RuMoR: Monitoring and Recovery of BPEL 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 distributed processes that are composed of dynamically bounded services. Since the overall system may only be available at runtime, static analysis is difficult to perform in this setting. Instead, these systems are many times checked dynamically, by monitoring their behavior during runtime. Our tool performs monitoring of web service applications, and, when violations are discovered, we automatically propose and rank recovery plans which users can then select for execution. Properties, specified using property patterns, are transformed into finite-state automata. Finite execution traces of web services described in BPEL are checked for conformance at runtime. For some property violations, recovery plans essentially involve "going back" – compensating the executed actions until an alternative behaviour of the application is possible. For other violations, recovery plans include both "going back" and "re-planning" – guiding the application towards a desired behaviour. These plans are generated using techniques adapted from AI planning. **MC: Do not like the last sentence. Maybe: just remove "AI"?**

Keywords

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

THIS IS AN OLD VERSION – paper2.tex is the correct version

1. INTRODUCTION

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*. These components – which can be written in a traditional compiled language such as Java, or in an XML-centric language such as BPEL [5] – communicate through published interfaces. Since an application is a composition of several distributed business processes, its correctness depends on the correctness of its partners and their interactions.

Web service applications 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 infrustructures 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, but the automatic application of compensations does not allow the user to choose between them.

This paper describes RuMoR, a *user-guided* recovery tool for web services. This tool takes as input the target BPEL application, a set of safety and liveness properties of the application (specified using the Specification Pattern System [1]), and the maximum length and number of recovery plans. Our approach has three phases: *Preprocessing*, *Monitoring* and *Recovery*. Properties are translated into monitors during the *Preprocessing* phase. During the *Monitoring* phase, RuMoR runs these monitors in parallel with the application, stopping when one of the monitors is about to be violated. This triggers the *Recovery* phase, where various techniques are used to generate recovery plans.

For violations of safety properties, recovery 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.

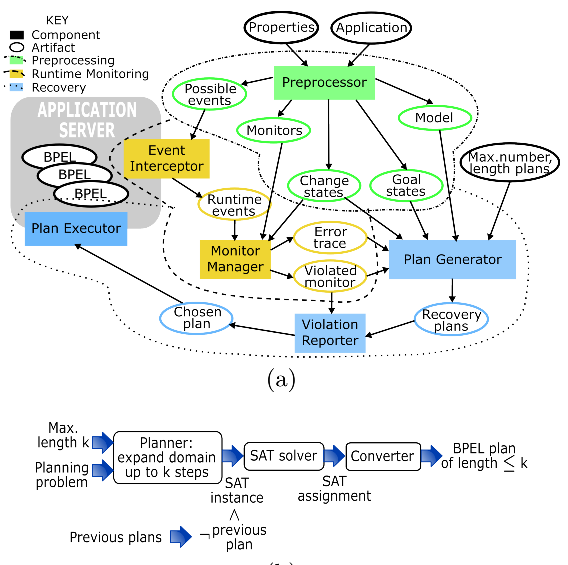


Figure 1: (a) Tool architecture; (b) Recovery plan generation for liveness properties.

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.

We generate such plans by adapting techniques from the field of AI planning. The application itself defines a planning domain, i.e., which events can be executed and when. A recovery plan is a sequence of events that, starting at the current error state, leads to an application state from which the current liveness property can be satisfied. We call these states "goal states", and compute them through static analysis during the *Preprocessing* phase. When there are multiple recovery plans available, RuMoR automatically ranks 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.

2. DESIGN AND IMPLEMENTATION

We have implemented RuMoR using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts. The *Preprocessing* and *Monitoring* phases of our tool are the same for both safety and liveness properties, but different components are required for generating plans from the two types of properties. We show the architecture of our tool in Fig. 1a (rectangles and ovals are components and artifacts, respectively).

Developers create properties for their web services using property patterns and system events. During the preprocessing phase, the *Property Translator* (PT) component receives the specified properties and turns them into monitors. The *LTS Extractor* (LE) component extracts an LTS model of the BPEL program (using the WS-Engineer extension for LTSA [3]) and creates a second LTS model with compensation. The *LTS Analyzer* (LA) computes goal links and change states using the techniques described in [6].

During the execution of the application, the *Event Interceptor* (EI) component intercepts application events and sends them to the *Monitor Manager* (MM) for analysis. MM updates the state of each active monitor, until an error has been found (which activates the recovery phase) or all partners terminate. MM also stores the intercepted events.

During the recovery phase, artifacts from both the preprocessing and the runtime monitoring phases are used to generate recovery plans. The *Safety Plan Generator* generates recovery plans that can only compensate executed activities. For liveness properties, plans can compensate executed activities and execute new activities. In this case, the *Liveness Plan Generator* first generates the corresponding planning problem using the model of the application, error trace and violated monitor. The planning problem is expressed in STRIPS [2] – an input language to the planner Blackbox [4], which we use to convert it into a SAT instance. The maximum plan length is used to limit the size of the planning graph generated by Blackbox, effectively limiting the size of the plans that can be produced. As shown in Fig. 1b, we modify the initial SAT instance in order to produce alternative plans. 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. For our method of generating multiple plans, where each SAT instance is more restricted than the previous one, this is particularly useful, leading to efficient analysis.

All computed plans are presented to the application user through the *Violation Reporter* (VR), and the chosen plan is executed by the *Plan Executor* (PE). VR generates a web page snippet with violation information, as well as a form for selecting a recovery plan. Developers must include this snippet in the default error page, so that the computed recovery plans can be shown when an error is detected. If no monitor is violated during the execution of the chosen plan (MM updates the states of the active monitors during the plan execution), the framework switches back to runtime monitoring.

3. CONCLUSION

We presented RuMoR, a recovery tool based on monitoring and planning to detect and fix runtime errors detected in BPEL applications. Recovery plans are generated by doing a combined static and dynamic analysis of the target application and user-specified properties of the application. Our experience [6] shows that we can effectively generate user-expected plans, and experiments so far suggest scalability with respect to runtime complexity.

4. REFERENCES

[1] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE'99*, pages 411–420, May 1999.
[2] R. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.*, 2(3/4):189–208, 1971.
[3] 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.
[4] H. A. Kautz and B. Selman. Unifying SAT-based and Graph-based Planning. In *IJCAI'99*, pages 318–325, 1999.
[5] OASIS. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, Accessed January 2009.

[6] J. Simmonds, S. Ben-David, and M. Chechik. Guided Recovery for Web Service Applications. In *Proc. of FSE'10*, 2010. To appear.

APPENDIX

A. DEMO

Demonstration plan:

1. Explain example application
2. *Preprocessing:*
   (a) Convert properties into monitors
   (b) Generate the application model
3. *Monitoring and Recovery:*
   (a) Execute scenario S_1 (S_2), which violates P_1 (P_2)
   (b) Generate planning domain
   (c) Generate multiple recovery plans
   (d) Execute a plan chosen by the audience

A.1 Example - the Trip Advisor System (TAS)

Fig. 2a shows the BPEL-expressed workflow of the Trip Advisor System. In a typical scenario of this system, 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 system is to make sure the customer has the transportation needed to get to her destination (this is a desired behavior which we refer to as P_1) 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 which we refer to as P_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). 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> (scope enclosed in bold, blue lines —, 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).

A.2 Preprocessing

Fig. 2b shows the TAS LTS model (with compensation). To increase legibility, each transition represents an action and its compensation, labeled in the form $a/\bar{a}$, where a is the application activity and $\bar{a}$ is its compensation. Monitor $\mathtt{A}_1$ in Fig. 3a represents P_1: if the application terminates before rd appears, the monitor moves to the (error) state 3. State 1 is a good state since the monitor enters it once the booked transportations reach the destination (rd). Monitor $\mathtt{A}_2$ in Fig. 3b represents P_2. It enters its error state (4) when either a limo was booked and later an expensive flight, or an expensive flight was booked first and then a limo.

A.3 Scenario S_1

Consider the execution of TAS in which the customer chooses the air/ground option (carAndFlight), and then tries to book the flight before the car. In this example, there is a communication problem with the flight system partner, and the invocation of the cf service time outs. This scenario corresponds to the trace t_1, depicted by dotted transitions in Fig. 2b.

The application server detects that the cf invocation timed out, and sends a TER event (not shown in Fig. 2b) to the application. Our framework intercepts this TER event and determines that executing it turns t_1 into a failing trace, because the monitor $\mathtt{A}_1$ would enter its error (red) state 3. In response, our framework does not deliver the TER event to the application, and instead initiates recovery.

If we set the maximum plan length to be 10, the *Liveness Plan Generator* computes the plans shown in Fig. 4a. 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 then 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.

Fig. 5a shows the snippet generated by the *Violation Reporter*. Fig. 5b shows the (simplified) source code of such an error reporting page, where the bolded line has the instruction to include the snippet, and Fig 5c shows a screen shot of `error.jsp` after recovery plans for P_1 have been computed. We will pick one of these plans and execute it.

A.4 Scenario S_2

In another scenario, the customer attempts to arrive at her destination via a limo (bl) and an expensive flight (expF). This corresponds to the trace t_2, depicted by dashed transitions in Fig. 2b. As the monitor A_2 has a transition on expF to an error state, our framework delays the execution of this event from application state 21. In this example, executing expF will make A_2 enter its error state 4, so t_2 is also a failing trace. The expF event is not delivered, and the recovery phase is activated.

The *Safety Plan Generator* returns the recovery plans shown in Fig. 4b. We add state names between transitions for clarity and refer to plans r_s 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 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 A_2.

B. TOOL MATURITY AND AVAILABILITY

The following components of our tool are still under development: *Property Translator*, *Event Interceptor*, and *Plan*

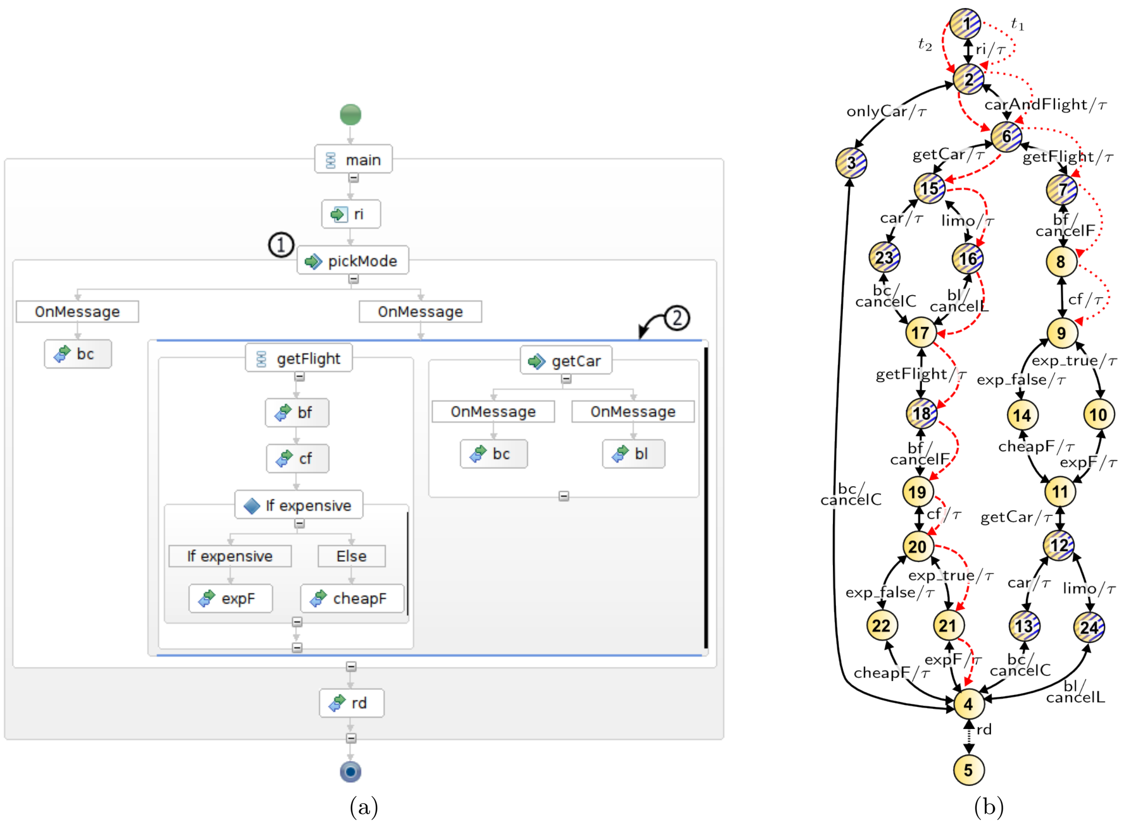


Figure 2: (a) Workflow of TAS; (b) corresponding LTS model.

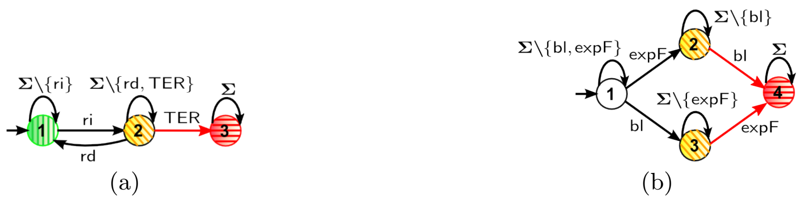


Figure 3: Monitors: (a) $\mathtt{A}_1$, (b) $\mathtt{A}_2$. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

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

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

Figure 4: Recovery plans for TAS: (a) plans of length ≤ 10 for Scenario S_1; (b) plans for Scenario S_2.

Executor. The tool is not available online, as it includes IBM intellectual property.

snippet.jsp

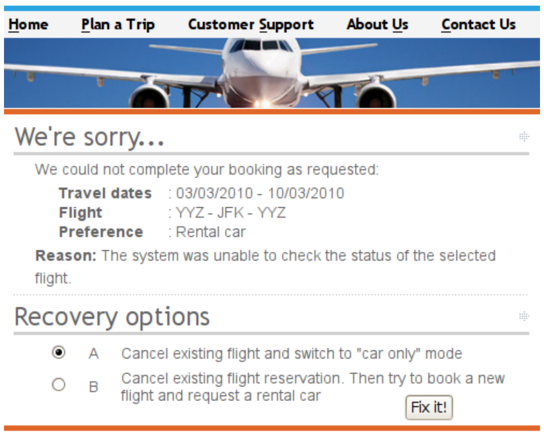
```
<p><strong>Reason:</strong> The system was unable to check
            the status of the selected flight.</p>

<h1 class="title">Recovery options</h1>
<br>
<html:form name="recovery_option" action="execute_plan.do" ...>
<table width="100%">
 <tr>
  <td width="10%"><html:radio property="plans" value="p0" /></td>
  <td width="10%" align="center">A</td>
  <td>Cancel existing flight and switch to "car only" mode</td>
 </tr>
 <tr>
  <td><html:radio property="plans" value="p1" /></td>
  <td align="center">B</td>
  <td>Cancel existing flight reservation. Then try to book a new
      flight and request a rental car</td>
 </tr>
 <tr><td></td><td></td>
  <td align="center"><html:submit value="Fix it!"/></td>
 </tr>
</table>
</html:form>
```

(a)

error.jsp

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html>
<head>...</head>
<body>
 <h1 class="title">We're sorry...</h1>
 <div class="content">
  <p>We could not complete your booking as requested:</p>
  <br>
  <table width="80%">
   <tr><th>Travel dates:</th><td>...</td></tr>
   <tr><th>Flight:      </th><td>...</td></tr>
   <tr><th>Preference:  </th><td>...</td></tr>
  </table>
  <br>
 </div>
 <%@ include file = "snippet.jsp" %>
</body>
```

(b)



We're sorry...

We could not complete your booking as requested:

Travel dates	: 03/03/2010 - 10/03/2010
Flight	: YYZ - JFK - YYZ
Preference	: Rental car

Reason: The system was unable to check the status of the selected flight.

Recovery options

(●) A Cancel existing flight and switch to "car only" mode

( ) B Cancel existing flight reservation. Then try to book a new flight and request a rental car

(c)

Figure 5: Violation reporting: (a) `snippet.jsp`, automatically generated snippet that contains recovery plans; (b) `error.jsp`, the application error handling page; (c) `error.jsp` displayed on a browser.