Optimizing Computation of Recovery Plans for BPEL Applications

Jocelyn Simmonds Shoham Ben-David Marsha Chechik

University of Toronto
Toronto, ON, Canada
{jsimmond,shoham,chechik}@cs.toronto.edu

Web service applications are distributed processes that are composed of dynamically bounded services. In our previous work [15], we have described a framework for performing runtime monitoring of web service against behavioural correctness properties (described using property patterns and converted into finite state automata). These specify forbidden behavior (safety properties) and desired behavior (bounded liveness properties). Finite execution traces of web services described in BPEL are checked for conformance at runtime. When violations are discovered, our framework automatically proposes and ranks recovery plans which users can then select for execution. Such plans for safety violations essentially involve "going back" – compensating the executed actions until an alternative behaviour of the application is possible. For bounded liveness violations, recovery plans include both "going back" and "re-planning" – guiding the application towards a desired behaviour. Our experience, reported in [16], identified a drawback in this approach: we compute too many plans due to (a) overapproximating the number of program points where an alternative behaviour is possible and (b) generating recovery plans for bounded liveness properties which can potentially violate safety properties. In this paper, we describe improvements to our framework that remedy these problems and describe their effectiveness on a case study.

1 Introduction

A BPEL application is an orchestration of (possibly third-party) web services. These services, which can be written in a variety of languages, communicate through published interfaces. Third-party services can be dynamically discovered, and may be modified without notice. BPEL includes mechanisms for dealing with termination and for specifying compensation actions (these are defined on a "per action" basis, i.e., a compensation for booking a flight is to cancel the booking); yet, they are of limited use since it is hard to determine the state of the application after executing a set of compensations.

In [15], we proposed a framework for runtime monitoring and recovery that uses user-specified behavioral properties to automatically compute recovery plans. This framework takes as input the target BPEL application, enriched with the compensation mechanism that allows us to undo some of the actions of the program, and a set of properties (specified as desired/forbidden behaviors). When a violation of a property is detected at runtime, this framework outputs a set of ranked recovery plans and enables applying the chosen plan to continue the execution. Such plans for safety violations consist just of the "going back" part, until an alternative behavior of the application is possible. For bounded liveness violations, recovery plans include both the "going back" and the "re-execution" part – guiding the application towards a desired behavior (such plans are schematically shown using a dashed line in Figure 4).

For example, consider the Travel Booking System (TBS) shown in Figure 2, which provides travel booking services over the web. In a typical scenario, a customer enters the expected travel dates, the destination city and the rental car location – airport or hotel. The system searches for available flights, hotel rooms and rental cars, placing holds on the resources that best satisfy the customer preferences. If

Submitted to:
TAV-WEB 2010

the customer chooses to rent a car at the hotel, the system also books the shuttle between the airport and the hotel. If the customer likes the itinerary presented to him/her, the holds are turned into bookings; otherwise, the holds are released. Some correctness properties of TBS are P_1: "there should not be a mismatch between flight and hotel dates" (expressing a safety property, or a forbidden behavior), P_2: "a car reservation request will be fulfilled regardless of the location (i.e., airport or hotel) chosen" (expressing a bounded liveness property or a desired behavior), and P_3: "ground transportation must not be picked before a flight is reserved" (forbidden behavior).

If the application exhibits a forbidden behavior, our framework suggests plans that 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 non-idempotent partner calls (i.e., those where a repeated execution with the same arguments may yield a different outcome) [15]. For example, if the TBS system produces an itinerary with inconsistent dates, a potential recovery plan might be to cancel the current hotel booking and make a new reservation that is consistent with the booked flight's dates.

Another possibility is that the system fails to produce a desired behavior when calls to some partners terminate, leaving it in an unstable state. In such cases, our framework computes plans that redirect the application towards executing new activities, those that lead to completion of the desired behavior. For example, if the car reservation partner for the hotel location fails (and thus the "shuttle/car at hotel" combination is not available), the recovery plans would be to provide transportation to the user's destination (her "goal" state) either by trying to book another car at the hotel, or by undoing the shuttle reservation and try to reserve the rental car from the airport instead.

Effectiveness and scalability of a recovery framework like ours is in (quickly) generating a small number of highly relevant plans. While our framework can generate recovery plans as discussed above, in our experience with TBS, reported in [16], we observed that it generates too many plans. At least two factors contribute to this problem:

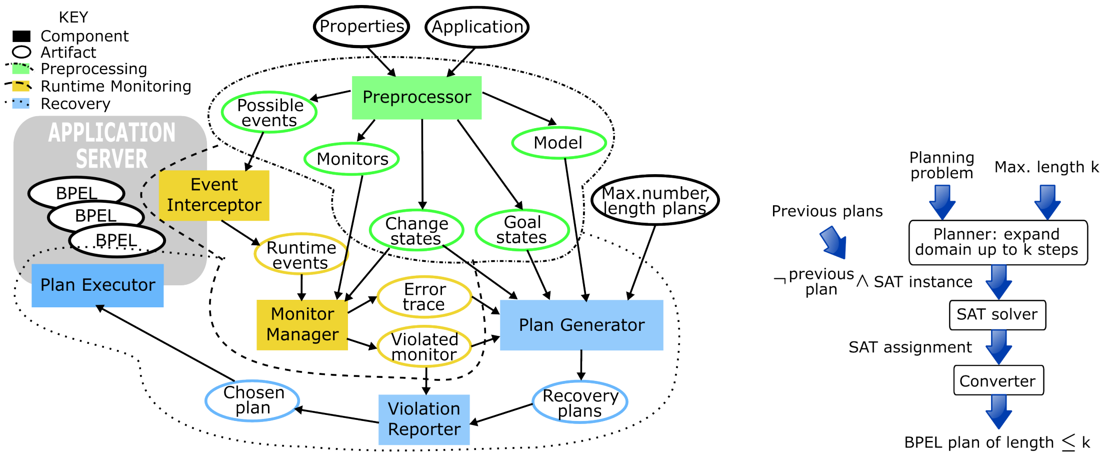
1. we over-approximate the set of change states and thus offer plans where compensation cannot produce an alternative path through the original system to avoid an error; and
2. some recovery plans for desired behavior violations will (necessarily) lead to violations of forbidden behaviors when executed, and thus should not be offered to the user.

In this paper, we present two improvements that try to address these issues. The first improvement identifies the non-idempotent service calls that are relevant to the violation, i.e., their execution may affect the control flow of the current execution. The second improvement identifies computed plans that always lead to violations of forbidden behaviors, as the execution of these plans will cause another runtime violation and thus they should not be offered to the user.

In what follows, we give a brief overview of the framework (Sec. 2) and our previous experience with the Travel Booking System (TBS) (Sec. 3). In Sec. 4, we use the TBS example to discuss the two plan generation improvements as well as their effectiveness. We conclude in Sec. 5 with a summary of the paper, related work and suggestions for future work.

2 Overview of the Approach

We have implemented our RUntime MOnitoring and Recovery framework (RUMOR) using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts. The architecture of our tool is shown in Fig. 1a, where components and artifacts have been grouped by phase (Preprocessing, Monitoring or Recovery). In the Preprocessing phase, the correctness properties specifying desired and



(a) (b)

Figure 1: a) Architecture of the tool; b) Recovery plan generation for violating a desired behavior.

forbidden behaviors are turned into finite-state automata (monitors). We use the WS-Engineer extension for LTSA [6] to translate the BPEL application into a Labeled Transition System (LTS), enriched with compensation actions (model). We also compute change and goal states during this phase.

The Monitoring phase is implemented on top of the IBM WebSphere Process Server[1], a BPEL-compliant process engine for executing BPEL processes. Monitoring is done in a non-intrusive manner – the Event Interceptor component intercepts runtime events and sends them to the Monitor Manager, which updates the state of the monitors. The use of 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. RUMOR does not require any code instrumentation, does not significantly affect the performance of the monitored system (see [17]), and enables reasoning about partners expressed in different languages.

During the Recovery phase, the Plan Generator component generates recovery plans using SAT-based planning techniques (see [15] for details). In the case of forbidden behavior violations, the Plan Generator determines which visited change states are reachable by executing available compensation actions. Multiple change states can be encountered along the way, thus leading to the computation of multiple plans. In the case of desired behavior violations, the Plan Generator tries to solve the following planning problem: "From the current state in the system, find all plans (up to length k) to achieve the goal, that go through a change state". The actions that a plan can execute are defined by the application itself; thus, the domain of the planning problem is the LTS model of the application. The initial and goal states of the planning problem are the current error state and the precomputed goal states, respectively.

The process for computing recovery plans for desired behavior violations is shown in Fig. 1b. RUMOR uses Blackbox [11], a SAT-based planner, to convert the planning problem into a SAT instance. The maximum plan length is used to limit how much of the application model is unrolled in the SAT instance, effectively limiting the size of the plans that can be produced. Multiple plans are generated by modifying the initial SAT instance: new plans are obtained by ruling out those computed previously. Plans are extracted from the satisfying assignments produced by the SAT solver SAT4J and converted into BPEL for displaying and execution. SAT4J is an *incremental* SAT solver, i.e., it saves results from one search and uses them for the next. We take advantage of this for generating multiple plans.

All computed plans are presented to the application user through the Violation Reporter component.

[1] http://www-306.ibm.com/software/integration/wps/

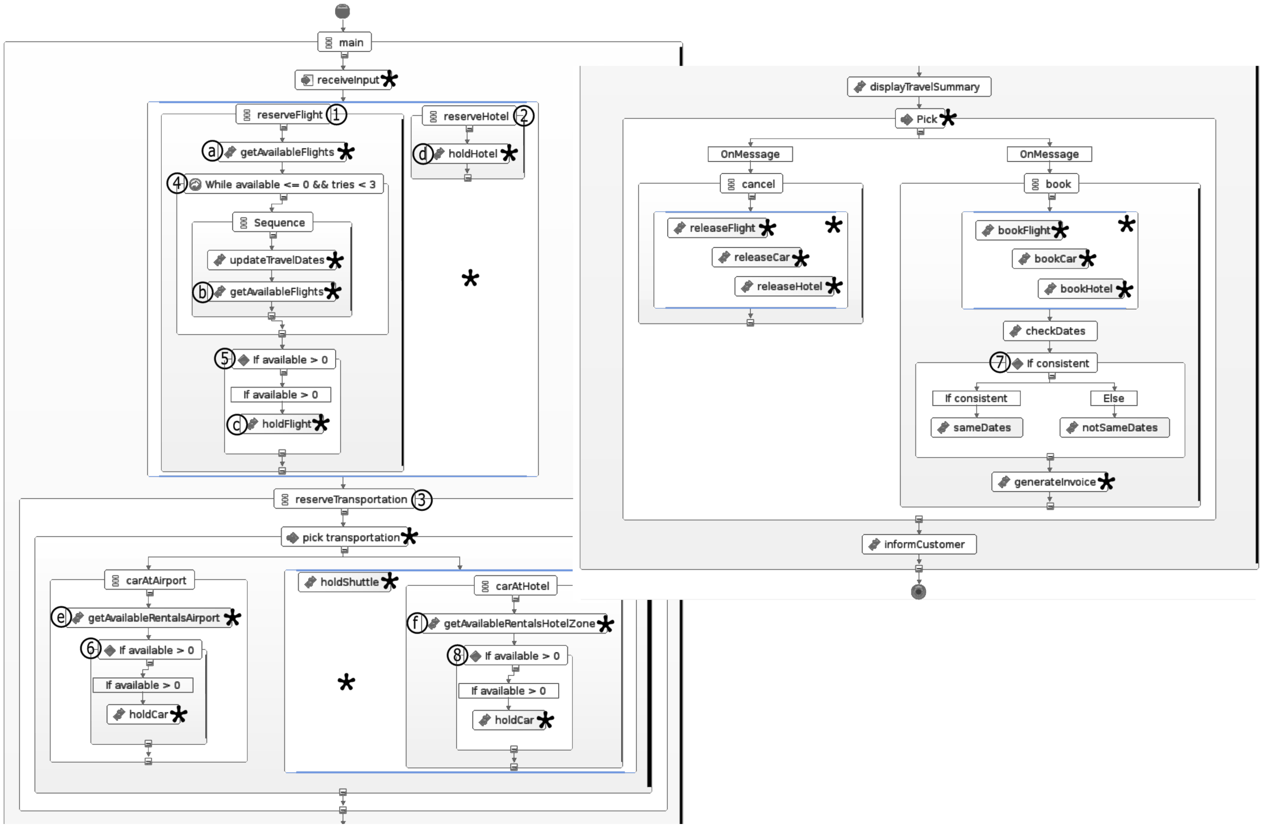


Figure 2: BPEL implementation of the Travel Booking System.

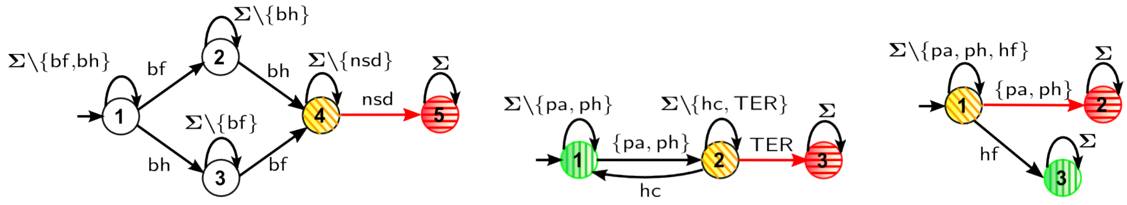
It generates a web page snippet with violation information as well as a web page for selecting a recovery plan. The application developer must include this snippet in the default error page, so that the computed recovery plans are displayed as part of the application when an error is detected. The Plan Executor executes the selected plan using dynamic workflows [18]. RuMoR takes advantage of their implementation as part of IBM WebSphere.

3 Monitoring The Travel Booking System

3.1 BPEL Model

Figure 2 shows the BPEL implementation of this system. TBS interacts with three partners (FlightSystem, HotelSystem and CarSystem), each offering the services to find an available resource (flight, hotel room, car and shuttle), place a hold on it, release a hold on it, book it and cancel it. Booking a resource is compensated by canceling it and placing a hold is compensated by a release. All other activities can be simply undone, i.e., they do not have explicit compensation actions. All external service calls are non-idempotent. In the rest of this paper, bf, bh and hc represent the service calls bookFlight, bookHotel and holdCar, respectively.

The workflow begins by <receive>'ing input (receiveInput), followed by <flow> (i.e., parallel composition) with two branches, since the flight and hotel reservations can be made independently. The branches are labeled ① and ②: ①) find and place a hold on a flight, ②) place a hold on a hotel room



(a) (b) (c)

Figure 3: Monitors: (a) $\mathtt{A}_1$, (b) $\mathtt{A}_2$ and (c) $\mathtt{A}_3$. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

(this branch has been simplified in this case study). If there are no flights available on the given dates, the system will prompt the user for new dates and then search again (up to three tries). After making the hotel and flight reservations, the system tries to arrange transportation (see the <pick> (i.e., making the external choice) activity labeled ③): the user <pick>'s a rental location (pickAirport or pickHotel, abbreviated as pa and ph, respectively) and the system tries to place holds on the required resources (car at airport, or car at hotel and a shuttle between the airport and hotel).

Once ground transportation has been arranged, the reserved itinerary is displayed to the user (display−TravelSummary), and at this point, the user must <pick> to either book or cancel the itinerary. The book option has a <flow> activity that invokes the booking services in parallel, and then calls two local services: one that checks that the hotel and flight dates are consistent (checkDates), and another that generates an invoice (generateInvoice). The result of checkDates is then passed to local services to determine whether the dates are the same (sameDates) or not (notSameDates, abbreviated as nsd). The cancel option is just a <flow> activity that invokes the corresponding release services. Whichever option is picked by the user, the system finally invokes another local service to inform the user about the outcome of the travel request (informCustomer).

3.2 Monitoring Behavioral Properties

In general, the framework described in [15] allows the system developer to express desired and forbidden behavior as bounded liveness and safety properties, respectively. These are expressed using property patterns [4], converted into quantified regular expressions (QRE) [13] and then become monitoring automata. For example, the TBS properties described in Sec. 1 are expressed as follows:

P_1: **Absence** of a date mismatch event (notSameDate) **After** both a flight and hotel have been booked (bookFlight and bookHotel, in any order).

P_2: **Globally** hold a car (holdCar) in **Response** to a rental location selection (pickHotel or pickAirport).

P_3: **Existence** of flight reservation (holdFlight) **Before** the rental location selection (pickHotel or pickAirport).

In our framework, monitors are finite state automata that accept *bad* computations. In order to facilitate recovery, we assign colors to the monitor states:

- Accepting states are colored red, signaling violation of the property. State 5 of Fig. 3a, state 3 in Fig. 3b, and state 2 in Fig. 3c are red states.
- Yellow states are those from which a red state can be reached through a single transition. State 4 in Fig. 3a, state 2 in Fig. 3b, and state 1 in Fig. 3c are yellow states.
- Green states are states that can serve as good places to which a recovery plan can be directed. We define green states to be those states that are not red or yellow, but that can be reached through a single transition from a yellow state. State 1 in Fig. 3b, and state 3 in Fig. 3c are green states.

Scenario	k	Change states	Vars	Clauses	Plans	Time (s)
t_1	5	2	–	–	2	0.01
	10	5	–	–	5	0.02
	15	8	–	–	8	0.02
	20	12	–	–	12	0.02
	25	13	–	–	13	0.02
	30	13	–	–	13	0.02
t_2	5	4	108	464	0	0.01
	10	7	883	30,524	2	0.14
	15	10	1,456	74,932	8	1.37
	20	10	2,141	135,047	18	4.72
	25	10	3,298	246,210	60	29.16
	30	10	5,288	464,654	68	61.34

Table 1: Plan generation data. "–" mark cases which are not applicable, such as references to SAT for recovery from forbidden behavior violations.

The details of the QRE translation and the formal definition of state colors can be found in [16].

The monitor $\mathtt{A}_1$ in Fig. 3a enters its error state (5) when the application determines that the hotel and flight booking dates do not match (the hotel and flight can be booked in any order). The monitor $\mathtt{A}_2$ in Fig. 3b represents property P_2: if the application terminates (i.e., sends the TER event) before hc appears, the monitor moves to the (error) state 3. State 1 is a good state since the monitor enters it once a car has been placed on hold (hc). The monitor $\mathtt{A}_3$ in Fig. 3c represents property P_3: it enters the good state 3 once a hold is placed on a flight (hf), and enters its error state 2 if the rental location (pa or ph) is picked before a flight is reserved (hf).

3.3 From 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 [7, 10, 14]. In this work, we use Foster's [5] approach of using a Labeled Transition System (LTS) as the underlying formalism.

Definition 1 (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 $(s, a, s') \in \delta$. An *execution*, or a *trace*, of an LTS is a sequence $\mathsf{T} = s_0 a_0 s_1 a_1 s_2 ... a_{n-1} s_n$ such that $\forall i, 0 \leq i < n$, $s_i \in S$, $a_i \in \Sigma$ and $s_i \xrightarrow{a_i} s_{i+1}$.

The set of labels Σ is derived from the possible application events: service invocations and returns, messages, scope entries, and conditional valuations. [5] specifies the mapping of all BPEL 1.0 activities into LTS. Conditional activities like <if> and <while> statements are represented as states with two outgoing transitions, one for each valuation of the condition. <pick> is also a conditional activity, but can have one or more outgoing transition, one for each <onMessage> branch. <sequence> and <flow> activities result in the sequential and parallel composition of the enclosed activities, respectively.

In [15], we describe how we augmented Foster's translation so that we can model termination, as well as BPEL compensation. According to our translation, the TBS LTS has 52 states and 67 transitions, and $|\Sigma| = 33$. 20 of the BPEL activities (highlighted with a ★ symbol in Figure 2) yield a total of 35 change states in the LTS.

3.4 Experience: Recovery from a safety property violation

We generated a recovery plan for the following scenario (called trace t_1, of length $k = 21$) which violates property P_1: The application first makes a hotel reservation (holdHotel) and then prompts the user for new travel dates (updateTravelDates), since there were no flights available on the current travel dates. The car rental location is the airport (pickAirport). The system displays the itinerary (displayTravelSummary) but the user does not notice the date inconsistency and decides to book it. The TBS makes the bookings (bookFlight, bookHotel and bookCar) and then checks for date consistency (checkDates). Since the dates are not the same (notSameDates), we detect violation of P_1 and initiate recovery.

We generated plans starting with length $k = 5$ and going to $k = 30$ in increments of 5. In order to generate all possible plans for each k, we chose n – the maximum number of plans generated – to be MAX_INT. Table 1 summarizes the results. A total of 13 plans were generated, and the longest plan, which reaches the initial state, is of length 21 (and thus the rows corresponding to $k = 25$ and $k = 30$ contain identical information). Since t_1 violates a safety property, no SAT instances were generated, and the running time of the plan generation is trivial.

The following plans turn t_1 into a successful trace: p^1_A – cancel the flight reservation and pick a new flight using the original travel dates, and p^1_B – cancel the hotel reservation and pick a new hotel room for the new travel dates. Our tool generated both of these plans, but ranked them 11th and 12th (out of 13), respectively. They were assigned a low rank due to the interplay between the following two characteristics of our case study: (i) the actual error occurs at the beginning of the scenario (in the flight and hotel reservation <flow>), but the property violation was only detected near the end of the workflow (in the book flow), and (ii) t_1 passes through a relatively large number of change states, and thus many recovery plans are possible.

The first of these causes could be potentially fixed by writing "better" properties – the ones that allows us to catch an error as soon as it occurs. We recognize, of course, that this can be difficult to do. The second stems from the fact that not all service calls marked as non-idempotent are relevant to P_1 or its violation. In Sec. 4.1, we present a method for identifying the non-idempotent service calls that are relevant to the violation, i.e., their execution may affect the control flow of the current execution. By reducing the number of change states considered, fewer recovery plans will be generated.

3.5 Experience: Recovery from a bounded liveness property violation

The following scenario (we call it trace t_2, with length 14) violates property P_2. Consider an execution where the user reserves a hotel room (reserveHotel), and a flight (reserveFlight). He then chooses to rent a car at the hotel (pickHotel), but no cars are available at that hotel. TBS makes flight, hotel and shuttle reservations (holdFlight and holdHotel), but never makes a car reservation (holdCar). The user does not notice the missing reservation in the displayed itinerary (displayTravelSummary) and decides to book it. The TBS tries to complete the bookings, first booking the hotel (bookHotel) and then the car (bookCar). When the application attempts to invoke bookCar, the BPEL engine detects that the application tries to access a non-initialized process variable (since there is no car reservation), and issues a TER event. Rather than delivering this event to the application, we initiate recovery.

We are again using $n =$ MAX_INT and varying k between 5 and 30, in increments of 5, summarizing the results in Table 1. The first thing to note is that our approach generated a relatively large number of plans (over 60) as k approached 30. While in general the further we move away from a goal link, the more alternative paths lead back to it, this was especially true for TBS which had a number of <flow> activities. The second thing to note is that our analysis remained tractable even as the length of the plan and the number of plans generated grew (around 1 min for the most expensive configuration).

Executing one of the following plans would leave TBS in a desired state: p_A^2 – attempt the car rental at the hotel again, and p_B^2 – cancel the shuttle from the airport to the hotel and attempt to rent a car at the airport. Unlike t_1, the error in this scenario was discovered soon after its occurrence, so plans p_A^2 and p_B^2 are the first ones generated by our approach. p_A^2 actually corresponds to two plans, since the application logic for reserving a car at a hotel is in a <flow> activity, enabling two ways of reaching the same goal link. Plan p_B^2 was the 3rd plan generated.

The rest of the plans we generated compensate various parts of t_2, and then try to reach one of the three goal links. While these longer plans include more compensations and are ranked lower than p_A^2 and p_B^2, we still feel that it may be difficult for the user to have to sift through all of them. As in the case of safety property violations, we can reduce the number of plans generated by picking relevant change states. Furthermore, some of the computed recovery plans, when executed, lead to violations of safety properties, and thus should not be offered to the user. In Sec. 4.2, we present a method for identifying such recovery plans that always lead to violations of safety properties.

4 Reducing the Number of Generated Plans

As discussed above, our tool produces a set of recovery plans for each detected violation. However, in some cases this set includes unusable plans. In this section, we look at techniques for filtering out two types of unusable plans: those that require going through unnecessary change states, where re-executing the partner call cannot affect the (negative) outcome of the trace (see Sec. 4.1), and plans that fix a liveness property at the expense of violating some safety properties (see Sec. 4.2).

4.1 Relevant Change States

As discussed before, change states are application states from which *flow-changing* actions can be executed. These are user choices (<pick>), modeling the <flow> activity, and service calls whose outcomes are not completely determined by their input parameters (to which we refer as non-idempotent). For example, getAvailableFlights is a non-idempotent service call (and leads to the identification of various change states), since each new invocation of the service, with the same travel dates, may return different available flights. Non-idempotent service calls are identified by the developer.

Let us reexamine the trace t_1. This trace visited 13 change states, of which 11 correspond to non-idempotent service calls. The two flow activities executed on the trace identify two change states that coincide with two states already identified using non-idempotent service calls (holdHotel and bookCar). The remaining two change states correspond to the two <pick> activities on the trace (choice between rental locations, and choice between booking/canceling the itinerary).

As <pick> and <flow> activities are flow-altering actions by definition, the change states identified by these activities are always relevant to the current violation. On the other hand, not all service calls marked as non-idempotent are relevant, i.e., their execution cannot modify the current execution trace. For example, bookFlight and bookHotel are both non-idempotent service calls that appear in t_1, and so define two recovery plans. However, these two plans are not useful: after their execution, the application is forced to complete the execution of t_1 in its entirety. This happens because none of the later control predicates depend on the output produced by these service calls. This example suggests a definition of *relevant* change state:

Definition 2 (Relevant Change State) *A change state is relevant if it is identified by: 1) a <flow> or <pick> activity, or 2) a non-idempotent service call, and a variable that appears in a control activity is data dependent on the outcome of this service call.*

Trace	Id	Activity	Label	Predicate	Non-idempotent service calls
t_1	1	<while>	④	availableFlights <= 0&&tries < 3	getAvailableFlights labeled ⓐ, ⓑ
t_1	2	<if>	⑤	availableFlights > 0	getAvailableFlights labeled ⓐ, ⓑ
t_1	3	<if>	⑥	availableCars > 0	getAvailableRentalsAirport labeled ⓔ
t_1	4	<if>	⑦	consistent == true	holdHotel, holdFlight, labeled ⓓ, ⓒ, respectively
t_2	5	<if>	⑥	availableFlights <= 0	getAvailableFlights labeled ⓐ
t_2	6	<if>	⑧	availableCars <= 0	getAvailableRentalsHotel labeled ⓕ

Table 2: Predicates appearing on both traces, and the non-idempotent service calls that affect their values.

In order to carry out the data dependency analysis on the application LTS, we must first determine which BPEL activities define and use process variables, and how to map this information to the LTS model. <invoke> and <assign> activities both define and use variables. For example, the getAvailableFlights service call takes as input the travelRequest variable (use) and modifies the availableFlights variable (definition). Both <while> and <if> activities use the variables that appear in the activity's predicate. <flow> and <pick> do not use or define variables.

We can now define the following two sets of variables for each LTS transition ($s \xrightarrow{a} s'$): the set of variables defined by the action a (Def($s \xrightarrow{a} s'$)), and the set of variables used by action a (Use($s \xrightarrow{a} s'$)). Formally:

$$\mathrm{Def}(s \xrightarrow{a} s') = \begin{cases} \{\mathsf{inVar}\} & \text{if } a \text{ represents <invoke } \ldots \text{ inputVariable = "inVar" } \ldots \text{>} \\ \{\mathsf{fromVar}\} & \text{if } a \text{ represents <assign><from>fromVar</from>} \ldots \text{>} \\ \emptyset & \text{otherwise} \end{cases} \quad \text{and}$$

$$\mathrm{Use}(s \xrightarrow{a} s') = \begin{cases} \{\mathsf{outVar}\} & \text{if } a \text{ represents <invoke } \ldots \text{ outputVariable = "outVar" } \ldots \text{>} \\ \{\mathsf{toVar}\} & \text{if } a \text{ represents <assign><to>toVar</to>} \ldots \text{>} \\ \{\mathsf{v_1}, \mathsf{v_2}, \ldots \mathsf{v_n}\} & \text{if } a \text{ represents a <while> or <if> branch, and } \{\mathsf{v_1}, \mathsf{v_2}, \ldots \mathsf{v_n}\} \\ & \quad \text{appear in the corresponding <condition>} \\ \emptyset & \text{otherwise} \end{cases}$$

The set of variable definitions that occur on a trace is the union of the definitions that occur on the individual transitions of the trace: for a trace $\mathsf{T} = s_0 a_0 s_1 a_1 s_2 \ldots a_{n-1} s_n$, $\mathrm{Def}(\mathsf{T}) = \bigcup_i \mathrm{Def}(s_i \xrightarrow{a_i} s_{i+1})$. Now we can define *direct data dependency*: a transition v is directly data dependent on another transition u if and only if v uses a variable defined by u, and there is a path from u to v where this variable is not redefined. Formally,

Definition 3 (Directly Data Dependent) *A transition* $(q \xrightarrow{b} q')$ *is* directly data dependent *on a transition* $(p \xrightarrow{a} p')$ *if and only if there is a trace* $\mathsf{T} = s_0 a_0 s_1 a_1 s_2 \ldots a_{n-1} s_n$ *such that* $p' = s_0$*,* $q = s_n$ *and* $\big(Def(p \xrightarrow{a} p') \bigcap Use(q \xrightarrow{b} q')\big) - Def(\mathsf{T}) \neq \emptyset$*.*

For example, the <if> activity labeled ⑤ and the holdFlight service call labeled ⓒ are both directly data dependent on the getAvailableFlights service calls at ⓐ and ⓑ.

We can now define *data dependency*: a transition v is data dependent on another transition u if there exists a path from u to v that can be divided into sections, where each section is directly data dependent on a previous section. For example, the bookFlight service call is directly data dependent on the invocation of holdFlight, so bookFlight is data dependent on both invocations of the getAvailableFlights service.

Now apply data dependency analysis on trace t_1. This trace executed four control activities: 1) the <while> labeled ④, 2) the <if> labeled ⑤, 3) the <if> labeled ⑥, and 4) the <if> labeled ⑦. Table 2 lists the corresponding predicates, as well as the non-idempotent service calls that can affect the values of these predicates. For example, the <while> condition is availableFlights <= 0&&tries < 3. This use of the availableFlights variable is directly data dependent on both appearances of the non-idempotent

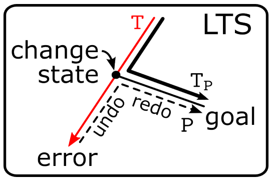


Figure 4: A schematic view on plan generation and filtering.

Scenario	k	Baseline (from Table1)		Relevant Change States		Avoiding Forbidden Behaviors	Both improvements
		change states	plans	change states	plans	plans	plans
t_1	5	2	2	0	0	–	–
	10	5	5	2	2	–	–
	15	8	8	5	5	–	–
	20	12	12	8	8	–	–
	25	13	13	8	8	–	–
	30	13	13	8	8	–	–
t_2	5	4	0	2	0	0	0
	10	7	2	4	2	2	2
	15	10	8	6	5	8	5
	20	10	18	6	15	11	8
	25	10	60	6	41	32	23
	30	10	68	6	41	38	23

Table 3: Results of applying both improvements (separately, and then combined) to the TBS case study. "–" marks cases which are not applicable, since the second improvement only applies to bounded liveness properties.

getAvailableFlights service. On the other hand, the tries variable is not data dependant on any non-idempotent service calls, since it is updated by an <assign> statement inside the <while> activity.

The data dependency analysis for predicates 2 and 3 is similar to that of predicate 1, and the results of the analysis are summarized in Table 2. In the case of predicate 4, the variable consistent is directly data dependent on the idempotent service checkDates, which is directly data dependent on the non-idempotent service calls holdHotel and holdFlight (since these services modify reservationData, the input parameter of the checkDates service).

So, only five of the 10 non-idempotent service calls on trace t_1 are identified as relevant. The <flow> and <pick> activities on trace t_1 identify another three relevant change states, so RUMOR now generates a total of 0 ($k = 5$), 2 ($k = 10$), 5 ($k = 15$) and 8 ($k = 20, 25, 30$) plans for this trace. The desired plans p^1_A and p^1_B are still generated (at $k = 20, 25, 30$), but are now ranked 6th and 7th (instead of 11th and 12th). These two plans are still ranked low because of the amount of compensation they require, but by omitting plans that cannot alter the control flow of the current execution, we reduced the number of plans presented to the user by 50%.

We carried out the same analysis on trace t_2: six of the original 10 change states are marked as relevant. Since trace t_2 visits the same <pick> and <flow> activities as t_1, four of the relevant change states are those identified by these activities, the remaining two relevant change states correspond to the non-idempotent service calls associated to predicates 5 and 6, also summarized in Table 2.

4.2 Avoiding Forbidden Behaviors

Our second method aims to remove those plans that result in the system performing behavior which is explicitly forbidden. That is, we use safety properties to help filter recovery plans for liveness properties. This process is outlined in Figure 4: given a failing trace T, we compute a plan P which first "undoes" the trace until a change state and then computes an alternative path to a certain goal (shown using dashed lines). P is *unsuitable* if the path from the initial state going through this change state and continuing via the computed alternative path towards the goal (shown using a thick line and denoted $\mathtt{T_P}$) is forbidden. That is, there exists a safety monitor $\mathtt{A}_i$ which enters an error state when executed on $\mathtt{T_P}$.

The simplest method, presented here, applies the filtering w.r.t. safety properties *after* the set of recovery plans has already been produced. That is, given a trace T and a plan P, we can compute $\mathtt{T_P}$ and simulate every safety monitor on it, removing P from consideration if any monitor fails.

The path from the initial state to the change state used in P can be very long, and thus we feel that simulating each monitor on the entire trace $\mathtt{T_P}$ is very inefficient. We also cannot execute monitors backwards from the error state of T along the "undo" part of P: while our monitors are deterministic, their inverse transition relations do not have to be deterministic, making the execution in reverse problematic.

Instead, we aim to maintain enough data during the execution of the trace T in order to be able to restart monitors directly from the change state, moving along the new, recomputed path of the plan. To do so, as T executes, we record the states of all monitors in the system in addition to the states and transitions of the application. Thus, for each state s of the application reached during the execution of trace T, we store a tuple $(s, s_{\mathtt{A_1}}, ..., s_{\mathtt{A_n}})$, where $s_{\mathtt{A_i}}$ is a state of the monitor $\mathtt{A_i}$ as the application is in state s. To check whether P is a valid plan, we go directly to the change state s_c in P, extract the tuple $(s_c, s_{\mathtt{A_1}}, s_{\mathtt{A_2}}, ..., s_{\mathtt{A_n}})$ stored as part of T and then simulate each safety monitor $\mathtt{A_i}$ starting it from the state $s_{\mathtt{A_i}}$ along P which starts at state s_c.

As an example, consider the TBS system and trace t_2, described in Sec. 3.5, violating the property P_2. Our approach produces over 60 plans to recover from this violation, for plan lengths $k \geq 25$ (see Table 1). Consider the plan that goes back all the way until encountering the change state associated with the call to getAvailableFlight, canceling the booked flights on the way. Afterwards, this plan attempts to re-book a flight, but fails to do so. It continues executing, and tries to pick a car at the airport instead. However, this plan violates property P_3 (i.e., monitor $\mathtt{A_3}$ would enter its error state upon seeing an action pa). Thus, we automatically filter this plan out and do not present it to the user.

Overall, applying this approach to recovery for trace t_2 reduces the number of plans from over 60 to 41. Furthermore, combining it with the computation of the relevant change states, the number of plans is further reduced to 23 (see Table 3). While this number is still relatively large, it presents a considerable improvement and enables the user to pick a desired plan more easily.

5 Summary and Related Results

In this paper, we briefly summarized the RuMoR approach to runtime monitoring and recovery of web services w.r.t. behavioral properties expressed as desired or forbidden behaviors. We have also described two optimizations to the recovery plan generation: reducing the number of change states and using monitors to filter those plans which represent forbidden behaviors.

Hallé and Villemaire in [8, 9], suggest a monitoring framework where data-aware properties are written in LTL enriched with first-order quantifications. Generating automata for runtime monitoring w.r.t. such an expressive language is significantly more complex than in our framework. Recovery in the work of Hallé and Villemaire is based on executing a predefined function, associated with an individual property – i.e., all failures of the same property are treated in the same way, statically. In contrast, our method is dynamic and generates recovery plans customized for each error.

An emerging research area in recent years is that of *self-adaptive* and *self-managed* systems (see [1, 12, 3] for a partial list). A system is considered self-adaptive if it is capable of adjusting itself in response to a changing environment. This approach is different from ours, since our framework does not change the system itself, and recovery plans are discovered and executed using the original application.

The work of Carzaniga et al. [2] 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.

Our work in this space is on-going. Specifically, we are interested in further case studies, optimized usage of SAT solving for better plan generation (e.g., so that we encode forbidden behaviors as part of the SAT problem rather than filtering them out after the plan has been generated), ways to harvest and effectively express behavioral properties, since this is key to the usability of our approach.

References

[1] Y. Brun and N. Medvidovic. Fault and Adversary Tolerance as an Emergent Property of Distributed Systems' Software Architectures. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems, (EFTS'07*, pages 1–7, September 2007.

[2] A. Carzaniga, A. Gorla, and M. Pezze. Healing Web Applications through Automatic Workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, 2008.

[3] B. H. C. Cheng, R. de Lemos, D. Garlan, H. Giese, M. Litoiu, J. Magee, H. A. Müller, and R. Taylor. Seams 2009: Software engineering for adaptive and self-managing systems. In *31st International Conference on Software Engineering, (ICSE'09), Companion Volume*, pages 463–464, May 2009.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, March 1998.

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

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a Tool for Model-Based Verification of Web Service Compositions and Choreography. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 771–774, May 2006.

[7] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proceedings of the 13th international conference on World Wide Web (WWW'04)*, pages 621–630, May 2004.

[8] S. Hallé and R. Villemaire. Runtime Monitoring of Message-Based Workflows with Data. In *Proceedings of the 12th IEEE Enterprise Distributed Object Computing Conference (ECOC'08)*, pages 63–72, 2008.

[9] S. Hallé and R. Villemaire. Browser-Based Enforcement of Interface Contracts in Web Applications with BeepBeep. In *Proceedings of Computer Aided Verification (CAV'09)*, pages 648–653, 2009.

[10] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proceedings of the 3rd International Conference on Business Process Management (BPM'05)*, volume 3649 of *LNCS*, pages 220–235, 2005.

[11] H. A. Kautz and B. Selman. Unifying SAT-based and Graph-based Planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 318–325, 1999.

[12] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. In *The ICSE'07 Workshop on the Future of Software Engineering (FOSE'07)*, pages 259–268, May 2007.

[13] K. M. Olender and L. J. Osterweil. "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation". *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[14] 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. *Science of Comp. Prog.*, 67(2-3):162–198, 2007.

[15] J. Simmonds, S. Ben-David, and M. Chechik. Guided Recovery for Web Service Applications. In *Proceedings of Int. Conf. on Foundations of Software Engineering (FSE'10)*, pages 1–10, 2010. To appear.

[16] J. Simmonds, S. Ben-David, and M. Chechik. Monitoring and Recovery of Web Service Applications. In *Smart Internet*, Lecture Notes in Computer Science, pages 1–35. Springer, 2010. To appear.

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

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