Guided Recovery for Web Service Applications

Jocelyn Simmonds, Shoham Ben-David, Marsha Chechik
Department of Computer Science
University of Toronto
Toronto, ON M5S 3G4, Canada
{jsimmond, shoham, chechik}@cs.toronto.edu

ABSTRACT

Web service applications are dynamic, highly distributed, and loosely coupled orchestrations of services which are notoriously difficult to debug. In this paper, we describe a user-guided recovery framework for web services. When behavioural correctness properties (safety and bounded liveness) of an application are violated at runtime, we automatically propose and rank recovery plans which users can then select for execution. For safety violations, such plans essentially involve "going back" – compensating the occurred actions until an alternative behavior of the application is possible. For bounded liveness violations, such plans include both "going back" and "re-planning" – guiding the application towards a desired behavior. We report on the implementation and our experience with the recovery system.

Keywords

Web services, LTS, behavioural properties, runtime monitoring, planning, SAT solving.

1. INTRODUCTION

Recent years have seen the increased reliance on being able to conduct business over the Internet. The Service-Oriented Architecture (SOA) framework is a popular guideline for building web-based applications. A SOA-based application is an orchestration of services offered by (possibly third-party) components written in a traditional compiled language such as Java, or in an XML-centric language such as BPEL[1].

Web services are distributed systems, where partners are dynamically discovered and are going on- and off-line as the application runs. Their failures can be caused by bugs in the service orchestration, e.g., due to faulty logic and bad data manipulation, or by problems with hardware, network or system software, or by incorrect invocations of services. With runtime failures of web services inevitable, infrastructures for running them typically include the ability to define faults and compensatory actions for dealing with exceptional situations. Specifically, the *compensation* mechanism is the application-specific way of reversing completed activities. For example, the compensation for booking a car would be to cancel the booking.

Existing infrastructures for web services, e.g., the BPEL engine, include mechanisms for fault definition, for specification of compensation actions, and for dealing with termination. When an error is detected at runtime, they typically try to compensate all completed activities for which compensations are defined, with the default compensation being the reversal of the most recently completed action. This approach presents several major problems: (1) The application is often allowed to continue running until the fault is discovered, thus executing and then compensating for a lot of unnecessary and potentially expensive activities. (2) It is hard to determine, a priori, the state of the application after executing compensation mechanisms. (3) There might be multiple compensations available, based on global information (i.e., avoid canceling the flight since it has a dollar cost associated with it, and try to cancel the hotel instead), but the automatic application of compensations does not allow the user of such a system to choose between them.

This paper describes a *user-guided* recovery framework for web services, instantiating it on BPEL programs. We concentrate on behavioural correctness, and specifically, on the correct interaction between service partners. The overview of the approach is given in Fig. 1a. Our approach consists of three phases: Preprocessing, Monitoring and Recovery. It admits the following user guidance: (I) Application *developers* define a set of behavioral correctness properties that need to be maintained at runtime, as well as compensation costs and idempotent service calls (see Sec. 3.2) (II) (Optional) Application *users* provide criteria for choosing between possible recovery plans, i.e., based on the plan length, compensation cost, etc. (III) Application users manually choose the desired recovery plan among those automatically computed and ranked by our system.

We consider *behavioral correctness properties* to be scenarios that the system should exhibit and scenarios that the system should not exhibit. For example, consider a simple web-based Trip Advisor System (TAS). In a typical scenario, a customer either chooses to arrive at her destination via a rental car (and thus books it), or via an air/ground transportation combination, combining the flight with either a rental car from the airport or a limo. The requirement of the

[1] `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`

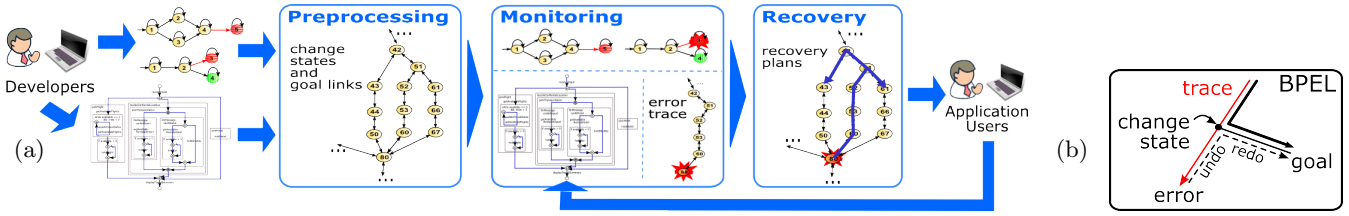


Figure 1: (a) Overview of our approach; (b) a schematic view on plan generation.

system is to make sure the customer has the transportation needed to get to her destination (this is a desired behaviour) while keeping the costs down, i.e., she is not allowed by her company to reserve an expensive flight and a limo (this is a forbidden behavior). Such desired and forbidden behaviours can come from use-cases, global invariants, simulation, or a variety of other sources. They can be expressed in (temporal) logic or in visual notations, such as UML's Sequence Diagrams [24]. The problem of collecting and expressing properties has been addressed by many researchers [1, 5, 15], and we consider it to be orthogonal to this paper. We also refer to the specification of negative behaviour as *safety* properties, and positive behaviour – as *(bounded) liveness properties*. In the interest of space, we further assume that properties are represented by finite-state monitors.

A description of the desired orchestration and the set of monitors describing safety and liveness properties are then passed to the runtime monitoring framework which runs the monitors in parallel with the application, stopping when one of the monitors enters its error state. We build on an earlier work of defining and implementing an unintrusive online framework for runtime monitoring of conversations between partners [24]. Using high-level properties allows us to detect the violation, and our event interception mechanism allows us to stop the application *right before* the violation occurs, and enable recovery.

Our main contribution is the development of recovery plans from runtime errors. Given an application path which led to a failure and a monitor which detected it, our goal is to compute a set of suggestions, i.e., *plans*, for recovering from these failures. For violations of safety properties, such plans use compensation actions to allow the application to "go back" to an earlier state at which an alternative path that potentially avoids the fault is available. We call such states "change states"; these include user choices and certain partner calls. For example, if the TAS system produces an itinerary that is too expensive, a potential recovery plan might be to undo the limo reservation (so that the car can now be booked) or to undo the flight reservation and see if a cheaper one can be found.

Yet just merely going back is insufficient to ensure that the system can produce a desired behaviour. Thus, in order to satisfy (bounded) liveness properties, we aim to compute plans that redirect the application towards executing new activities, those that lead to goal satisfaction. For example, if the flight reservation partner fails (and thus the air/ground combination is not available), the recovery plans would be to provide transportation to the user's destination (her "goal" state) either by calling the flight reservation again or by undoing the reserved ground transportation from the airport, if any, and try to reserve the rental car from home instead. The overall recovery planning problem is then stated as follows:

> From the current (error) state in the system, find a plan to achieve the goal that goes through a change state.

This process is shown schematically in Fig. 1b. When there are multiple recovery plans available, we automatically rank them based on user preferences (e.g., the shortest, the cheapest, the one that involves the minimal compensation, etc.) and enable the application user to choose among them.

This paper makes the following contributions: (1) We propose a user-guided framework for recovery from run-time violations of behavioural properties representing desired and prohibited conversations between partners in web service applications. (2) We show the difference between recovery from a violation of safety and bounded liveness properties and propose automated strategies for such recoveries. (3) We pose recovery problem as a plan generation problem and in turn reduce it to a SAT instance which allows us to control the size of the resulting plans and compute multiple plans.

The rest of this paper is organized as follows. We describe inputs to our system, BPEL models and monitors representing properties, in Sec. 2. We define the representation of BPEL models as Labeled Transition Systems (LTS) and show how to use these representations for *static* identification of change states and goal transitions in Sec. 3. We briefly discuss runtime monitoring in Sec. 4 and describe our main contribution, recovery for violations of safety and bounded liveness properties in Secs. 5 and 6 respectively. We report on our implementation (Sec. 7) and use it to compute recovery plans for two web service examples (Sec. 8). After comparing our work with related approaches in Sec. 9, we conclude in Sec. 10 with a summary of the paper and suggestions for future work.

2. INPUT

Inputs to our system are a BPEL program enriched with compensation actions and a set of behavioral correctness properties described by monitors. We describe these below.

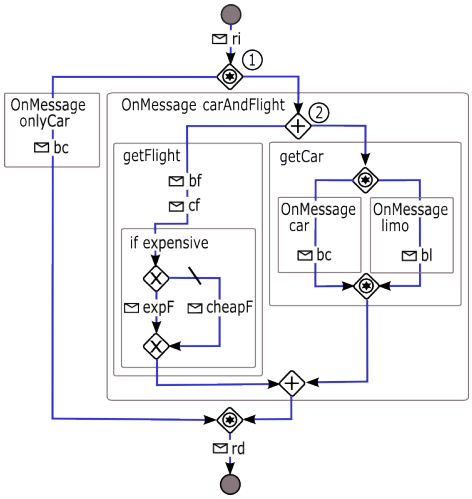
2.1 BPEL Programs

BPEL is a standard for implementing orchestrations of web services (provided by partners) by specifying an executable workflow using predefined activities. The basic BPEL activities for interacting with partner web services are <receive>, <invoke> and <reply>, which are used to receive messages, execute web services and return values, respectively. The control flow of the application is defined using structural activities such as <while>, <if>, <sequence>, <flow> and <pick>.

Fig. 2a shows the BPEL-expressed workflow of the Trip Advisor System (`TAS`), introduced in Sec. 1. We use the NetBeans SOA notation[2]. `TAS` interacts with four external services: 1) book a rental car (`bc`), 2) book a limo (`bl`), 3) book a flight (`bf`), and 4) check price of the flight (`cf`).

[2] `http://www.netbeans.org`

(a)



(b)

```
<invoke ... operation = "bf"  ...
            outputVariable = "flightConf">
  <compensationHandler cost="9">
          <invoke ... operation = "cancelF" />
  </compensationHandler>
</invoke>
```

Figure 2: (a) Workflow of TAS; (b) Compensation for booking a flight (bf).

The result of cf is then passed to local services to determine whether it is expensive (expF) or cheap (cheapF). Service interactions are preceded by a ✉ symbol.

The workflow begins with <receive>'ing input (ri), followed by <pick>'ing (indicated by ❖ labeled ①) either the car rental (OnMessage onlyCar) or the air/ground transportation combination (OnMessage carAndFlight). The latter choice is modeled using a <flow> (⊕, labeled ②) since air (getFlight) and ground transportation (getCar) can be arranged independently, so they are executed in isolation. The air branch sequentially books a flight, checks if it is expensive and updates the state of the system accordingly. The ground branch <pick>'s between booking a rental car and a limo. The end of the workflow is marked by a <reply> activity, reporting that the destination has been reached (rd).

Compensation. BPEL's *compensation* mechanism allows the definition of the application-specific reversal of completed activities. For example, the compensation for booking a flight (bf) is to cancel the booking (cancelF); this is described in BPEL as shown in Fig. 2b.

Compensation handlers (CH) are attached to <scope> and <invoke> activities (a <scope> activity is used to logically group activities) and are executed by fault, termination and compensation via the <compensate> and <compensateScope> activities. The default compensation respects the forward order of execution of the scopes being compensated:

> If a and b are two activities, where a completed execution before b, then compensate$(a;b)$ is compensate(b); compensate(a).

Any attempt to compensate a scope for which the CH either has not been installed, or has been installed and executed, is treated as executing an empty activity τ.

We further extended BPEL to allow users to associate compensations with different costs, e.g., to indicate that canceling a flight might be significantly more expensive than canceling a car. We do this by adding an extra attribute cost to the definition of <compensationHandler>. For example, the flight booking compensation defined in Fig. 2b has been assigned a cost of 9 (out of 10), indicating that this is an expensive compensation and should be avoided if possible.

2.2 Properties and Monitors

The second input to our system is a set of correctness properties. As mentioned in Sec. 1, we view the question of harvesting properties as well as the exact language for specifying them as orthogonal to our work, and we assume that properties are expressed as monitors (see Def. 1 below). We differentiate between monitors representing *safety* properties – negative behaviors that should not appear in the application — and monitors representing *liveness* properties – positive behaviors that the system must posses. In general, liveness properties are violated only by infinite behaviors and thus are not monitorable [16]. However, BPEL applications are run-to-completion programs which are expected to terminate. By the explicit addition of the "terminate" (TER) event, liveness properties become bounded and thus are readily monitorable.

For example, the two requirements of the TAS system are to make sure that the customer has the transportation needed to get to her destination (desired behavior), while keeping the costs down (forbidden behavior). More formally, they become P_1 (liveness): "if requested (ri), TAS will guarantee that the transportation booked reaches the customer's destination (rd), regardless of the type of transportation chosen", and P_2 (safety): "the user cannot book both a limousine (bl) and an expensive flight (expF)". Property P_1 is represented by the monitor $\mathtt{A}_1$ in Fig. 3a: the property is satisfied once the booked transportations reaches the destination (rd), and the automaton is in state 4 (colored green and shaded vertically). If the application terminates before rd, the monitor moves to the (error) state 3 (colored red and shaded horizontally). Σ is the alphabet of the monitor, i.e., every event occurring in the application, defined formally in Sec. 3.1. States 1 and 2 are unshaded and colored yellow to indicate that P_1 is neither satisfied nor violated.

The monitor $\mathtt{A}_2$ for property P_2 (see Fig. 3b) goes to the error state 4 when travel includes booking an expensive flight and a limo, in any order. All other states are yellow, since the negative behavior can be detected but its absence cannot be established. We formalize (colored) monitors below.

Definition 1 (monitor). *A* monitor *is a 5-tuple* $A = (S, \Sigma, \delta, I, F)$*, where* S *is a set of states,* Σ *is an alphabet,* $\delta \subseteq S \times \Sigma \times S$ *is a transition relation, and* $I \subseteq S$ *and* $F \subseteq S$ *are sets of* initial *and* final *states, respectively.*

A state s of a monitor A is called a *sink* state if all outgoing transitions from s are self-loops. For example, states 3 and 4 in Fig. 3a and state 4 in Fig. 3b are sinks. We say that A *accepts* a word $a_0a_1a_2...a_{n-1} \in \Sigma^*$ iff there exists an execution $s_0a_0s_1a_1s_2...a_{n-1}s_n$ of A such that $s_0 \in I$ and $s_n \in F$. In our case, the accepted words correspond to *bad* computations, and the set F of accepting states represents error (red) states. A green state is a desired monitor state: when it is reached, the corresponding property cannot be violated in the current computation.

Definition 2 (Colored Monitor). *Let* $A = (S, \Sigma, \delta, I, F)$ *be a monitor, and let* $K \subseteq S$ *be the set of all* sink *states of* A*. Then,* $\forall s \in F \cdot$ color$(s) =$ *red,* $\forall s \in K \setminus F \cdot$ color$(s) =$ *green (all non-accepting* sink *states), and* $\forall s \in S \setminus (F \cup K) \cdot$ color$(s) =$ *yellow.*

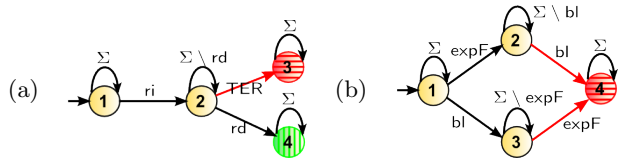


Figure 3: Monitors: (a) A_1, and (b) A_2. Red states are shaded horizontally, green states are shaded vertically, and yellow states are solid.

DEFINITION 3. *A is a* liveness monitor *if it includes at least one green state; otherwise, it is a* safety monitor.

We assume that we are given a set $\mathsf{SM} = \{\mathsf{A}^s_1, ..., \mathsf{A}^s_m\}$ of m safety monitors and $\mathsf{LM} = \{\mathsf{A}^\ell_1, ..., \mathsf{A}^\ell_n\}$ of n liveness monitors. As with any other property-based specification, it is possible that the property list is incomplete (i.e., some behavioural requirements are not captured) or even inconsistent (i.e., satisfying the entire set of requirements is not possible).

3. PREPROCESSING

Inputs to the preprocessing stage are the BPEL program B, the set of safety monitors SM and a set of liveness monitors LM. We begin with converting B into a formal representation, L(B), which is a labeled transition system (LTS). We then enrich it with transitions on compensation actions to get $\mathsf{L_C}$(B)(Sec. 3.1). We formalize *change states* and *goal* transitions and provide an algorithm for computing these statically on $\mathsf{L_C}$(B) (Sec. 3.2).

3.1 BPEL to LTS

In order to reason about BPEL applications, we need to represent them formally, so as to make precise the meaning of "taking a transition", "reading in an event", etc. Several formalisms for representing BPEL models have been suggested [12, 17, 19]. In this work, we build on Foster's [9] approach of using an LTS as the underlying formalism.

DEFINITION 4 (LABELED TRANSITION SYSTEMS). *A* Labeled Transition System *LTS is a quadruple* (S, Σ, δ, I)*, where* S *is a set of states,* Σ *is a set of labels,* $\delta \subseteq S \times \Sigma \times S$ *is a transition relation, and* $I \subseteq S$ *is a set of initial states.*

Effectively, LTSs are state machine models, where transitions are labeled whereas states are not. We often use the notation $s \xrightarrow{a} s'$ to stand for $(s, a, s') \in \delta$.

[9,10] specify mapping of all BPEL 1.1 activities into LTS. For example, Fig. 4 shows the translation of the <invoke> activity bf which returns a confirmation number. The activity is a sequence of two transitions: the actual service invocation (invoke_bf) and its return (receive_bf)[3]. Conditional activities like <while>, <if> and <pick> are represented as states with two outgoing transitions, one for each valuation of the condition. <sequence> and <flow> activities result in the sequential and the parallel composition of the enclosed activities.

The set of labels Σ of the resulting translation L(B) is derived from the possible events in the application B: service invocations and returns, OnMessage events, <scope> entries, and condition valuations. It also includes the new system event TER, modeling termination. The set of states

[3]Foster's translation uses names to include traceability information to the BPEL's scopes. We omit these in this paper for simplicity.

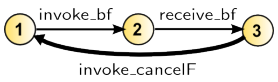


Figure 4: LTS translation of the <invoke> activity bf and its compensation (bold).

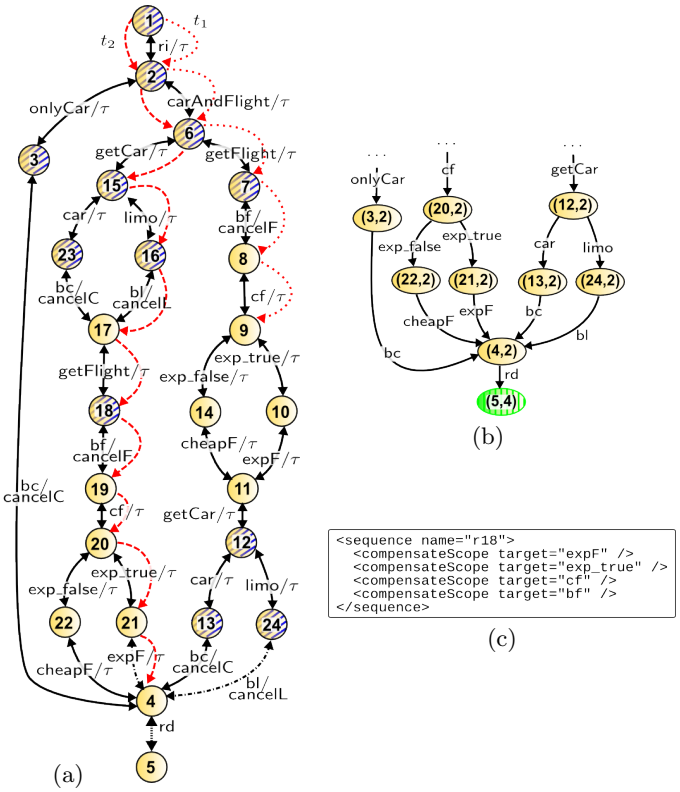


Figure 5: (a) LTS $\mathsf{L_C}$(TAS), showing traces t_1 (dotted) and t_2 (dashed); (b) a fragment of L(TAS) × A_1; (c) a recovery plan in XML.

S in L(B) consists of the states produced by the translation as well as a new state t. This state is reached from any state of S via a TER event: $\forall s \in S \setminus \{\mathsf{t}\}, (s, \mathsf{TER}, \mathsf{t}) \in \delta$.

In order to capture BPEL's compensation mechanism, we introduce additional, backwards transitions. For example, the compensation for bf, specified in Fig. 2b, is captured by adding the transition $3 \xrightarrow{\text{invoke\_cancelF}} 1$ as shown in Fig. 4. Taking this transition effectively leaves the application in a state where bf has not been executed. We denote by τ an 'empty' action, allowing undoing of an action without requiring an explicit compensation action.

Note that we have made a major assumption that compensation returns the application to one of the states that has been previously seen. Thus, given a BPEL program B and its translation to LTS $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I)$, we translate B with compensation into an LTS $\mathsf{L_C}(\mathsf{B}) = (S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I)$, where Σ_c is the set of compensation actions (including τ) and δ_c is the set of compensation transitions.

Fig. 5a shows $\mathsf{L_C}$(TAS). To increase legibility, we do not show the termination state t and transitions to it. Also, we only show one transition for each service invocation, abstracting the return transition and state. In this notation, the LTS in Fig. 4 has two transitions: $1 \xrightarrow{\text{bf}} 3$ and $3 \xrightarrow{\text{cancelF}} 1$. This allows us to visually combine an action and its compensation into one transition, labeled in the form $a/\bar{a}$, where a is the application activity and $\bar{a}$ is its compensation. In

other words, each transition $s \xleftrightarrow{a/\bar{a}} t$ in Fig. 5a represents two transitions: $(s, a, t) \in \delta$ and $(t, \bar{a}, s) \in \delta_c$.

The <pick> activity (labeled ① in Fig. 2a) corresponds to state 2 of Fig. 5a. The choice between onlyCar and carAnd−Flight is represented by two outgoing transitions from this state: (2, onlyCar, 3) and (2, carAndFlight, 6). Since these actions do not affect the state of the application, they are compensated by τ. The <flow> activity (labeled ② in Fig. 2a) results in two branches, depending on the order in which the air and ground transportation are executed. The compensation for these events is also τ.

3.2 Identifying Goal Transitions and Change States

The second part of the preprocessing phase *statically* identifies strategic behaviors of the application $\mathsf{L}(\mathsf{B})$, aimed to help find an efficient recovery plan when a violation is encountered (see Sec. 5 and Sec. 6).

3.2.1 Goal Transitions

In order to find a good recovery plan, we first need to compute a set of *goal transitions*, that is, transitions taken by the application which (immediately) result in satisfaction of some properties. We compute these on a per-property basis. Further, recall that only liveness properties can be satisfied, which is indicated by the monitor reaching a green state; safety properties can only be violated. Thus, for each liveness monitor $\mathsf{A}_\mathsf{i}^\ell \in \mathsf{LM} = (S_i, \Sigma, \delta_i, I_i, F_i)$, we are looking for transitions in $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I)$ corresponding to $\mathsf{A}_\mathsf{i}^\ell$ entering its green state(s). To find those, we compute the cross-product $\mathsf{L}(\mathsf{B}) \times \mathsf{A}_\mathsf{i}^\ell$. $(s, a, s') \in \delta$ is a *goal transition* iff $\exists q, q' \in S_i \cdot (s, q) \xrightarrow{a} (s', q') \wedge \mathit{color}(q) \neq \text{green} \wedge \mathit{color}(q') = \text{green}$. That is, $s \xrightarrow{a} s'$ corresponds to taking a transition on a into a green state of $\mathsf{A}_\mathsf{i}^\ell$. The resulting set of goal transitions is denoted by $\mathsf{G}(\mathsf{B}, \mathsf{A}_\mathsf{i}^\ell)$.

For example, consider a fragment of $\mathsf{L}(\mathtt{TAS}) \times \mathtt{A_1}$ shown in Fig. 5b. The green state of $\mathtt{A_1}$ is state 4, with transition on rd leading to it. The only transition in $\mathsf{L}(\mathtt{TAS}) \times \mathtt{A_1}$ satisfying the above definition is $(4, 2) \xrightarrow{\mathsf{rd}} (5, 4)$, and thus $\mathsf{G}(\mathtt{TAS}, \mathtt{A_1}) = \{(4, \mathsf{rd}, 5)\}$ (depicted by tiny-dashed transitions in Fig. 5a).

When computing recovery plans, we need to direct the application towards taking its goal transitions.

3.2.2 Change States

Given an erroneous run, how far back do we need to compensate before resuming forward computation? If we want to avoid repeating the same error again, we need the application to take an alternative path. States of $\mathsf{L}(\mathsf{B})$ that have actions executing which can potentially produce a branch in control flow of the application are called *change states*.

Flow-changing actions are user choices, states modeling the <flow> activity (since each pass through this state may produce a different interleaving of actions), and those service calls whose outcomes are not completely determined by their input parameters but instead depend on the implicit state "of the world". This characteristics of services is sometimes referred to as *idempotence*, since multiple invocations of the same service yield the same results. Thus, non-idempotent service calls also identify change states. For example, cheapF is a call to determine whether a given flight is cheap and, unless the specification of what cheap means changes, returns the same answer for a given flight. On the other hand, bf books an available flight, and each successive call to this service can produce different results. Non-idempotent service calls are identified by the BPEL developer as XML attributes in the BPEL program.

We denote by $\mathsf{C}(\mathsf{B})$ the set of all change states in the LTS of the application B. For example, in the LTS in Fig. 5a, state 6 corresponds to the <flow> activity and represents the different serialization order of the branches. States 2, 12 and 15 model user choices. Non-idempotent partner calls are bf, bc, bl, and thus

$$\mathsf{C}(\mathtt{TAS}) = \{1, 2, 3, 6, 7, 12, 13, 15, 16, 18, 23, 24\},$$

identified in Fig. 5a by shading.

A recovery plan should pass through at least one change state, to allow a change in the execution.

Of course, it is possible that the computed recovery plan passes through a change state which does not affect its outcome, i.e., is irrelevant to the encountered error and its fix. We address computation of "relevant" change states in [22].

4. RUNTIME MONITORING

The runtime monitoring phase uses the set of safety (SM) and liveness (LM) monitors to analyze the BPEL program B as it runs on a BPEL-specific Application Server.

In [24], we have reported on an implementation of a runtime monitoring framework within the IBM WebSphere business integration products[4]. The *interception mechanism* captures events in Σ as they pass between the application server and the program. We use them to update the state of the monitors and store them as part of the execution trace T. This process continues until the current event is about to cause the application termination or entering the error state of one of the monitors. At this point, we stop the computation and begin the recovery process.

Formally, for the LTS $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I)$ we build the trace $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$, where $s_0 \in I$ and $\forall i \cdot a_i \in \Sigma$ and $s_i \in S$. T is a *successful* trace iff $\forall \mathsf{A_i} \in \mathsf{SM} \cup \mathsf{LM}$, $a_0 a_1 \ldots a_{n-1}$ is rejected by $\mathsf{A_i}$. Then $s_n = \mathsf{t}$, the termination state of the application. T is a *failure* (or an *error*) trace iff $\exists \mathsf{A_i} \in \mathsf{SM} \cup \mathsf{LM}$ s.t. $a_0 a_1 \ldots a_{n-1}$ is accepted by $\mathsf{A_i}$. Then state s_n is called an *error state* and is denoted by e. Note that e can be t as well; this occurs when $\mathsf{A_i}$ is a liveness monitor since absence of a desired sequence is determined when the application terminates.

For example, consider the execution of TAS in which the customer chooses the air/ground option but due to communication problems with the flight system partner, the invocation of cf times out and triggers termination of the application, leaving monitor $\mathtt{A_1}$ in its error state 3. This scenario corresponds to the trace t_1 depicted by dotted transitions in Fig. 5a. In another scenario, corresponding to the trace t_2, depicted by dashed transitions in Fig. 5a, the customer attempts to arrive at her destination via an expensive flight (expF) and a limo (bl). Executing this trace leaves monitor $\mathtt{A_2}$ in its error state 4.

Since the application properties are specified separately from the BPEL program, no code instrumentation is required in this step, enabling non-intrusive (and scalable) online monitoring.

[4] http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint

5. RECOVERY PLANS FROM SAFETY PROPERTY VIOLATIONS

Once an error has been detected during runtime monitoring, the goal of the recovery phase is to suggest a number of *recovery plans* that would lead the application away from the error.

DEFINITION 5 (PLAN). *A* plan *is a sequence of actions. A* BPEL recovery plan *is a sequence of actions consisting of user interactions, compensations (empty or not) and calls to service partners.*

Recovery plans differ depending on the type of property that failed. We treat safety properties below, and recovery from liveness properties is described in Sec. 6.

Computing Plans. The recovery procedure for a safety property violation receives $\mathsf{L_C(B)}$ – the LTS of the running application B with compensations (see Sec. 3.1), T – the executed trace ending in an error state e (see Sec. 4) and $\mathsf{C(B)}$ – the set of change states (see Sec. 3.2.2).

In order to recover, we need to "undo" a part of the execution trace, executing available compensation actions, as specified by δ_c. We do this until we either reach a state in $\mathsf{C(B)}$ or the initial state of $\mathsf{L_C(B)}$. Multiple change states can be encountered along the way, thus leading to the computation of multiple plans.

For example, consider the error trace t_2 described in Sec. 4 and shown in Fig 5a. $\{1, 2, 6, 15, 16, 18\}$ are the change states seen along t_2. This leads to the recovery plans shown in Fig 6a. We add state names between transitions for clarity and refer to plans as to mean "recovery to state s". A given plan can also become a prefix for the follow-on one. This is indicated by using the former's name as part of the definition of the latter. For example, recovery to state 16 starts with recovery to state 18 and then includes two more backward transitions, the last one with a non-empty compensation. Plan r_{18} can avoid the error if, after its application, the user chooses a cheap flight instead of an expensive one. Executing plan r_{15} gives the user the option of changing the limousine to a rental car, and plan r_2 – the option of changing from an air/ground combination to just renting a car. Both of these behaviours do not cause the violation of $\mathsf{A_2}$.

Computed plans are then converted to BPEL for presentation to the user. For example, plan r_{18} is shown in Fig 5c. The chosen plan can then be applied (see Sec. 7), allowing the program to continue its execution from the resulting change state.

The exact number of plans is determined by the number of change states encountered along the trace. Since each new plan includes the previous one, the maximum number of plans computed by our tool is set by user preferences either directly ("compute no more than 3 plans") or indirectly ("compute plans of up to length 20" or "compute plans while the overall sum of compensation actions is less than 10").

Discussion. Note that plan r_{16} which cancels the limo, would lead to rebooking it right away which may still leave the possibility of booking an expensive flight and violating the property P_2. The reason why this plan might not be as useful as others is that computation of change states in Sec. 3.2.2 treats all non-idempotent service calls as the same, whereas not all might be *relevant* to the satisfaction of properties of interest. See [22] for a description of computation and evaluation of effectiveness of relevant change states.

6. RECOVERY PLANS FROM LIVENESS PROPERTY VIOLATIONS

Failure of a liveness monitor during execution means that some required actions have not been seen before the application tried to terminate, and the recovery plan should attempt to perform these actions.

The recovery procedure receives A^ℓ – the monitor that identified the violation, $\mathsf{L_C(B)}$ – the LTS of the application, $\mathsf{G(B, A}^\ell\mathsf{)}$ – the set of goal transitions corresponding to A^ℓ, T – the executed trace ending in an error state e, and $\mathsf{C(B)}$ – the set of change states.

A recovery plan effectively "undoes" actions along T, starting with e and ending in a change state (otherwise, the plan would not be executable!) and then "re-plans" the behavior to reach the goal (see Fig. 1b for a schematic view of the overall process). Our solution adapts techniques from the field of *planning* [14], described below.

6.1 Recovery as a planning problem

A *planning problem* is a triple $P = (D, i, G)$, where D is the domain, i is the initial state, and G is a set of goal states.

In addition to P, a planner often gets as input k – the length of the longest plan to search for, and applies various search algorithms to find a plan of actions of length $\leq k$, starting from i and ending in one of the states in G. Typically, the plan is found using heuristics and is not guaranteed to be the shortest available. If no plan is found, the bound k can be increased in order to look for longer plans.

To convert a recovery problem into a planning problem, we use $\mathsf{L_C(B)}$ as the domain and e as the initial state. The third component needed is a set of goal states. Recall that $\mathsf{G(B, A}^\ell\mathsf{)}$ is a set of goal *transitions*. We define $\mathsf{G_s(B, A}^\ell\mathsf{)} = \{s \mid \exists a, s' \cdot (s, a, s') \in \mathsf{G(B, A}^\ell\mathsf{)}\}$. That is, $\mathsf{G_s(B, A}^\ell\mathsf{)}$ is a set of *sources* of transitions in $\mathsf{G(B, A}^\ell\mathsf{)}$. We can now define the planning problem

$$\mathsf{P(B, A}^\ell\mathsf{, T)} = (\mathsf{L_C(B)}, \mathsf{e}, \mathsf{G_s(B, A}^\ell\mathsf{)})$$

Note that when a plan p to a goal state s is computed, we need to extend it with an additional transition, $p \xrightarrow{a} s'$ to account for $(s, a, s') \in \mathsf{G(B, A}^\ell\mathsf{)}$. For example, consider the trace t_1 of Fig. 5a, described in Sec. 4, in which monitor $\mathtt{A_1}$ fails. We define the planning problem $\mathsf{P}(\mathtt{TAS}, \mathtt{A_1}, t_1) = (\mathsf{L_C}(\mathtt{TAS}), 9, \{4\})$, where 9 is the initial state (see Fig. 5a) and $\mathsf{G_s}(\mathtt{TAS}, \mathtt{A_1}) = \{4\}$ (see Sec. 3.2.1). The resulting plan p should be expanded to $p \xrightarrow{\mathsf{rd}} 5$.

Unfortunately, not every trace returned by solving $\mathsf{P(B, A}^\ell\mathsf{, T)}$ is acceptable: the recovery plans for liveness violations should also go through change states. Thus, we cannot simply use a planner as a "black box".

Instead, we look at how planners encode the planning graph and then manipulate the produced encoding directly, to add additional constraints. Consider the LTS in Fig. 7a, which is the planning domain, with s as both the initial and the goal state. The planning graph expanded up to length 3 is shown in Fig. 7b and is read as follows: at time 1 we begin in state s_1. If action a occurs (modeled as a_2), then at time 2 we move to state t (modeled as proposition t_2 becoming true); otherwise, we remain in state s (i.e., proposition s_2 is true). If action b occurs while we are in state t (modeled as b_3), then at time 3 we move to state s. Two plans of length 2 are extracted from this graph: a_2, b_3, corresponding to executing a first, followed by b, and "do nothing" – a

$$
\text{(a)}\quad
\begin{array}{lcl}
r_{18} &=& 4 \xrightarrow{\tau} 21 \xrightarrow{\tau} 20 \xrightarrow{\tau} 19 \xrightarrow{\mathsf{cancelF}} 18 \\
r_{16} &=& r_{18} \xrightarrow{\tau} 17 \xrightarrow{\mathsf{cancelL}} 16 \\
r_{15} &=& r_{16} \xrightarrow{\tau} 15
\end{array}
\qquad
\begin{array}{lcl}
r_6 &=& r_{15} \xrightarrow{\tau} 6 \\
r_2 &=& r_6 \xrightarrow{\tau} 2 \\
r_1 &=& r_2 \xrightarrow{\tau} 1
\end{array}
$$

$$
\text{(b)}\quad
\begin{array}{lcl}
p_0 &=& 9 \xrightarrow{\tau} 8 \xrightarrow{\mathsf{cancelF}} 7 \xrightarrow{\tau} 6 \xrightarrow{\tau} 2 \xrightarrow{\mathsf{onlyCar}} 3 \xrightarrow{\mathsf{bc}} 4 \\
p_1 &=& 9 \xrightarrow{\tau} 8 \xrightarrow{\mathsf{cancelF}} 7 \xrightarrow{\mathsf{bf}} 8 \xrightarrow{\mathsf{cf}} 9 \xrightarrow{\mathsf{exp\_true}} 10 \xrightarrow{\mathsf{expF}} 11 \xrightarrow{\mathsf{getCar}} 12 \xrightarrow{\mathsf{car}} 13 \xrightarrow{\mathsf{bc}} 4 \\
p_2 &=& 9 \xrightarrow{\tau} 8 \xrightarrow{\mathsf{cancelF}} 7 \xrightarrow{\mathsf{bf}} 8 \xrightarrow{\mathsf{cf}} 9 \xrightarrow{\mathsf{exp\_false}} 14 \xrightarrow{\mathsf{cheapF}} 11 \xrightarrow{\mathsf{getCar}} 12 \xrightarrow{\mathsf{car}} 13 \xrightarrow{\mathsf{bc}} 4 \\
p_3 &=& 9 \xrightarrow{\tau} 8 \xrightarrow{\mathsf{cancelF}} 7 \xrightarrow{\mathsf{bf}} 8 \xrightarrow{\mathsf{cf}} 9 \xrightarrow{\mathsf{exp\_false}} 14 \xrightarrow{\mathsf{cheapF}} 11 \xrightarrow{\mathsf{getCar}} 12 \xrightarrow{\mathsf{limo}} 24 \xrightarrow{\mathsf{bl}} 4
\end{array}
$$

Figure 6: Recovery plans for TAS: (a) plans for the safety violation of trace t_2; (b) plans of length ≤ 10 for recovery from the liveness violation of trace t_1.

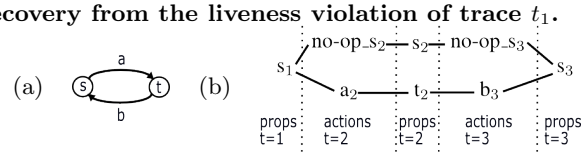


Figure 7: (a) a simple LTS and (b) its encoding as the planning graph of size 3.

planner-specific treatment of a sequence of no-ops.

Several existing planners, such as BlackBox [18], translate the planning graph into a CNF formula and then use a SAT solver, such as SAT4J[5], to find a satisfying assignment for it. Such an assignment, if found, represents a plan. For example, the CNF encoding of the planning graph in Fig. 7b is as follows:

$$
\begin{array}{rcl}
\mathsf{f}_{\mathsf{lts}} &=& (\neg\text{no-op\_}s_2 \vee s_1) \wedge (\neg a_2 \vee s_1) \wedge (\neg\text{no-op\_}s_3 \vee s_2) \\
&& \wedge(\neg b_3 \vee t_2) \wedge (\neg s_2 \vee \text{no-op\_}s_2) \wedge (\neg t_2 \vee a_2) \\
&& \wedge(\neg\text{no-op\_}s_3 \vee s_3) \wedge (\neg b_3 \vee s_3) \wedge (s_1) \wedge (s_3).
\end{array}
$$

Note that it explicitly models pre- and post-conditions of the execution of actions. Such a formula is passed to a SAT solver which produces a satisfying assignment s, if one exists. The desired plan is extracted from s by taking propositions that correspond to actions and that are assigned positive values in s. For the above example, these are a_2, b_3 and "do nothing".

In what follows, we first discuss how to produce a single recovery plan using a SAT-based approach (Sec. 6.2) and then show how to extend it to produce multiple plans (Sec. 6.3).

6.2 Producing a single recovery plan

Let $\mathsf{f_P}$ be the encoding of the planning problem $\mathsf{P}(\mathsf{B}, \mathsf{A}^\ell, \mathsf{T})$ produced by an existing planner. We augment $\mathsf{f_P}$ to follow our "undo until a change state and then redo" approach by adding conjuncts to $\mathsf{f_P}$ with the purpose of restricting its solutions. For efficiency, some additional filtering is done after all plans have been computed (see Sec. 6.4).

1. We want to make sure a recovery plan visits at least one of the change states encountered on the execution trace T. Let $\mathsf{S}(\mathsf{T})$ be the set of states on T. We define $\mathsf{C}(\mathsf{T}) = \mathsf{S}(\mathsf{T}) \cap \mathsf{C}(\mathsf{B})$ to be the change states that appear on T and denote by $c^1, ..., c^n$ the propositions that correspond to states in $\mathsf{C}(\mathsf{T})$. If k is the maximum length of the plan which is being searched for, propositions c_1^j, c_2^j, ..., c_k^j correspond to expansions of c^j to times 1 ... k. For example, consider Fig. 7 again. If t is a change state and $k = 3$, then propositions t_1, t_2, t_3 in $\mathsf{f_{lts}}$ correspond to expansions of t to times 1, 2, 3. We define $\mathsf{c} = (c_1^1 \vee ... \vee c_k^1 \vee ... \vee c_1^n ... \vee c_k^n)$, or, in the case of our example, $\mathsf{c} = (t_1 \vee t_2 \vee t_3)$. This formula is true when at least one of the change states in $\mathsf{C}(\mathsf{T})$ is part of the plan.

2. In order to further lead the planner towards the "undo and then redo" plans, we want to make sure that the only compensations used in the plan correspond to actions in the original trace T. More formally, let $\mathsf{T_C}$ be the set of compensation actions corresponding to the actions in T, and let $\Sigma^c \setminus \mathsf{T_C}$ be all other compensation actions. Let a be a formula which excludes (timed versions) of actions in $\Sigma^c \setminus \mathsf{T_C}$: i.e., neither of these compensation actions is true at any step in the plan. For example, for trace t_1 over the LTS $\mathsf{L_C(TAS)}$ (see Fig. 5a), formula a would exclude all compensations except $\mathsf{cancelF}$ and τ.

We now build a new propositional formula, based on $\mathsf{f_P}$:

$$\mathsf{R_0(f_P)} = \mathsf{f_P} \wedge \mathsf{c} \wedge \mathsf{a}$$

$\mathsf{R_0(f_P)}$ describes the original planning problem for $\mathsf{P}(\mathsf{B}, \mathsf{A}^\ell, \mathsf{T})$, and in addition requires that at least one of the change states is visited and no compensation actions for events that did not occur in T appear in the plan.

6.3 Producing multiple recovery plans

Let π_0 be the plan produced for $\mathsf{R_0(f_P)}$ (see Sec. 6.2), leading to a goal state $g \in \mathsf{G_s}(\mathsf{B}, \mathsf{A}^\ell)$. To give the user options for recovery, we want to produce other plans, different from π_0. The simplest way to do this is to remove g from $\mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)$ and repeat the process described in Sec. 6.2. The new plan will necessarily lead to a different goal transition and thus will be different from π_0. However, this method cannot produce multiple plans to the *same* destination.

Instead, we constrain $\mathsf{R_0(f_P)}$ to explicitly rule out π_0. For example, to rule out the plan a, b for the LTS in Fig. 7a, we use $\mathsf{R_0(f_{lts})}$ computed in Sec. 6.2 and modify it as

$$\mathsf{R_1(f_{lts})} = \mathsf{R_0(f_{lts})} \wedge (\neg a^2 \vee \neg b^3)$$

This guarantees that the plan, if found, is different from the previously found one in at least one action.

We continue this way, restricting $\mathsf{R_i(f_P)}$ with the set of previously computed plans to get $\mathsf{R_{i+1}(f_P)}$, until the number of desired plans is reached or until no new plan can be found, that is, $\mathsf{R_j(f_P)}$ is not satisfiable for some j.

We now apply this method to the `TAS` problem and the error trace t_1 shown in Fig. 5a and ending in state 9. Looking for plans up to length 10, we get plans p_0, p_1 and p_2 shown in Fig. 6b. And, as mentioned earlier, each plan is extended with the last goal transition $4 \xrightarrow{\mathsf{rd}} 5$.

Plan p_0 is the shortest: if unable to obtain a price for the flight, cancel the flight and reserve the car instead. Plans p_1 and p_2 also cancel the flight (since 8 is not a change state whereas 7 is) and then proceed to re-book it and book the car, regardless of the flight's cost. Increasing the plan length, we also get the option of taking the getCar transition out of state 6, book the car and then the flight.

The produced plans are than ranked based on the length of the plan and the cost of compensation actions in it. For

[5] `http://www.sat4j.org/`

example, plan p_0 is the shortest and the additional compensation, for action carAndFlight, is of zero cost. Thus, it is ranked the highest. Of course, this plan does not take into account the time the user will spend driving rather than flying, so she may choose one of the alternative plans instead.

Chosen plans are then converted to BPEL for execution. The compensation part of the plan is similar to the one shown in Fig. 5c, and the re-planning part consists of a sequence of BPEL <invoke> operations.

6.4 Discussion

Precision. Our treatment of goal transitions effectively means that we model satisfaction of the required sequence of actions of a liveness property by executing the last event in the sequence. Thus, our approach may include some plans that do not result in the satisfaction of the desired property (we did not encounter this problem in the examples reported in Sec. 8). One way to approach this problem that we intend to investigate in the future is to define goal *traces*, based on the computation tree of $\mathsf{L}(\mathsf{B})$. While this will lead to the extra precision in plan generation, we expect to pay a potentially steep price in performance.

In addition, we can aim to limit the number of recovery plans computed by taking two issues into consideration: (a) making sure that the plan goes through only "relevant" change states, i.e., those that affect the computation of the violating trace, and (b) removing those plans that result in the violation of one of the safety properties (see [22]).

Controlling unnecessary compensations. Plans p_1, p_2 and p_3 seem to be doing an unnecessary compensation: why cancel a flight and then re-book it if the check flight service call failed? The reason is that the application developer identified service call cf as idempotent. That is, she determined that executing this service again cannot change the flow of control of the application, and thus further compensations are necessary.

Of course, every service call can fail, and thus none are truly idempotent. Yet, having too many change states would undermine the effectiveness of our framework. We believe that the tradeoff we have made in this paper is reasonable but intend to revisit this issue as we gain more experience with the approach.

Furthermore, as plan lengths get large, the planner can generate plans with *compensation loops* which involve doing an action and then immediately undoing it. For example, in recovering from a violation in trace t_1 in LTS $\mathsf{L_C}(\mathsf{TAS})$, shown in Fig. 5a, the plan may include booking a flight and then canceling it several times (i.e., going between states 7 and 8 of $\mathsf{L_C}(\mathsf{TAS})$. Clearly, such situations should be avoided. We could have encoded a corresponding formula as the SAT problem, conjoining it to $\mathsf{R}_0(\mathsf{f_P})$: "at any point in the plan, when a non-compensatory action appears, all follow-on actions should not include compensation". However, we feel that this modification should make SAT computation significantly less efficient. Instead, we filter computed plans so that the ones with compensation loops are not presented to the user.

Can generated plans still fail? There are a number of reasons our plans can fail. The first one, addressed earlier in this subsection, are due to the inherent imprecision of our handling of required event sequences. The second reason is that any service in the recovery plan can fail; thus, the application will be unable to reach its goal, prompting further planning and recovery. Finally, for recovery of safety properties, it is possible that all paths from a change state may still lead the application to an error state. This problem can probably be addressed using additional static analysis.

7. TOOL SUPPORT

We have implemented the process described in this paper using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts. For more information, please refer to [23].

The preprocessing phase (see Sec. 3) receives as input a BPEL program B in BPEL4WS XML format. We use the WS-Engineer extension for LTSA [11] to translate B into an LTS $\mathsf{L}(\mathsf{B})$ and then export it in the Aldebaran format [4], with an extension `.aut`. Since WS-Engineer does not support full handling of BPEL compensations, we built our own `.aut`-to-`.aut` Python script (`add_comp.py`) which uses B and $\mathsf{L}(\mathsf{B})$ to produce $\mathsf{L_C}(\mathsf{B})$ as described in Sec. 3.1. Traceability between the BPEL and the resulting LTS is established by the WS-Engineer's encoding of BPEL scopes into names of LTS actions. This traceability allows us to convert computed plans to BPEL.

The safety and liveness monitors are specified in Aldebaran as well, and we built a script `compute_cp.py` to compute cross-products and identify change states and goal links for them, as described in Sec. 3.2.

The monitoring phase is implemented on top of the IBM WebSphere Process Server. It allows us to intercept events that pass between the application server and the program. In this phase we also build the trace, registering the encountered change states. When recovering for safety properties, we use these states to compute and rank plans.

In the liveness recovery phase, we first use our own script (`gen_plan_prob.py`) to translate $\mathsf{L_C}(\mathsf{B})$ into a planning problem which starts in the error state e and ends in the source of one of the goal transitions (see Sec. 6.2). The planning problem is expressed in STRIPS [8] – an input language to the planner Blackbox [18] which we use to convert it into a CNF formula $\mathsf{f_P}$ (see Sec. 6). Another new script, `GenPlans.java`, modifies $\mathsf{f_P}$ to produce alternative plans, calls the satisfiability solver SAT4J, extracts plans from the satisfying assignments produced by SAT4J, ranks them and converts them to BPEL4WS XML format for displaying and execution. SAT4J is an *incremental* SAT solver, i.e., it saves results from one search and uses them for the next. For our method of generating multiple plans (see Sec. 6.3), where each SAT instance is more restricted than the previous one, this is particularly useful, leading to efficient analysis.

Dynamic workflows [25], implemented in IBM WebSphere Integration Developer 6.2, allow us to execute the generated plans at runtime.

8. CASE STUDIES

In this section, we report on our experience applying our approach to recover from two known vulnerabilities [6] in the Flickr system. [6] modeled each of the aspects of the system as a finite-state machine and showed how to use redundancies in the system in order to "work around" these vulnerabilities. A much bigger example of the use of our framework on the Travel Booking System is reported in [21].

8.1 Examples

App.	Our approach					[6]	
	k	vars	clauses	plans	time (s)	length	plans
FV	15	797	16,198	2	0.04	≤ 2	1
	22	1,436	33,954	4	0.74	≤ 3	5
	26	1,804	44,262	8	1.14	≤ 4	13
	42	3,276	85,494	40	3.12	≤ 8	412
FC	4	42	159	1	0.01	≤ 1	0
	6	95	592	2	0.02	≤ 2	2
	12	321	3,248	4	0.15	≤ 3	8
	16	554	7,393	5	0.27	≤ 4	22
	20	856	14,427	13	1.38	≤ 8	484
TAS	6	135	254	1	0.01	-	-
	8	798	10,355	5	0.13	-	-
	13	1,398	25,023	13	0.27	-	-

Table 1: Plan generation data.

Flicker visibility. Flickr is a web-based photo-management application. Photos are initially uploaded as either *public*, *family* or *private*, and a photo's visibility should be changeable anytime using the setPerm function. The identified vulnerability is "when a photo is initially loaded as *private*, its visibility cannot be changed to *family* at a later date".

We created the Flicker visibility system (FV) by reverse-engineering the model given in [6] and expressing it in BPEL. We then expressed its properties "If a user tries to set a photo's visibility to X, Flickr will guarantee that the photo will have the visibility X", where X is each of the possible visibilities as separate liveness automata. An instance with $X = family$ will "catch" the identified vulnerability in the case where a photo is initially loaded as *private*.

The BPEL model FV, described fully in [20], consists of 28 activities (8 with explicit compensations). Two of these, upload and change are non-idempotent. Converted to LTS, the resulting model has 28 states and 37 transitions.

Flicker comments. Flickr lets users comment on uploaded photos. While any user can add a comment to a *public* photo, only authorized users can comment on *private* and *family* photos. The identified vulnerability is "after uploading a photo as *public*, no comments could be added". Using the same process as for FV, we created the BPEL model FC (see [20]), consisting of 16 activities (6 with compensations). The resulting LTS model has 18 states and 22 transitions. We expressed FC's property "if a user adds a comment to a *public* photo that has comments enabled, the comment should be successfully added to the photo's comments" as a liveness monitor.

8.2 Experience

The number of recovery plans generated for failed traces of FV and FC is shown in Table 1. For example, for the plan length up to 26, we have generated 8 plans for FV. The longest plan was of length 42.

We now look at the effectiveness of the plan generation process. For FV, one of the plans we generate for $k = 22$ is "compensate changes in visibility until the photo becomes *private* again, set the photo visibility to *public* and change visibility to *family*", which corresponds to the workaround plan chosen by [6]. For FC, the plan corresponding to the chosen workaround is "delete the problematic comment, toggle the comments permission and then try to add the comment again", generated when $k = 12$.

To compare the precision of our approach, i.e., the number of plans generated, we look at the list of workaround sequences computed by [6] (see Table 1). The work in [6] modeled the Flickr behavior directly and the model did not include BPEL-induced actions such as entering scopes. Further, the workaround sequences did not include the "going back" part – they were plans on how to execute a task starting from the initial state. Thus, the plans we generate are somewhat longer. For example, the workaround sequences of length ≤ 2 correspond to our plans of length $k = 15$. With this adjustment, Table 1 shows that we generate significantly fewer plans of the corresponding length. We also generate every plan marked by [6] as desired.

Our experience with the Flickr examples suggests that combining simple properties with the compensation mechanism is effective for producing recovery plans.

8.3 Scalability

To check whether SAT-solving done as part of the planning is the bottleneck of our approach, we measured sizes of SAT problems for FV, FC and our running example, TAS, listing them in Table 1. For all three systems, the number of variables and the number of clauses grows linearly with the length of the plan, as expected, and the running time of the SAT solver remains in seconds.

While the web applications we have analyzed have been small (e.g., TAS has 14 activities, and its LTS encoding – 22 states and 27 transitions), our experience suggests that SAT instances used in plan generation remain small and simple and scale well as length of the plan grows. Given that modern SAT solvers can often handle millions of clauses and given that individual web services are intended to be relatively compact (with tens rather than thousands of partner calls), we have a good reason to believe that our approach to plan generation is scalable to realistic systems.

9. RELATED WORK

The main contribution of our work is a recovery framework for web applications via planning. Bertoli et al. [3] used planning for the synthesis of web service *orchestrations*. In contrast, we assume the orchestration is given, and use planning to help recover when an error is detected.

Several works have suggested "self-healing" mechanisms for web-service applications. The Dynamo framework [2] uses *annotation rules* in BPEL in order to allow recovery once a fault has been detected. Such rules need to be installed by the developers before the system can function. In contrast, our work uses an existing compensation mechanism and requires no extra effort from developers.

[13] propose a framework for self-healing web services, where all possible faults and their repair actions are predefined in a special registry. This approach relies on being able to identify and create recovery from all available faults. Our approach uses compensations for *individual* actions and can dynamically recover from errors as they are detected.

[7] uses fault tolerance patterns to transform the original BPEL process into a fault-tolerant one at compile time. It is done by adding redundant behavior to the application which may result in a significantly bigger, and slower, program. Our work is non-intrusive and does not slow down the application if no errors are found.

The work of Carzaniga et al. [6] is the closest to ours in spirit. It exploits redundancy in web applications to find workarounds when errors occur, assuming that the application is given as a finite-state machine, with an identified error state as well as the "fallback" state to which the application should return. The approach generates all possible recovery plans, without prioritizing them. In contrast, our

framework not only detects runtime errors but also calculates goal and change states and in addition automatically filters out unusable recovery plans (those that do not include change states) and ranks the remaining ones. See Sec. 8 for a detailed comparison.

10. CONCLUSION AND FUTURE WORK

In this paper, we have used BPEL's compensation mechanism to define and implement an online system for suggesting, ranking and executing recovery plans. Our experience has shown that this approach computes a small number of highly relevant plans, doing so quickly and effectively.

We have evaluated our approach on relatively small and simple examples. While we expect web service applications to be small, it is still important to conduct further case studies to assess scalability and, more importantly, usability of our approach. Furthermore, throughout the paper we have identified several precision issues related to the identification of goals and change states. We intend to apply static analysis techniques to help improve it and conduct further experiments to better understand the tradeoffs between the more expensive analyses and the effective computation of recovery plans. Some preliminary work towards this end is reported in [22].

Another limitation of our approach is that we model compensations as going back to states visited earlier in the run. While this model is simple, clean and enables effective analysis, the compensation mechanism in languages like BPEL allows the user to execute an arbitrary operation and thus end up in a principally different state. In fact, our approach will encounter this situation as soon as we start modeling data in addition to control. For example, if we model the amount of money the user has as part of the state, then booking and then canceling a flight brings her to a different state – the one where she has less money and no flight. Thus, extending our framework to situations where compensation affects data remains a challenge.

In fact, reasoning about properties which involve the actual *data* exchanged by conversation participants may be challenging from the perspective of expressing the properties and converting them into monitoring automata as well as from the scalability perspective (e.g., computing the goal links, expressing the formal model of BPEL with data as a state machine, etc.).

Acknowledgements

We thank IBM CAS Toronto (specifically, Bill O'Farrell, Elena Litani and Leho Nigel), members of the IFIP 2.9 WG, and FSE anonymous reviewers for their helpful feedback on this work and its presentation. This research has been funded by NSERC, IBM Toronto, MITACS, and by the Ontario Post-Doctoral Fellowship program.

11. REFERENCES

[1] M. Autili, P. Inverardi, and P. Pelliccione. A Scenario Based Notation for Specifying Temporal Properties. In *Proc. of SCESM (at ICSE'06)*, 2006.

[2] L. Baresi and S. Guinea. Dynamo and Self-Healing BPEL Compositions. In *Proc. of ICSE'07 (Companion)*, pages 69–70, 2007.

[3] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Continuous Orchestration of Web Services via Planning. In *Proc. of ICAPS'09*, 2009.

[4] M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the ALDÉBARAN Toolset. *STTT*, 1(1-2):166–184, 1997.

[5] T. Bultan. Modeling Interactions of Web Software. In *Proc. of WWV'06*, pages 45–52, 2006.

[6] A. Carzaniga, A. Gorla, and M. Pezze. Healing Web Applications through Automatic Workarounds. *STTT*, 10(6):493–502, 2008.

[7] G. Dobson. Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In *Proc. of EUROMICRO'06*, pages 126–133, 2006.

[8] R. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proc. of IJCAI'71*, pages 608–620, 1971.

[9] H. Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, Imperial College, 2006.

[10] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model Checking Service Compositions under Resource Constraints. In *Proc. of ESEC-FSE '07*, pages 225–234, 2007.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a Tool for Model-Based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774, 2006.

[12] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630, 2004.

[13] M. Fugini and E. Mussi. Recovery of Faulty Web Applications through Service Discovery. In *Proc. of SMR'06*, 2006.

[14] F. Giunchiglia and P. Traverso. Planning as Model Checking. In *Proc. of ECP'99*, pages 1–20, 1999.

[15] R. Grosu and S. A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *Proc. of ACSD'05*, pages 6–14, 2005.

[16] K. Havelund and G. Rosu. Testing Linear Temporal Logic Formulae on Finite Execution Traces. Technical report, 2001.

[17] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proc. of BPM'05*, volume 3649 of *LNCS*, pages 220–235, 2005.

[18] H. A. Kautz and B. Selman. Unifying SAT-based and Graph-based Planning. In *Proceedings of IJCAI'99*, pages 318–325, 1999.

[19] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[20] J. Simmonds. *Dynamic Analysis of Web Services*. PhD thesis, Department of Computer Science, University of Toronto, 2010. (in preparation).

[21] J. Simmonds, S. Ben-David, and M. Chechik. Monitoring and Recovery of Web Service Applications. In *Smart Internet*, LNCS, pages 1–35. Springer, 2010. To appear.

[22] J. Simmonds, S. Ben-David, and M. Chechik. Optimizing Computation of Recovery Plans for BPEL Applications, 2010. Submited.

[23] J. Simmonds, S. Ben-David, and M. Chechik. RuMoR: Monitoring and Recovery of BPEL Applications, May 2010. Submitted.

[24] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse. Runtime Monitoring of Web Service Conversations. *IEEE Trans. on Serv. Comp.*, 2009.

[25] W. M. P. van der Aalst and M. Weske. Case Handling: a New Paradigm for Business Process Support. *Data Knowl. Eng.*, 53(2):129–162, 2005.