RuMoR: Monitoring and Recovery for BPEL Applications

Jocelyn Simmonds, Marsha Chechik
Department of Computer Science
University of Toronto, Toronto, Canada
{jsimmond, chechik}@cs.toronto.edu

ABSTRACT

We describe a **RU**ntime **MO**nitoring and **R**ecovery framework (RuMoR) for BPEL applications. Our tool checks for behavioral conformance with respect to a set of user-specified properties. When runtime violations are discovered, RuMoR automatically proposes and ranks recovery plans which users can then select for execution. These plans are generated using an adaptation of a SAT-based planning technique.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*

General Terms

Design, Verification

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. Services can be dynamically discovered, and partners can modify services while they are being used. 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), they are of limited use since it is hard to determine the state of the application after executing a set of compensations.

In [5], we proposed a framework for runtime monitoring and recovery that uses user-specified properties to automatically compute recovery plans. In this paper, we present RuMoR, an *implementation* of this technique built on top of IBM WebSphere. RuMoR takes as input the target BPEL application, enriched with the compensation mechanism allowing us to undo some of the actions of the program, and a set of properties (specified as desired/forbidden behaviours) that need to be maintained by the application as it runs. 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 system is to make sure the customer has the transportation needed to get to her destination (desired behavior) while keeping the costs down, i.e., she is not allowed by her company to reserve an expensive flight and a limo (forbidden behavior).

When one of the property violations is detected at runtime, RuMoR outputs a set of ranked recovery plans and enables applying the chosen plan to continue the execution. If the application exhibits a forbidden behaviour, RuMoR 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) [5]. For example, if TAS produces an itinerary that is too expensive, a potential recovery plan might be to undo the limo reservation (so that a car can now be booked) or to undo the flight reservation and see if a cheaper one can be found.

The system fails to produce a desired behaviour when calls to some partners terminate, leaving it in an unstable state. In such cases, RuMoR computes plans that redirect the application towards executing new activities, those that lead to completion of the desired behaviour. 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 trying to reserve the rental car from home instead. In what follows, we give a brief description of the architecture of the tool (Sec. 2) and some experience applying it on several examples (Sec. 3).

2. IMPLEMENTATION

We have implemented 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 top of Fig. 1. In the Preprocessing phase, the properties are turned into finite-state automata (monitors). We use the WS-Engineer extension for LTSA [2] to translate the BPEL application into a labelled transition system (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

ASE'10, September 20–24, 2010, Antwerp, Belgium.
ACM 978-1-4503-0116-9/10/09.

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

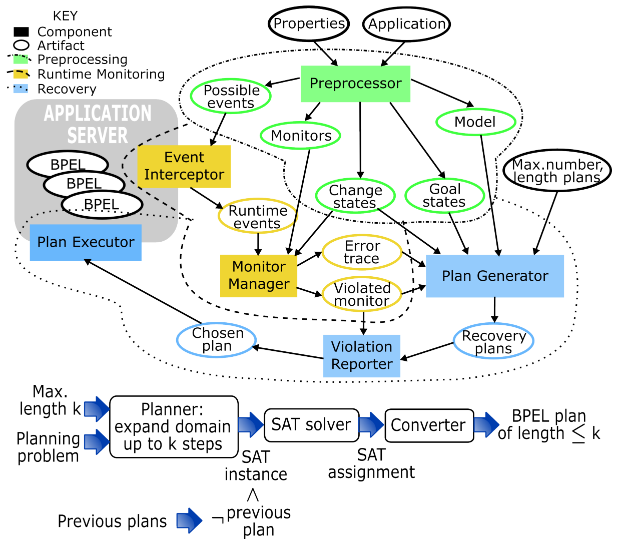

Figure 1: Architecture of the tool (top) and recovery plan generation for violating a desired behavior (bottom).

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, and enables reasoning about partners expressed in different languages.

During the recovery phase, the Plan Generator component generates recovery plans (see [5] for a description of the various techniques used). For example, in the case of desired behavior violations, RuMoR tries to solve the following planning problem: "From the current state in the system, find a plan to achieve the goal that goes 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 (with compensation). The initial and goal states of the planning problem are the current error state and the precomputed goal states, respectively. The process for computing plans is shown in the bottom of Fig. 1. RuMoR uses Blackbox [4], 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, leading to efficient analysis (see Sec. 3).

All computed plans are presented to the application user through the Violation Reporter component. 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 on 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 [6]. RuMoR takes advantage of their implementation as part of IBM WebSphere.

App.	k	vars	clauses	# plans	time (s)
TAS	6	135	254	1	0.01
25 states	8	798	10,355	5	0.13
31 trans.	13	1,398	25,023	13	0.27
TBS	5	108	464	0	0.01
52 states	10	883	30,524	2	0.14
67 trans.	15	1,456	74,932	8	1.37
	20	2,141	135,047	18	4.72
	25	3,298	246,210	60	29.16
	30	5,288	464,654	68	61.34
FV	15	797	16,198	2	0.04
28 states	22	1,436	33,954	4	0.74
37 trans.	26	1,804	44,262	8	1.14
	42	3,276	85,494	40	3.12
FC	4	42	159	1	0.01
18 states	6	95	592	2	0.02
22 trans.	12	321	3,248	4	0.15
	16	554	7,393	5	0.27
	20	856	14,427	13	1.38

Table 1: Plan generation data for desired behaviors.

3. EVALUATION AND SUMMARY

We have tested RuMoR on four BPEL applications: TAS, Flickr visibility and comments (FV and FC, respectively, both adapted from [1]), and the Travel Booking System (TBS, adapted from [3]). For each application, we specified a set of simple properties and a test case that would violate a desired behavior. RuMoR was used to generate a set of (ranked) recovery plans for each test case and maximum plan length (k) (see Table 1). "vars" and "clauses" indicate the size of the SAT instance generated; "# plans" is the total number of plans of up to length k; and "time (s)" indicates the total time (in seconds) needed to generate these plans. For all four systems, the number of variables and the number of clauses grows linearly with the length of the plan, and the running time of the SAT solver remained in seconds (see [5] for details).

To summarize, RuMoR is a recovery tool based on monitoring and planning to detect and fix runtime violations of behavioural properties of BPEL applications, using its compensation mechanism. Our experience has shown that this approach computes a small number of highly relevant plans, doing so quickly and effectively.

4. REFERENCES

[1] A. Carzaniga, A. Gorla, and M. Pezze. Healing Web Applications through Automatic Workarounds. *STTT*, 10(6):493–502, 2008.
[2] 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.
[3] Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse. Runtime Monitoring of Web Service Conversations. In *Proc. of CASCON'07*, pages 42–57, 2007.
[4] H. A. Kautz and B. Selman. Unifying SAT-based and Graph-based Planning. In *IJCAI'99*, pages 318–325, 1999.
[5] J. Simmonds, S. Ben-David, and M. Chechik. Guided Recovery for Web Service Applications. In *Proc. of FSE'10*, 2010. To appear.
[6] 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.