachable. The monitoring automata for functional property P_1 is shown in Figure 3. Given a set of properties, we define the monitoring automata of a composite service CS as $\mathcal{M}_{CS} = \langle \mathcal{A}(P_1), \dots, \mathcal{A}(P_N) \rangle$, where P_i is a functional property (for $1 \leq i \leq N$). Given an execution π in $L(CS)$, π *satisfies* $\mathcal{M}_{CS}$, denoted as $\pi \models \mathcal{M}_{CS}$, if π is rejected by all automata $\mathcal{A} \in \mathcal{M}_{CS}$. Otherwise, π *violates* $\mathcal{M}_{CS}$, denoted by $\pi \not\models \mathcal{M}_{CS}$.

3.4 Recovery Plan

Consider again the LTS $L(\text{TBS})$ in Figure 2. An execution starts from state s_1 to state s_8 as shown using dotted arrow ($\cdots\rangle$). At state s_4, the `ba2` service is invoked and subsequently evolves into state s_8. Since the `ba2` service is unreachable and timeout by the BPEL runtime engine, this could lead the system to the error state s_{Err}. However, the service monitor discovers the anomalies, and interferes the current process. To recover from the error, a recovery plan is calculated. A recovery plan is a guideline of execution that is used to compensate the current error (using backward actions) to a migration state, and choose an alternative path that could lead to the terminal state (using forward actions). In TBS, a possible recovery plan r is to compensate from state s_{Err} to migration state s_2 using backward actions, and go forward from state s_2 to state s_{12}. The recovery plan of TBS, denoted by r_{TBS}, is shown using dashed arrow ($--\rangle$).

Definition 4. *A* recovery plan r *is an execution* $s_{\text{Err}} \overset{a_1}{\rightarrow} s_1 \overset{a_2}{\rightarrow} \dots \overset{a_m}{\rightarrow} s_m \overset{a_{m+1}}{\rightarrow} s_{m+1} \overset{a_{m+2}}{\rightarrow} \dots \overset{a_n}{\rightarrow} s_n$ *where s_n is a terminal state, s_m is a migration state with $0 \leq m \leq n$, $\forall j \leqslant m$, $a_j \in \Sigma_B$, and $\forall\, k > m$, $a_k \in \Sigma_F$.*

A *prefix* of the recovery plan r is a fragment of execution of r that starts with s_{Err} and ends with s_i where $i \leq n$. Sometimes, we also use the term *partial recovery plan* to denote a prefix of a recovery plan. A *suffix* of a recovery plan r is the fragment execution of r that starts with any state in the execution, and ends with terminal state s_n.

Controllability of a recovery plan. Consider the recovery plan r_{TBS} for TBS. At migration state s_2, according to the recovery plan r_{TBS}, it needs to proceed to state s_3. However, the semantics of the `<pick>` activity chooses which branch to execute depending on the messages (`airline1` or `airline2`) that are received from the user. In such case, it is a violation of semantics if we follow the recovery plan. Therefore, we extend the `<pick>` activity with an attribute `isControllable` by using BPEL extension attribute [5] feature, so that users are allowed to specify which activities are controllable by the recovery module. In our example, the `<pick>` activity that is activated at state s_2 is specified to be controllable, by setting the `isControllable` attribute to true (see Figure 1b). Since the `<pick>` activity is specified as controllable, the activity would follow the recovery plan. Besides the `<pick>` activity, the user also needs to specify the controllability of the `<flow>` and `<if>` activities. If the `<flow>` activity is set to be controllable, then the runtime engine would disregard the concurrent semantics of the `<flow>`, and follow the recovery plan using sequential semantics. If the `<if>` is set to be controllable, then the runtime engine would disregard the valuation of guard condition and execute the branches that are chosen by the recovery plan. Suppose that the `isControllable` of the `<flow>` activity that is enabled at state s_3 and the `<if>` activity that is enabled at state s_9 is set to be true and false respectively. In this case, the recovery process would proceed until state s_9. At state s_9, since the `<if>` activity is uncontrollable, the recovery process ends, and normal execution proceeds. During normal execution, the `<if>` activity will decide to enter state s_{10} or s_{11} depending on the value of `amount`.

We call the maximal controllable portion of an execution its *controllable prefix*. For the case of TBS, the controllable prefix is from state s_{Err} to state s_9. We say that state s_9 is an *uncontrollable state*, which is a state that puts an end to the controllable prefix. Similarly, we denote the portion of an execution starting from uncontrollable state as its *uncontrollable suffix*. Although the uncontrollable suffix (i.e., from state s_9 to state s_{12}) is not executed as part of the recovery process, but it provides an insight on the executions that starts from uncontrollable state. During the calculation of recovery plan, the uncontrollable suffix could help to find a recovery plan that ends up in an uncontrollable state that has a better executions starting from it. Therefore, the composite service has higher chance to conform with both functional and non-functional requirement when recovery process ends and normal execution starts.

4 Service Recovery as a GA Problem

Our work is based on genetic algorithms (GA) for calculation of the recovery plan.

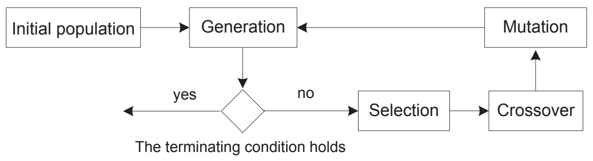


Figure 4: Typical flow of genetic algorithms

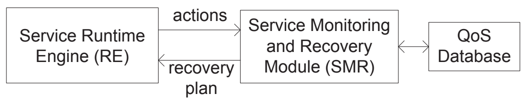

Service Runtime Engine (RE)
actions
recovery plan
Service Monitoring and Recovery Module (SMR)
QoS Database

Figure 5: Service monitoring and recovery framework

4.1 Preliminaries of Genetic Algorithms

GA [15] are stochastic search methods based on principles of biological evolution, inspired by the "survival of the fittest" principle of the Darwinian theory of natural evolution. GA encode a potential solution to a specific problem using a simple chromosome-like data structure, and apply genetic operators to these structures in such a way to preserve critical information. GA are typically suited for optimization problems where the problem space is large and complex.

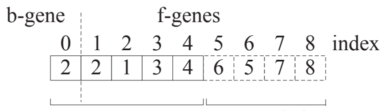
Figure 4 introduces a typical workflow of GA. A GA begins with an (typically random) initial generation of chromosomes, which we call it *initial population*. Genetic operators, such as *selection*, *crossover*, and *mutation*, are applied on a generation, to evolve the next generation of chromosomes. Genetic operators operate based on the *fitness* of chromosomes – the highly-fit chromosomes have higher chance to be evolved into the next generation. The fitness of chromosomes is typically quantified by the *fitness value* of the chromosome. The evolution continues until the terminating condition. An example of the terminating condition could be that the number of generations exceeds a predefined upper bound $n \in \mathbb{Z}^{>0}$.

We name our GA-based approach, *rGA*. To support on-the-fly partial exploration of state-space in *rGA*, the recovery plan is encoded in dynamic-length chromosome, in contrast to the typical fix-length chromosome. The details of encoding will be provided in Section 4.3. Subsequently, we introduce the genetics operators that manipulate the chromosomes in Section 4.4, and demonstrate how the fitness value of a chromosome is calculated in Section 4.5. To allow fast convergence of *rGA*, we propose an enhanced initial population policy, as explained in Section 4.6. In the following, we discuss the architecture of the service monitoring and recovery framework that is used in this work.

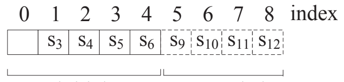
4.2 Architecture

The architecture of our work is shown in Figure 5. The service runtime engine (RE) is an environment used to execute the BPEL composite services; here, we are using ApacheODE [3], an open-source runtime engine for BPEL composite services. The Service Monitoring and Recovery Module (SMR) contains the monitoring automata, $\mathcal{M}_{CS}$, of the composite service *CS* that is executing in the RE. During the execution of *CS*, the SMR intercepts the actions from the RE. The intercepted actions are used to update the states of all monitoring automata $m_i \in \mathcal{M}_{CS}$ that are stored in the SMR, and these actions will also be recorded as part of the execution π_{CS} for the composite service *CS*. In addition, after the RE communicating with a component service S, the SMR will update the QoS database with the latest QoS information (e.g., response time and availability) of component service S.

By checking the status of each monitoring automata $m_i \in \mathcal{M}_{CS}$, the SMR could detect whether the functional properties of *CS* are violated. If so, service recovery is initiated to calculate the recovery plan. The recovery plan will be calculated based on the execution π_{CS}, and estimated QoS attributes from QoS database. Subsequently, the recovery plan would be returned to RE and RE would resume with the recovery according to the recovery plan. The details of the calculation of recovery plan will be introduced in the rest of this section.



(a) Chromosome for recovery plan r_{TBS}



(b) Global state array

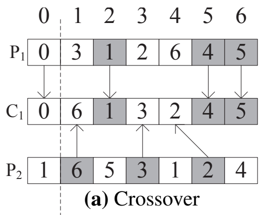
Figure 6: Genetic encoding of recovery plan

4.3 Genetic Encoding of a Recovery Plan

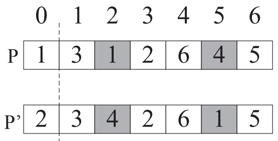
We now introduce the representation of recovery plans as chromosomes. The technical challenge when developing the representation is that classic GA use fixed-size chromosomes, while the recovery plan lying within an LTS has an unknown number of states and transitions. Providing a unique representation of a recovery plan requires an exhaustive exploration of the LTS in order to know the chromosome length required to encode the recovery plan; this might encounter the infamous state-explosion problem. In order to address this problem, we propose *dynamic-length chromosomes* to encode the recovery plan, where the length of chromosomes is adjusted adaptively during the (partial) exploration of the LTS for the optimal recovery plan.

We adopt here array-based chromosomes. The chromosome in Figure 6a represents the recovery plan r_{TBS}. Given a chromosome of length n, array indices are numbered from 0 to $n-1$. A gene is an element of the array, and the value of a gene ranges over non-negative integer number. Given a recovery plan, there are two parts: backward execution and forward execution. The backward execution contains only backward actions, followed by the forward execution that contains only forward actions. Similarly, the genes are divided into two parts: a b-gene (backward-gene) and a set of f-genes (forward-genes), to represent the backward execution and forward execution respectively. The b-gene is located at index 0 of the chromosome, and f-genes are located from indices 1 to $n - 1$. We demonstrate genetic encoding of recovery plan using the example shown in Figure 2. The value of b-gene shows the number of backward actions are used to compensate. Assume the value of the b-gene is 3: compensating three steps from error state s_{Err} would reach the migration state s_2.

After compensation, we consider forward actions. The forward execution is encoded differently from the backward execution. To encode the forward execution, we need to make use of a global state array (see Figure 6b) which is shared by all chromosomes. Intuitively, the values in f-genes give the *priority values* of the states in state-array. We introduce f-genes and state-array using the recovery plan r_{TBS} that has been compensated to migration state s_2 according to the value of b-gene. Initially, f-genes and the state array are empty. From migration state s_2, two states s_3 and s_4 are enabled, calculated by *EnableStates*(s_2). Since these two states do not exist in the state array, they are added to the array at indices 1 and 2 (index 0 is always left empty because of b-gene). At the same time, assume two values, 2 and 1 are added to f-genes. Details on how these values are decided will be given in Section 4.6. Values in f-genes represent the *priority values* of the states in state-array at



(a) Crossover



(b) Mutation

Figure 7: Genetic operations

```
Algorithm 1: Crossover
   input  : Chromosomes $P_1$, $P_2$
   output: Chromosomes $C_1$, $C_2$
1 $C_1 \leftarrow P_1$ ;  $C_2 \leftarrow P_2$;
2 if $rand(0,1) \leq P_{cross}$ then $\langle C_1, C_2 \rangle \leftarrow pCrossOver(C_1, C_2)$;
3 return $\langle C_1, C_2 \rangle$;
```

the same index – states s_3 and s_4 have priority values of 2 and 1 respectively. Since state s_3 has higher priority than state s_4 ($2 > 1$), state s_3 is chosen to be the next state in the recovery plan. This process goes until state s_6. At this point, both the state array and chromosome are *dynamically expanded* to contain additional states and their priority values.

4.4 Genetic Operators

GA make use of crossover and mutation to create new chromosomes for the next generation of a population. We introduce these two operators, adapted from [17].

Crossover. We make use in *rGA* of a position-based crossover operator. Two new chromosomes are produced from each crossover operation. The algorithm of crossover operator is shown in Algorithm 1. At line 2, $rand(0,1)$ randomly chooses a real number between 0 and 1. If the number is less than P_{cross} then it performs the positional crossover *pCrossOver* (line 2) on chromosomes P_1 and P_2.

We illustrate the positional crossover *pCrossOver*, using an example shown in Figure 7a. In Figure 7a, a new chromosome C_1 is produced by applying positional crossover *pCrossOver* to chromosomes P_1 and P_2. The b-gene of C_1 is created by choosing the b-gene from P_1 or P_2 randomly. The f-genes of C_1 are produced by taking some f-genes from P_1 at random positions; in the example, we take f-genes at positions 2, 5, and 6 from P_1. Subsequently, the empty positions of C_1, viz., positions 1, 3, and 4, are filled up by performing left-to-right scan on P_2, and the unused numbers will be used to fill in the empty positions. The production of another new chromosome C_2 (not shown in the graph) is symmetric to the production of C_1. For the b-gene, the crossover operator chooses from P_2, as P_1 has been chosen by C_1. Subsequently, it takes the f-genes positions 2, 5, and 6 from P_2, and fills in the empty position by performing left-to-right scan on P_1. The resulting chromosome of C_2 is $\langle 1, 3, 5, 1, 6, 2, 4 \rangle$.

If the number is greater than P_{cross}, it simply returns the chromosomes P_1, and P_2 at line 3.

Mutation. The swap-based mutation operator is used for the mutation operation. The algorithm of mutation operator is given in Algorithm 2. At line 4, *getBackwardSteps*() returns the set of numbers for b-gene that could lead to a migration state, and the *rand* function chooses one of them randomly. At line 7, the value of the gene at position i is randomly swapped with a gene from position 1 to $n-1$.

Figure 7b shows how a new chromosome P' is produced by applying the mutation operator to chromosome P. The b-gene is mutated by randomly picking a number that could compensate to a

```
Algorithm 2: Mutation
   input  : Chromosome $P$
   output: Chromosome $C$
1 $C \leftarrow P$;
2 $n \leftarrow |P|$;
3 if $rand(0,1) \leq P_{mut}$ then
4     $C[0] \leftarrow rand(getBackwardSteps())$;   // for b-gene
5 for $i = 1$ to $n-1$ do
6     if $rand(0,1) \leq P_{mut}$ then
7         $swap(C[i], C[randInt(1, n-1)])$;   // for f-genes
8 return $C$;
```

migration state. For f-genes, two genes are chosen randomly and their values are swapped.

4.5 Calculating the Fitness Value

4.5.1 QoS Optimality

In this work, we focus on quantitative QoS attributes that can be quantitatively measured using metrics. There are two classes of attributes, namely positive ones (e.g., availability) and negative ones (e.g., response time). Positive attributes have a positive effect on the QoS, and therefore need to be maximized. Conversely, negative attributes need to be minimized. For simplicity, we only consider negative attributes in this work, since positive attributes can be transformed into negative attributes by multiplying their value with -1. Given n QoS attributes of a service s, we use an attribute vector $Q_s = \langle q_1(s), \ldots, q_r(s) \rangle$ to represent it, where $q_i(s)$ is the ith QoS attributes of Q_s.

A composite service S makes use of a finite number of component services to accomplish a task. Let $C = \{s_1, \ldots, s_n\}$ be the set of all component services that are used by S. The composite service communicates with component services using the communication activities, which includes `<invoke>` and `<onMessage>` activities. Given an action a belonging to the communication activity, $S(a)$ denotes the component service the communication activities communicates with.

Given an execution $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$, we use a vector $Q'_\pi = \langle q'_1(\pi), \ldots, q'_n(\pi) \rangle$ to represent its aggregated QoS attributes, where $q'_k(\pi)$ is the kth aggregated QoS attributes. $q'_k(\pi)$ is calculated as follows:

$$q'_k(e) = \underset{a \in R}{\mathrm{F}_k} \Big(q_k\big(S(a)\big) \Big), \tag{1}$$

with $R = \{a_i \mid i \in \{1, \ldots, n\} \wedge a_i$ is an action belonging to synchronous communication activity$\}$ and F_k is the QoS aggregation function for attribute k, defined below:

Response time	Availability	Throughput
$\sum_{i=1}^{n} q(s_i)$	$\prod_{i=1}^{n} q(s_i)$	$\min_{i=1}^{n} q(s_i)$

Other QoS attributes share the similar aggregation functions, e.g, the cost attribute has the same aggregation function as the response time attribute. Component services have multi-dimensional attributes, and we need a methodology to facilitate their comparison in term of their QoS. In this work, we use a simple additive weighting (SAW) technique [28] to obtain a score for multi-dimensional attributes. This simple additive weighting technique uses two stages for producing the score. The *normalization* stage normalizes the QoS attribute values so that they are independent of their units and range to allow comparison. The *weighting* stage allows users

Algorithm 3: Fitness

```
input  : Population popul, chromosome csome
output: Fitness value of csome
1  $Exec \leftarrow \emptyset$ ;  $state \leftarrow$ failure state;
   // for b-gene
2  for $int\ i = 1$ to $ind[0]$ do
3      $Exec \leftarrow Exec ^\frown state$;  $state \leftarrow compensate(state)$;
4  $Exec \leftarrow Exec ^\frown state$;
   // for f-genes
5  $states \leftarrow EnableStates(state)$;
6  while $true$ do
7      foreach $s \in states$ do
8          if $s.id = \emptyset$ then
9              $currId \leftarrow currId + 1$;
10             if $currId \geq chromo\_size$ then
11                 $eArr \leftarrow createNewGenes(csome, stateArr)$;
12                 foreach $csome \in popul$ do
13                     $csome \leftarrow csome ^\frown eArr$;
14             $s.id \leftarrow currId$;
15     $state \leftarrow \arg\max_{s \in states}(csome[s.id])$;
16     $Exec \leftarrow Exec ^\frown state$;  $states \leftarrow EnableStates(state)$;
17 $feasible \leftarrow verify(\mathcal{M}, Exec)$; $ncState \leftarrow getNcState(Exec)$;
18 $ncState.execs.add(getNcExec(Exec))$;
19 if $feasible$ then
20     return $0.5 + 0.5 * (G(getCExec(Exec)) + G(ncState))$;
21 else
22     return $0.5 * (G(getCExec(Exec)) + G(ncState))$;
```

to specify their preferences on different QoS attributes. Normalization of aggregated QoS of an execution π is done by comparing with the maximum and minimum aggregated QoS. The maximum (resp. minimum) aggregated QoS can be obtained by aggregating maximum (resp. minimum) QoS attribute values. Formally: $Q_{min}(k) = F_{i=1}^{m} K_{min}$ and $Q_{max}(k) = F_{i=1}^{m} K_{max}$ with $K_{min} = \min_{s \in cs} q_k(s)$ and $K_{max} = \max_{s \in cs} q_k(s)$, where $Q_{min}(k)$ and $Q_{max}(k)$ are the minimum and maximum aggregated values for kth QoS attribute of execution π, m is the number of states in the longest execution of a composite service, and CS is the set of all component services that are used by the composite service. m can be easily obtained with static analysis on the composite service.

Suppose each service has r QoS attributes; the *QoS optimality of the execution* π, $Q(\pi)$, is calculated as follows using SAW:

$$Q(\pi) = \begin{cases} \sum_{k=1}^{r} \frac{Q_{max}(k) - q'_k(\pi)}{Q_{max}(k) - Q_{min}(k)} \cdot w_k & \text{if } Q_{max}(k) \neq Q_{min}(k) \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

where $w_k \in \mathbb{R}^+$ is the weight of q_k and $\sum_{k=1}^{r} w_k = 1$.

Given an uncontrollable state s, the *QoS optimality of the state* s, $Q(s)$, is the average value of the QoS optimality of execution that starts from the state s, i.e.:

$$Q(s) = \frac{1}{|E|} \sum_{e \in E} Q(e) \tag{3}$$

where E is the set of execution that starts from state s and ends in a terminal state. Given E_r a controllable prefix of recovery plan r, and S_r an uncontrollable state of r, the QoS optimality of r, $Q(r)$, is:

$$Q(r) = Q(E_r) + Q(S_r) \tag{4}$$

where $Q(E_r)$ and $Q(S_r)$ are calculated using Equation (2) and Equation (3) respectively.

4.5.2 Global Optimality

The global optimality of a recovery plan concerns both the QoS optimality and whether the recover plan satisfies the functional requirements. The global optimality $G(r)$ of a recovery plan r is:

$$G(r) = \begin{cases} 0.5 + 0.5 \cdot Q(r) & \text{if } r \models \mathcal{M} \\ 0.5 \cdot Q(r) & \text{otherwise.} \end{cases} \tag{5}$$

The global optimality for a recovery plan such that $r \models \mathcal{M}$ (resp. $r \not\models \mathcal{M}$) has its value ranging from 0.5 to 1 (resp. 0 to 0.5). Therefore, it can be guaranteed that a recovery plan satisfying the functional requirements has a higher global optimality value than any recovery plan violating the functional requirements.

DEFINITION 5. *Given a composite service CS, and the set of all feasible recovery plans R_f, the optimal recovery plan r_m is the feasible recovery plan with the maximal optimal value, i.e., $r_m = \arg\max_{r \in R_f}(G(r))$.*

In the following, we present a heuristic method *rGA*, used to find the optimal recovery plan.

4.5.3 Fitness Function

Given a chromosome, we need a metric to decide its worthiness as a candidate solution. The fitness function, denoted as $Fitness$, is used to provide the evaluation, and returns a value called fitness value that represents the worthiness of the candidate solution. The fitness value is typically used by the selector to decide which pair of chromosome instances will be chosen for mating. Highly fit chromosomes relative to the whole population will have higher chance of being selected for mating, whereas less fit chromosomes have a correspondingly low probability of being selected. Some chromosomes generated by the crossover and mutation operations might be infeasible, i.e., they do not satisfy the functional properties of the composite service. We do not simply discard infeasible chromosomes as they might provide candidates that are essential for the optimal solution. Therefore, the strategy is to allow infeasible chromosomes to stay in the population, but with a lower fitness value compared to any feasible chromosome. The fitness value of the chromosome C_r is the global optimality of the recovery plan r that it represents, i.e., $Fitness(C_r) = G(r)$.

The fitness function is computed using Algorithm 3. Lines 7-14 are the procedure discussed in Section 4.3 for the purpose of associating states in LTS with f-genes on the chromosome. At line 8, $s.id$ is the index of f-gene on the chromosome that state s has been associated with. At line 11, if the current size of the chromosome is insufficient for encoding the recovery plan, an extension array *eArr* is created, populated with unused and unique priority values. The new states that are encountered will be added to the global state array $stateArr$ at the same time. Subsequently, all chromosomes in the population are extended with *eArr* (line 13). At line 15, the enabled state with maximal priority value is chosen as the next state. At line 17, *verify* checks whether the execution could satisfy $\mathcal{M}$ using approach discussed in Section 3.3; then, *getNcState* gets the uncontrollable state of the execution. At line 18, *getNcExec* gets the part of execution *Exec* that starts from the uncontrollable state, and is added to *ncState.execs*, which is a set of uncontrollable suffixes that are associated with the *ncState*. At lines 20 and 22, *getCExec* gets the controllable prefix of the execution. The calculation of fitness value in lines 19 to 22 is according to Equation (5). For the calculation of $Q(s)$ for uncontrollable state s using Equation (3),

Algorithm 4: Initial Population

```
input  : $n$ (population size), $l$ (chromosome size)
output: An initial population $P$
1  $P \leftarrow \langle c_1, c_2, \dots, c_n\rangle$;    $stateArr \leftarrow \langle \emptyset_1, \emptyset_2, \dots, \emptyset_l\rangle$ ;
2  foreach $c_i \in P$ do
3      $c_i \leftarrow \langle 0_1, 0_2, \dots, 0_l\rangle$;    $len \leftarrow |stateArr|$;
4      $c_i[0] \leftarrow rand(1, len)$ ;
5      $c_i[1 \dots len] = shuffle(\{1, \dots, len\})$;
6      $S \leftarrow EnableStates(R(c))$ ;
7      if $rand(0,1) \leq P_{EIPP}$ then
8          $S^r \leftarrow rankWithFitness(R[c], S \setminus stateArr)$;
9          foreach $s \in S^r$ do
10             $stateArr.Add(s)$;    $len \leftarrow |stateArr|$;
11             $c_i[len] \leftarrow len$;
12     else
13         $stateArr.AddAll(S \setminus stateArr)$;
14         $c_i[len \dots |stateArr|] \leftarrow shuffle(\{len, \dots, |stateArr|\})$;
```

Algorithm 5: GA Algorithm

```
input  : Abstract LTS $LTS$
output: Recovery plan $R_{max}$ with the best global optimality
        value over all generations
1  $popul \leftarrow init\_popul(pop\_size, chromo\_size)$;
2  $gen \leftarrow 1$ ;  $R_{max} \leftarrow \emptyset$ ;
3  repeat
4      $newPopul \leftarrow max\_ind(popul)$;
5      if $fit(max\_ind(popul)) > fit(R_{max})$ then
6          $R_{max} \leftarrow max\_ind(popul)$;
7      foreach $\langle P_1, P_2\rangle \in sample(popul, pop\_size/2)$ do
8          $\langle C'_1, C'_2\rangle \leftarrow crossover(P_1, P_2)$;
9          $\langle C_1, C_2\rangle \leftarrow \langle mutation(C'_1), mutation(C'_2)\rangle$;
10         $newPopul \leftarrow newPopul \cup \{C_1, C_2\}$;
11     $popul \leftarrow newPopul$;
12     $gen \leftarrow gen + 1$;
13 until $gen > max\_gen$;
14 return $R_{max}$
```

since we may not have exact set E due to the partial exploration of the state space, the set E is approximated using the set of execution starts from s that we have explored so far, which is the set $s.execs$ (calculated in line 18).

4.6 Enhanced Initial Population Policy

We propose an Enhanced Initial Population Policy (EIPP) to overcome shortcomings resulting from randomness of genetic algorithm, such as slow convergence and great variance among the running results. The idea behind is that, by adding the chromosomes likely to contribute to high fitness values to the initial population, we have a higher chance to converge faster to an optimal value.

We introduce the EIPP using the recovery plan r_{TBS}. Suppose we now have value 3 in the b-gene, and the recovery plan would go from state s_{Err} to state s_2. At state s_2, there are two enabled states – states s_3 and s_4. Assume states s_3 and s_4 do not have their priority values assigned yet. EIPP decides the priority values based on the global optimality values of the partial recovery plans. In particular, we compare the global optimality values of partial recovery plan r_p from error state s_{Err} to states s_3 and s_4 respectively. The global optimality values of partial recovery plan r_p, $G(r_p)$, is calculated using Equation (5) with $Q(r_p)$ calculated using Equation (2).

Note that assigning priority values according to $G(r_p)$ does not always provide the optimal recovery plan. Suppose the partial recovery plan from state s_{Err} to s_3 and s_4 is r_p^3 and r_p^4 respectively. Assume $G(r_p^4) > G(r_p^3)$; in such a case, it would always require invoking the `ba2` from state s_4 or from state s_7. However, `ba2` has a low availability value, since it was previously unresponsive. Therefore, we would end up in getting a recovery plan of low global optimality value; we denote this as the *locality problem*.

To address this problem, the priority values are assigned based on the global optimality value of the partial recovery plan with a probability, denoted as EIPP probability $P_{EIPP} \in \mathbb{R} \cap (0.5, 1]$. The value EIPP of the probability P_{EIPP} is strictly larger than 0.5 to make the EIPP in favor of assigning the priority values for f-genes based on the global optimality values of partial recovery plans. Suppose that $P_{EIPP} = 0.7$ and $G(r_p^4) > G(r_p^3)$, given a population of 20 chromosomes, on average 6 chromosomes would choose to evolve to state s_4 from state s_2, which would lead to a recovery plan with better global optimality value. During the evolution, the poor recovery plans that choose state s_4 from state s_2 would be eliminated and the good recovery plans that choose state s_3 from state s_2 would be kept. Therefore, the EIPP probability could effectively mitigate the locality problem.

The algorithm for initializing the population with EIPP is provided in Algorithm 4. Line 1 initializes the population with n chromosomes, which have length l with values of genes set to 0 (line 3). The state array *stateArr* is also initialized to length l, where $\emptyset_i$ denotes an uninitialized value for the ith position. The function $rand(1, len)$ returns a random number $n \in \mathbb{Z} \cap [1, len]$ and assigns it to the b-gene (located at index 0) of chromosome c_i (line 3). Subsequently, we shuffle the numbers of the set $\{1, \dots, len\}$ randomly, and assign them to the f-genes from index 1 to *len* (line 5). Subsequently, we get the enable states of partial recovery plan that represented by c, denoted as $R(c)$, and assign them to variable S (line 6). If the random number $r \in \mathbb{R} \cap [0, 1]$ is not larger than P_{EIPP} (line 7), then the *rankWithFitness* function sorts the enabled states S not in the state array *stateArr* by their fitness values in ascending order, and assigns them to variable S^r. S^r is an ordered sequence of states ranked by the fitness values of their partial recovery plans (line 8). We add each state s to the state array, and assign the corresponding f-genes with value $|stateArr|$ just after adding s – this will effectively allow us to assign the priority values in the same order as their fitness values (lines 9–11). Otherwise, all the enable states that are not in state array *stateArr* are added to *stateArr*. Subsequently, we shuffle the numbers of the set $\{len, \dots, |stateArr|\}$ randomly, and assign them to the f-genes from index *len* to $|stateArr|$ (lines 13–14).

4.7 rGA Algorithm

The *rGA* algorithm is given in Algorithm 5. At line 1, the initial population is initialized using the EIPP given in Algorithm 4 with the default population size *pop_size* and the default chromosome size *chromo_size*. At line 4, the next population *newPopul* is extended with the chromosome with maximal fitness value in the current population *max_ind*(*popul*), due to the elitist selection adopted. At lines 5–6, the recovery plan with maximal fitness value so far is assigned to R_{max}. At line 7, *pop_size*/2 pairs of chromosomes (P_1, P_2) are sampled using the selection operator discussed in Section 4.4. At lines 8 and 9, crossover and mutation operations are applied to (P_1, P_2), and added to the population of the new generation *newPopul*. This process repeats until it has gone through the maximum number of generations specified by *max_gen*.

Soundness. For *rGA* to work correctly, we need to ensure that every chromosome uniquely represents a recovery plan (unique en-

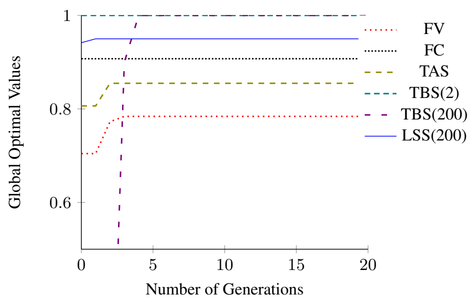


Figure 8: Convergence rate

coding property), and there does not exist any recovery plans that *rGA* avoid exploring (non-blocking property). We show *rGA* satisfies these two properties in the following.

LEMMA 1 (UNIQUE ENCODING). *Given any state in LTS as a starting state, the proposed chromosome uniquely encodes a recovery plan.*

PROOF. For the backward execution, given that the graph is acyclic and the compensation is deterministic (since there is a deterministic execution from the initial state to the error state s_{Err} where the failure occurs), the value of the b-gene uniquely determines the migration state. For the forward execution, once a state is added to the state array, it remains in its position in the state array. Therefore, starting from the migration state, we could choose an execution deterministically based on the priority values in f-genes. Combining both, we show that the chromosome can uniquely determine an abstract recovery execution. □

LEMMA 2 (ACYCLICITY). *Given any state in LTS as a starting state,* rGA *does not avoid the exploration of any recovery plan.*

PROOF. This holds due to the fact that there are no recursive activities in BPEL, and due to the assumption on the loop activities for which the upper bound on the number of iterations is known. □

5 Evaluation

We conducted experiments to evaluate our *rGA* approach. Specifically, we attempted to answer the following research questions.

RQ1. *How does the* performance *of* rGA *compare with the state-of-the-art?* We analyze how long *rGA* takes to calculate a recovery plan, and compare the performance with the state-of-the-art.

RQ2. *How is the* quality *of recovery plan that is selected by* rGA*?* We measure the quality using the formula

$$quality = \frac{G(r)}{G(r_{exact})} \tag{6}$$

where $G(r)$ and $G(r_{exact})$ are the global optimality values of recovery plan returned by the *rGA* method and the exact method (i.e., the method exhaustively enumerating every possible recovery plans), respectively.

RQ3. *How* scalable *is* rGA*?* To evaluate the scalability of *rGA*, we use the parameterized Large Scale Service (LSS), and Travel Booking Service (TBS) that contain the combinatorial explosion of recovery plans given large values for parameters. In such cases, it is impractical to solve by enumerating all recovery plans.

Experimental Setup. The experiments were conducted on an Intel Core I5 2410M CPU with 4 GiB RAM, running on Windows 7. The mutation and crossover rates for *rGA* are set to 0.01% and 0.9% respectively. In addition, the population size is set to 20, the number of generations is set to 20, and P_{EIPP} is set to 0.7. The algorithm *rGA* could terminate earlier if it discovers that the fitness value does

rGA				SAT	
case study	time (s)	quality	gen.	length	time (s)
FV	0.7	1	10	42	3.12
FC	0.12	1	6	20	1.38
TAS	0.22	1	6	13	0.27
TBS(2)	0.47	1	6	N/A	N/A
TBS(30)	0.54	1	8	N/A	N/A
TBS(60)	0.87	1	8	N/A	N/A
TBS(120)	1.24	1	10	N/A	N/A
TBS(200)	1.97	1	10	N/A	N/A
LSS(30)	0.85	0.97	7	N/A	N/A
LSS(60)	0.96	0.97	7	N/A	N/A
LSS(80)	1.42	0.96	8	N/A	N/A
LSS(120)	1.92	0.95	8	N/A	N/A
LSS(200)	2.57	0.94	8	N/A	N/A

Figure 9: Experiment with *rGA*

not improve for over 6 generations. To evaluate *rGA*, we explicitly construct an execution that violates the functional properties and leads to the error state s_{Err}, and explore the recovery plan from the error state s_{Err} using *rGA*. Since *rGA* could perform differently for each experiment, we took the average of 50 experiments for each case. In addition, after the fitness value is calculated for a chromosome, we cache the fitness value of the chromosome, so that the fitness value for the same chromosome does not need to be recalculated. We now introduce the case studies used for the experiments. To answer the previous research questions, we evaluate *rGA* using five case studies described in the following.

Flickr [4] is a Web application allowing users to upload and share their photos on the Web. Two known vulnerabilities in the Flickr Web application [9], namely *Flickr Visibility (FV)* and *Flickr Comment (FC)*, are used to evaluate the effectiveness of our approach. Both FV and FC have been translated to a BPEL model (see [22]).

Flickr Visibility (FV). The options for the photos' visibility are *public*, *family* and *private*; users can set the visibility through the `setPerms`() function. There is a reported issue [1] on this function to fail on changing the visibility to *family*, after uploading the photos with an initial *private* visibility. The BPEL model contains 28 activities and 8 with explicit compensation; its LTS consists of 36 states and 86 transitions. The functional property to be monitored for FV is "Flickr guarantees the photos to have the visibility set by the user".

Flickr Comment (FC). Flickr allows authorized users to add comments to private and family photos; for public photos, all users are allowed to comment. There is a reported issue [2] on the failing sequence of a single call to `addComment`() immediately after an `upload`() operation. The BPEL model contains 16 activities and 6 activities with explicit compensation; its LTS consists of 21 states and 51 transitions. The functional property to be monitored for FC is "if a user adds comments to a public photo, the comments should be added successfully into the photo's comments".

Trip Advisor System (TAS). This case study is introduced in [23]. The objective of TAS is to schedule the trip for user. It consists of 25 states and 34 transitions. The functional property to be monitored for TAS is "the user cannot book both a limousine and an expensive flight".

Travel Booking Service (TBS). This is the example used throughout the paper. In the case study it involves two `<onMessage>` for two airline services. We parameterize the case study using n of `<onMessage>` that for n distinct airline services, each of them involving airline and insurance booking. We denote TBS with k `<onMessage>` activities by *TBS*(k). For example, the example used in this paper has $k = 2$. TBS consists of $7 + 3k$ states and

$8 + 5k$ transitions. The functional property to be monitored for TBS is "the service always replies to the user". TBS contains a non-responsive airline booking service `ba2`, invoking `ba2` would lead to the error state s_{Err}, which violates the functional property.

Large Scale Service (LSS). To better evaluate the scalability of our approach, we built a BPEL example with a sequence of k `<pick>` activities. Each `<pick>` activity consists of two `<onMessage>` activities, where one has a good QoS, while the other has a bad QoS. The optimal recovery plan in such a scenario will always consist of the activities with good QoS. We denote LSS with k `<pick>` activities by *LSS*(k). *LSS*(k) contains at least $2k$ states and 2^k unique candidate recovery plans. The functional property to be monitored for *LSS* is "the service always replies to the user".

Results. We report the results of evaluating the case studies in Figure 8 and Figure 9. Figure 8 shows the global optimal values of the case studies by varying the number of generations, and we could observe the convergence rate for different case studies. It demonstrates the fast convergence rate for all case studies.

Figure 9 shows the details of evaluation for the cases studies. The "time (s)" column reports the time in seconds for *rGA* to produce the recovery plan. The "quality" column reports the quality of the recovery plan found by *rGA* calculated using Equation (6). The "gen." column reports the number of generations that are used to search for recovery plans. We compare the results with [23], which we call *SAT*, since their approach uses a SAT solver to find the best recovery plan. The objective of SAT is to find a set of recovery plans that are functionality correct, and the user is responsible for selecting recovery path manually. In addition, Their method is required to specify the maximum length k for the recovery plan, i.e., only the recovery plans less than k and fulfilling the functional requirements are returned. The "length" column contains the value for k. In contrast, in our approach, all recovery plans are explored, and functionality correct recovery plan is chosen automatically in term of their QoS before returning to the user. We compare our results with theirs using their results on three case studies – FV, FC, and TAS they have reported in [23]. For other case studies, their results are unavailable (denoted by N/A). Our method searches the entire state space for optimal recovery plan, without restricting the length of the recovery plan. Therefore, to facilitate fair comparison, we only compare with SAT, using the largest k values for the case studies that have been reported in [23].

Our results have shown to outperform theirs for all case studies. In addition, most recovery plans that returned by *rGA* are optimal, i.e., quality=1, except in the LSS case study, which has suboptimal quality, i.e., with quality closed to 1. In addition, we observe that although LSS(80) and LSS(200) used the same number of generation, but LSS(200) spent more time than LSS(80). This is because LSS(200) contains more states than LSS(80), which results in longer chromosome and slower processing time. The same observation can be applied, e.g., to TBS(120) and TBS(200).

Answer to Research Questions. For research questions RQ1–RQ3, the results in Figure 8 and Figure 9 show that *rGA* is efficient to offer a recovery plan of good quality, and it is scalable to large composite service.

6 Related Work

This work is related using genetic algorithm to fix software faults. Weimer *et al.* [27] and Arcuri *et al.* [6] investigate genetic programming as a way to automatically fix software faults. Their approach assumes the existence of test cases to test for the functional correctness of chromosomes. In contrast, our method generates a recovery plan, and the functional correctness is checked using the monitoring automata; this is a more lightweight procedure, and it is shown to be suitable for executing it online (see Section 5).

This work is related to research on fault-tolerant for service composition. Dodson [12] transforms the original BPEL process into a fault-tolerant one at compiling time, by considering common fault tolerance patterns. This approach introduces redundant behavior to BPEL programs, which may slow down the performance. In contrast, our service monitoring executes at BPEL runtime, which avoids such redundancy. Carzaniga *et al.* [8] propose the use of workaround by considering the equivalent sequences of faulty action in order to provide a temporary solution to mask the effects of the faults on applications. The approach generates all possible recovery plans, without prioritizing them. In contrast, our method filters out infeasible recovery plans; as for the feasible ones, they are ranked by QoS of involved component services.

This work is related to automated recovery for service composition. Baresi *et al.* [7] propose an idea of self-supervising BPEL processes by supporting both service monitoring and recovery for BPEL processes. They propose a backward strategy, which is to restore the system to a previously correct state. However, the strategy does not consider the potential satisfaction of functional properties, and neither is it QoS-aware. Simmonds *et al.* [23] propose an approach to divide a recovery plan to compensate the failures, and guide the application towards a desired behavior. This work is the closest to ours, and our approach has several advantages over theirs. Firstly, the method in [23] requires the exhaustive LTS exploration for the BPEL process by using a SAT solver for calculating the recovery plan. Our approach only requires a partial exploration of the LTS. Also, their method does not take into account the QoS of component services explicitly. Our approach accounts for various QoS aspects of a component service explicitly, and allows users to weight them according to their preferences.

This work is related to self-adaptation of service composition. In [21], Mukhija and Glinz propose an approach to adapt an application by recomposing its components dynamically, which is implemented by providing alternative component compositions for different states of the execution environment. Ghezzi *et al.* [18] describe an ADAM model-driven framework for adaptation by choosing a path that could maximize system's non-functional properties. These works are orthogonal to our work, as they are related to providing runtime adaptation for normal execution based on the runtime and contextual information, while our work is related to failure recovery.

In [26], we propose an automatic approach to synthesize local time requirement based on the given global time requirement of Web service composition. In [10], we propose an approach to verify the functional and non-functional requirements of Web service composition. Different from our previous works, this work is focused on automatic synthesis of recovery plan.

7 Conclusion and Future Work

In this paper, we address the problem of service recovery by proposing a new method (*rGA*) based on genetic algorithms. The method improves the efficiency of existing methods in flow-based recovery strategy by allowing partial exploration of the LTS for near-optimal recovery plan. In addition, the recovery plan selection is QoS-aware; therefore, it allows effective recovery from the failure state. As future work, we plan to investigate how to combine *rGA* with other techniques, such as differential evolution [25].

Acknowledgement

This work has been supported in part by research grant IDD11100102 (SUTD).

8 References

[1] http://www.flickr.com/help/forum/46985/.

[2] http://www.flickr.com/help/forum/15259/.

[3] Apache ODE. http://ode.apache.org/.

[4] Flickr. http://www.flickr.com/.

[5] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, IBM, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version, version 2.0. Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, April 2007.

[6] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168. IEEE, 2008.

[7] L. Baresi and S. Guinea. Self-supervising BPEL processes. *IEEE Transactions on Software Engineering*, 37(2):247–263, 2011.

[8] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *SIGSOFT FSE*, pages 237–246. ACM, 2010.

[9] A. Carzaniga, A. Gorla, and M. Pezzè. Healing web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, 2008.

[10] M. Chen, T. H. Tan, J. Sun, Y. Liu, J. Pang, and X. Li. Verification of functional and non-functional requirements of web service composition. In *ICFEM*, pages 313–328, 2013.

[11] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0. http://www.w3.org/TR/wsdl20/.

[12] G. Dobson. Using WS-BPEL to implement software fault tolerance for Web services. In *EUROMICRO-SEAA*, pages 126–133. IEEE, 2006.

[13] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, 2005.

[14] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.

[15] S. Forrest et al. Genetic algorithms – Principles of natural selection applied to computation. *Science*, 261(5123):872–878, 1993.

[16] H. Foster. *A rigorous approach to engineering Web service compositions*. PhD thesis, Citeseer, 2006.

[17] M. Gen, R. Cheng, and D. Wang. Genetic algorithms for solving shortest path problems. In *Evolutionary Computation*, pages 401–406. IEEE, 1997.

[18] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *ICSE*, pages 33–42, 2013.

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. Simple object access protocol (SOAP) version 1.2. http://www.w3.org/TR/soap12/.

[20] G. P. Inc. 36 human-competitive results produced by genetic programming. http://www.genetic-programming.com/, 2012.

[21] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *ARCS*, pages 124–138, 2005.

[22] J. Simmonds. *Dynamic Analysis of Web Services*. PhD thesis, University of Toronto, 2011.

[23] J. Simmonds, S. Ben-David, and M. Chechik. Guided recovery for Web service applications. In *SIGSOFT FSE*, pages 247–256, 2010.

[24] J. Simmonds and M. Chechik. Rumor: monitoring and recovery for BPEL applications. In *ASE*, pages 345–346, 2010.

[25] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[26] T. H. Tan, É. André, J. Sun, Y. Liu, J. S. Dong, and M. Chen. Dynamic synthesis of local time requirement for service composition. In *ICSE*, pages 542–551, 2013.

[27] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374. IEEE, 2009.

[28] K. Yoon and C. Hwang. *Multiple attribute decision making: An introduction*. Sage Publications, Incorporated, 1995.