Journal of Computing manuscript No.
(will be inserted by the editor)

Monitoring and Recovery for Web Service Applications

Jocelyn Simmonds · Shoham Ben-David · Marsha Chechik

Received: date / Accepted: date

Abstract Web service applications are distributed processes that are composed of dynamically bounded services. In this paper, we give a definitive description of a framework for performing runtime monitoring of web service applications against behavioural correctness properties described as finite state automata. These properties specify forbidden and desired interactions between service partners. Finite execution traces of web service applications described in BPEL are checked for conformance at runtime. When violations are discovered, our framework automatically proposes adaptation strategies, in the form of plans, which users can select for execution. Our framework also allows verification of stated pre- and post-conditions of service partners and provides guarantees of correctness of the generated recovery plans.

Keywords Web services · LTS · behavioural properties · pre- and post-conditions · runtime monitoring · planning · SAT solving

J. Simmonds
Departamento de Informática
Universidad Técnica Federico Santa María
Santiago, Chile
E-mail: jsimmond@inf.utfsm.cl

S. Ben-David
School of Computer Science
Hebrew University
Jerusalem, Israel
E-mail: shohambd@gmail.com

M. Chechik
Department of Computer Science
University of Toronto
Toronto, Canada
E-mail: chechik@cs.toronto.edu

1 Introduction

Web service applications are distributed processes, called *orchestrations*, composed of dynamically bounded services. Development and maintenance of quality web service applications presents a major challenge in practice, mostly due to this dynamic binding, and a variety of failures that can happen when potentially third party partners communicate over the web. Some of the failures are due to the faulty logic, to invalid input data, to incorrect service invocation, to hardware problems, and to network failures. Most of these problems are not detectable statically, and thus a web service needs to dynamically recover from errors as they are discovered at runtime.

An application is considered to be *self-healing* [15] if it can detect failures and diagnose faults, and can adjust itself in response. Error recovery frameworks omit the diagnosis phase and can thus be classified as *simple* self-healing systems.

Since runtime errors are inevitable (and potentially exposed to millions of users before they are found/fixed), frameworks for running these types of applications 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. These error recovery mechanisms can be used to minimize the impact of runtime bugs, but the developer must anticipate possible runtime errors since these mechanisms are statically defined. Moreover, it is hard to determine the state of the application after executing a set of compensations.

Several works [5, 22, 12, 13, 7] suggest "self-healing" mechanisms for web-service applications. These approaches vary in applicability and effectiveness; yet they do not provide *guarantees* of correctness of the resulting recovery strategies. A possible such guarantee can be as follows: if the provided service pre- and post-conditions are valid at runtime, then executing a recovery plan leaves the application in a non-error state from where regular execution can continue.

In this paper, we report on a runtime monitoring and recovery framework for orchestrations expressed in BPEL. Users specify desired and prohibited interactions between partners. We also take advantage of service contracts (pre- and post-conditions), if they are available, and use compensation – the "standard" error recovery mechanism built into BPEL. The system then executes the chosen plan(s) as means of adaptation. To the best of our knowledge, we are the first to suggest using this method for generating "provably correct" recovery plans.

1.1 Motivating 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

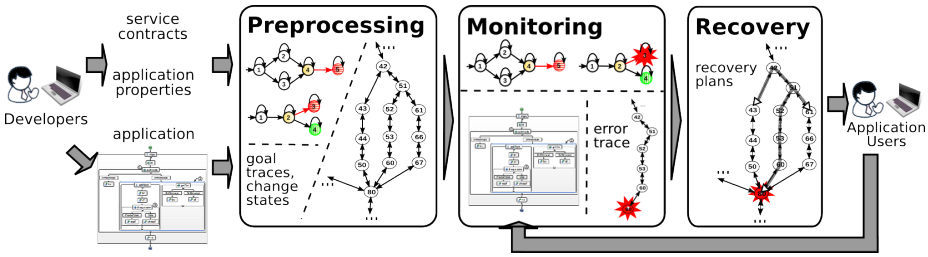


Fig. 1 Overview of our approach.

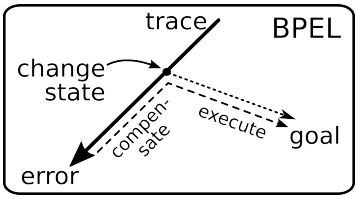


Fig. 2 A schematic view on plan generation.

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 `TAS` system interacts with two partners in order to achieve its business goals – the Car system (which offers two web services: one to reserve cars and another to reserve limos) and the Flight system (which offers two web services: one to reserve flights and another to check whether the flights are cheap or expensive). Since the `TAS` system is a composition of several distributed services, its correctness depends on the correctness of its partners and their interactions.

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).

If the system produces an itinerary that is too expensive, we initiate a recovery process. In this case, it will consist of either canceling the limo reservation (so that a car can now be booked) or canceling the flight reservation to see if a cheaper one can be found. In general, recovery from observing an undesired behavior or violating service contracts entails using 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.

Yet just merely going back is insufficient to ensure that the system can produce a desired behaviour. For example, the Flight system can go down while the user attempts to book air transportation, thus preventing the entire system from getting the user to her destination since the air/ground combination is no longer available. To adapt from this fault, the system may suggest

that the user rent a car for the whole trip instead. More precisely, the recovery plan to the user's destination (her "goal" state) includes either calling the flight reservation again or canceling the reserved ground transportation from the airport, if any, and trying to reserve the rental car from home instead. In order to achieve such behaviours, we aim to compute plans that redirect the application towards executing new activities, those that lead to goal satisfaction.

1.2 Overview of the Approach

In our approach, developers supply a BPEL program, a set of service contracts (pre- and post-conditions for partner invocations) and a set of correctness properties (in the form of required and prohibited interactions between partners) that need to be maintained by the program as it runs. The BPEL program is enriched, by its developers, with the compensation mechanism, which allows us to undo some of the actions of the program. Correctness properties are turned into monitors using techniques described in [38]. In this paper, we focus on runtime monitoring and generation of recovery plans should a violation be detected. Availability of service contracts allows us to provide guarantees of correctness of the recovery plans. Fig. 1 shows a schematic view of our approach to runtime monitoring and error recovery.

In the Preprocessing phase, a formal model is extracted from the given BPEL program and enriched with the compensation information. The Runtime Monitoring phase runs the monitors in parallel with the BPEL application, stopping when one of the monitors is about to enter its error state, or when service contracts are about to be violated. The use of high-level properties and/or service pre- and post-conditions allows us to detect the violation, and our event interception mechanism allows us to stop the application *right before* the violation occurs and begin the Recovery phase.

In the Recovery phase, we identify a set of possible plans that recover 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. The overall recovery planning problem is 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. 2.

When there are multiple recovery plans available, we automatically sort 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.

1.3 Contributions

Parts of this project have been published elsewhere [43, 39, 42, 40, 41]. The novel contribution of this paper is our ability to give guarantees of correctness of the generated recovery plans. To do so, we extend our previously published approach in two ways: (1) we specify service pre- and post-conditions and collect information about select predicates in states of our formal model and (2) we define a concept of *adequate compensation* – an ability to correctly compensate an action by returning to the state where this action can be executed again. The collected information allows us to check correctness of service contracts and adequacy of compensation. We can then prove that recovery from an undesired behavior leaves the application in a previously observed state from which an alternative behavior that avoids the violation is possible. We can also prove that recovery from being unable to produce a desired behavior correctly returns the application to a previously observed state from which the generated plan implements the desired behavior.

This paper presents a definitive description of the input, analysis and recovery formalism. It also provides experiments measuring the cost and effectiveness of collecting information about program states and checking it for adequacy of compensation and violations of service contracts. In addition, we suggest and implement a number of heuristics for improving the quality of plan computation and reducing the number of unusable plans presented to the user.

The rest of this paper is organized as follows: In Section 2, we present some required notation. In Section 3, we describe inputs to our system: BPEL models (including compensation mechanisms and service contracts) and correctness properties expressed as monitors. In Section 4, we define the representation of BPEL models as Doubly Labeled Transition Systems (L^2TS), and show how to use this representation to check that compensation is adequately defined and to identify change states and goal traces. We discuss runtime monitoring in Section 5 and describe recovery from violations of behavioral properties in Sections 6 and 7. Both of these sections also include proofs of correctness of the generated recovery plans. We report on our implementation in Section 8. We have tested our framework on various case studies, with full results available in [38]. Instead, Section 9 focusses on assessing costs and benefits of doing the adequate compensation check, using two web service examples. We also measure the impact of the optimizations we have implemented to improve the quality of generated plans. We compare our work with related approaches in Section 10. Finally, in Section 11, we summarize the article and give suggestions for future work.

2 Preliminaries

In this work, we formalize BPEL using Doubly Labelled Transition Systems (L^2TS) which we define below.

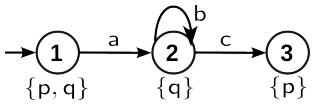


Fig. 3 Example of an L^2TS.

Definition 2.1 (LTS [32]) *A* Labelled Transition System *LTS is a quadruple* (S, Σ, δ, I)*, where* S *is a set of states,* Σ *is a set of actions,* $\delta \subseteq S \times \Sigma \times S$ *is a transition relation, and* $I \in S$ *is the initial state. 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 M 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 $s_i \xrightarrow{a_i} s_{i+1}$.

Definition 2.2 (L^2TS [34]) *A* Doubly Labelled Transition System L^2 *TS is a quintuple* $\mathcal{D} = (S, \Sigma, \delta, I, \mathscr{L})$*, where* (S, Σ, δ, I) *is an LTS and* $\mathscr{L} : S \to 2^{\mathbf{AP}}$ *is a propositional labelling function that associates a set of atomic propositions to each state. We denote the substructure* (S, Σ, δ, I) *of* $\mathcal{D}$*, i.e., the underlying LTS associated to an* L^2 *TS, by LTS(*$\mathcal{D}$*).*

An example L^2TS is shown in Fig. 3, where $S = \{1, 2, 3\}$, $\Sigma = \{a, b, c\}$, $\delta = \{(1, a, 2), (2, b, 2), (2, c, 3)\}$, $I = 1$, and $\mathscr{L} = \{(1, \{p, q\}), (2, \{q\}), (3, \{p\})\}$. The notion of a trace in an L^2TS is defined analogously to the notion of a trace in an LTS.

3 Input

Inputs to our system are a BPEL program enriched with compensation actions and a set of behavioral correctness criteria specified as simple safety and liveness properties. We describe these below. More information about the core BPEL activities and variable definition is given in the appendix.

3.1 BPEL Programs

BPEL [35] is a standard for implementing orchestrations of web services (provided by partners) by specifying an executable workflow using predefined activities, over a set of global variables. Service interfaces, i.e., operations and their input/output messages and ports, are described using WSDL (Web Service Description Language) [46]. These services are then made available to the BPEL application by defining partner bindings within the BPEL specification.

3.1.1 Process definition

Orchestrations are created by composing different types of activities, such as service invocations (<invoke>) and variable assignments (<assign>), using standard control structures. Activities can be logically grouped using

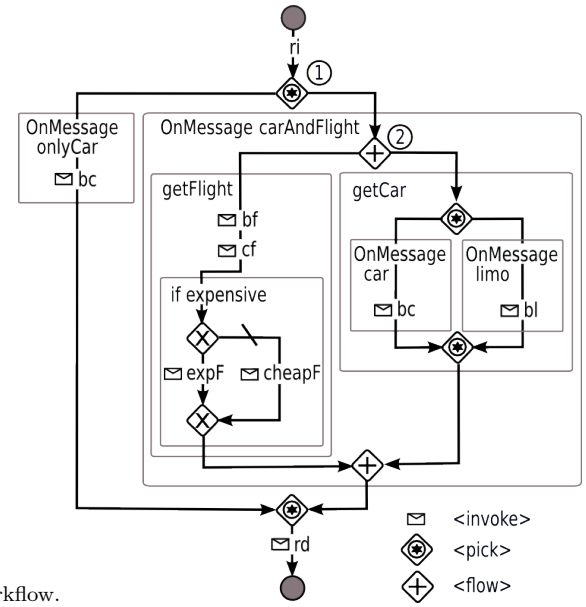


Fig. 4 `TAS` workflow.

<scope>s. The <pick> activity is used to wait for one of several possible messages (specified using <onMessage>) to arrive. An <empty> activity does nothing when executed.

Fig. 4 shows the BPEL-expressed workflow of the Trip Advisor System (`TAS`), introduced in Section 1. We use the NetBeans SOA notation [36]. `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 used to decide which local service to invoke, broadcasting whether the flight is cheap (cheapF) or expensive (expF). Service interactions are preceded by a ✉ symbol.

The workflow begins by <receive>'ing and validating 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. The getFlight 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).

BPEL global variables are those accessible throughout the BPEL process; local variables are only available to the activities within the <scope> in which they are defined. Allowed variable types are defined using an XSD schema; these types are then used to define service input and output messages, as well as global BPEL variables. Fig. 5a shows the definition of the input and output

```
<message name="in_bc" xmlns:x="tas.xsd"/>
   <part name="city" element="x:destCity"/>
   <part name="fromDate" element="x:fromDate"/>
   <part name="toDate" element="x:toDate"/>
</message>

<message name="out_bc" xmlns:x="tas.xsd"/>
   <part name="bookingNo" element=
      "x:carBookingNo"/>
   <part name="status" element="x:carStatus"/>
   <part name="type" element="x:carType"/>
</message>
```

(a)

```
<variables>
   <variable name="tripData" element=
      "x:tTripData"/>
   <variable name="inputBookCar" messageType=
      "x:in_bc"/>
   <variable name="outputBookCar" messageType=
      "x:out_bc"/>
      ...
</variables>
```

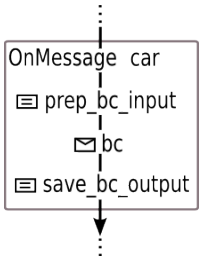
(b)

Fig. 5 (a) Part of the WSDL file `tas.wsdl`, showing the definition of the WSDL message types for the `bc` service; and (b) partial declaration of `TAS`'s global variables.

messages of the `bc` service. For example, `in_bc`, `bc`'s input message, consists of three parts: `city`, `fromDate` and `toDate`, where `city` is a string and both date fields are integers. Fig. 5b shows the definition of three global variables of the `TAS` system: `tripData`, `inputBookCar` and `outputBookCar`. The first variable is used to maintain the state of the application, while the second two are the input and output variables of the `bc` service. The details of how these variables are defined can be found in the appendix.

Before a service can be invoked, its input message must be initialized by copying the pertinent variable values from `tripData`. This is done via an <assign>ment activity placed right before the service invocation (shown in Fig. 6a using a ▤ symbol). Assignment activities consist of multiple <copy> rules, each with their own <from> and <to> parts, representing the source and destination data, respectively. For example, Fig. 6b shows the BPEL definition of the `prep_bc_input` assignment activity, which has three simple <copy> rules, one for each message part of `in_bc` defined in Fig. 5a.

In a similar fashion, when a service returns a message to the application, the application state must be updated to reflect the outcome of the service invocation. This is done by placing an <assign> activity right after the service invocation. The post-service invocation <assign> activity for `bc` is shown in Fig. 6c. Here, $`tripData.carBookingNo`, $`tripData.carStatus` and $`tripData.carType` are updated with values extracted from the service's output message. In the `TAS` example, each service invocation is surrounded by two assignment activities like those described for `bc`; however, these have been omitted from Fig. 4 for clarity.



(a)

```
<assign name="prep_bc_input">
   <copy>
      <from>$tripData.destCity</from>
      <to>$inputBookCar.city</to>
   </copy>
   <copy>
      <from>$tripData.fromDate</from>
      <to>$inputBookCar.fromDate</to>
   </copy>
   <copy>
      <from>$tripData.toDate</from>
      <to>$inputBookCar.toDate</to>
   </copy>
</assign>
```

(b)

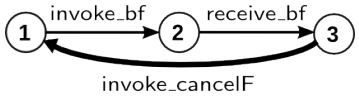
```
<assign name="save_bc_output">
   <copy>
      <from>$outputBookCar.bookingNo</from>
      <to>$tripData.carBookingNo</to>
   </copy>
   <copy>
      <from>$outputBookCar.status</from>
      <to>$tripData.carStatus</to>
   </copy>
   <copy>
      <from>$outputBookCar.type</from>
      <to>$tripData.carType</to>
   </copy>
</assign>
```

(c)

Fig. 6 (a) A fragment of Fig. 4, showing the <assign> activities placed before and after the bc service invocation. The BPEL definitions of (b) prep_bc_input and (c) save_bc_output.

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

(a)



(b)

Fig. 7 (a) BPEL definition of a flight booking service invocation (bf), including its compensation; and (b) LTS translation of the bf activity and its compensation (bold).)

3.1.2 Compensation

BPEL's *compensation* mechanism allows the definition of the application-specific reversal of completed activities. This is done by attaching a compensation handler (CH) to a <scope> or <invoke> activity: the activities defined in the CH are executed when compensation is initiated using the <compensate> activity. For example, the compensation for booking a flight (bf) is to cancel the booking (cancelF). This is described in BPEL as shown in Fig. 7a.

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 $\mathsf{compensate}(a; b)$ is $\mathsf{compensate}(b); \mathsf{compensate}(a)$.

If an activity does not provide compensation, or has already been compensated, then attempts to invoke compensation are treated as executing an <empty> activity, denoted by τ.

3.1.3 Service Pre- and Post-conditions

As mentioned at the beginning of Section 3.1, BPEL service interfaces are defined in WSDL. These service interface definitions are semantically poor: neither service requirements nor capabilities are specified. One way of improving these service specifications is to apply the principles of Design by Contract [31] to web services, where assertions are used to formally specify service contracts in terms of pre- and post-conditions: conditions that must hold before and after the execution of the service. Various projects (e.g., [24, 6, 45]) have adopted this approach for improving the quality of service specifications, each proposing their own specification language. In this work, we specify service contracts using WSCoL [6] (Web Service Constraint Language).

The grammar of the WSCoL subset used in this article can be found in the appendix, along with some examples of predicates. BPEL variables are accessed using XPath expressions, and the typical Boolean, relational and mathematical operators, as well as some pre-defined functions and quantifiers are used to build more complex expressions. Unlike other assertion languages such as JML [10], WSCoL does not use a special keyword to distinguish between the value of a variable prior to a service invocation and the value afterwards. This is because web services are treated as black boxes that expose public methods, which produce an output while leaving their input unchanged.

We can now define contracts for our services. For example, the `bc` service requires a non-empty destination city and a valid range of dates (dates are in the `YYYYMMDD` format):

```
pre_bc : (($inputBookCar/city) != ""
  && $inputBookCar/fromDate < $inputBookCar/toDate,
```

and returns a 3-digit car booking number, as well as two messages – the booking status and rental type. If the booking was successful, these messages should be set to "booked" and "car", respectively:

```
post_bc : (($outputBookCar/bookingNo) >= 1000
  && ($outputBookCar/status) == "booked"
  && ($outputBookCar/type) == "car")
```

3.2 Application Properties

Apart from the system to be monitored, our framework also receives a set of properties that the application must satisfy. These properties, provided by

the developer, are then used to monitor the run, detect errors and guide the production of recovery plans. As with any other property-based specification, it is possible that the property list is incomplete (i.e., some system requirements are not captured) or even inconsistent (i.e., satisfying the entire set of requirements is not possible). In this work, we restrict ourselves to simple properties describing *negative* behaviors (that should not appear in the application), and properties describing *positive*, or desired, behaviors (that the system must have).

For example, property P_1 of the TAS system (see Section 1) describes a positive behavior (the destination must be reached), while P_2 describes a negative scenario that should be avoided (a limousine and an expensive flight are booked). Negative scenarios are commonly called *safety* properties, and require a finite sequence of actions to witness their violations. For property P_2, one such violating witness is "book an expensive flight, and then book a limo". For safety properties, no finite sequence of actions can show satisfaction.

Positive behaviors, on the other hand, can also be (locally) satisfied. This happens when the desired sequence is fully seen even though the property calls for repeated sequences of desired behaviour. For example, for property P_1, if rd has been seen, and a new ri was not yet initiated, the specification is locally satisfied. Our notion of local satisfaction of positive behaviours is related to *finitary liveness* [2] in which there exists a fixed bound b such that something good must happen within b transitions. The bound b may be arbitrarily large, but in our case it is limited by the length of the trace collected at runtime. In many cases, properties may have both a negative and a positive component, and thus we refer to such properties as *mixed*[1].

In order to be verified, properties are translated into deterministic finite automata (DFAs), which we call "monitors". For example, Fig. 8a shows the monitor that checks property 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 transportation reaches the destination (rd). Monitor $\mathtt{A}_2$ in Fig. 8b represents the two execution paths that lead to a violation of property 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 (violating P_2). We formalize (coloured) monitors below.

Definition 3.1 (Monitor) *A* monitor *is a 5-tuple* $A = (S, \Sigma, \delta, I, F)$*, where* (S, Σ, δ, I) *is an LTS and* $F \subseteq S$ *is a set of* final *states.*

Monitor 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 states.

Let $A = (S, \Sigma, \delta, I, F)$ be a monitor. In order to facilitate recovery, we assign colours to states in S. Accepting states are coloured red, signalling

[1] Formally, mixed properties are either finitary liveness properties or a mixture of finitary liveness and safety properties.

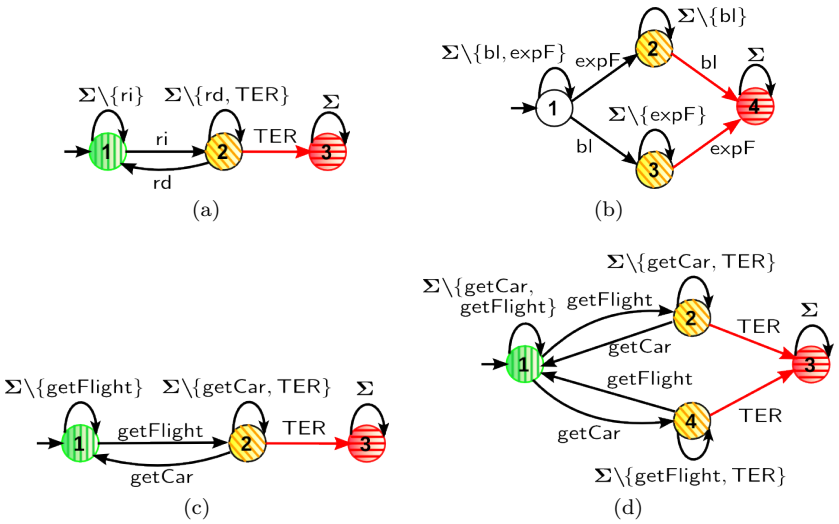


Fig. 8 Monitors: (a) $\mathtt{A}_1$, (b) $\mathtt{A}_2$, (c) $\mathtt{A}_3$, and (d) a more permissive version of A_3. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

violation of the property. State 3 of Fig. 8a and state 4 in Fig. 8b are red states (also shaded horizontally). Yellow states are those from which a red state can be reached through a single transition. Formally, for a state $s \in S$,

$$\mathit{colour}(s) = \text{yellow if } \exists a \in \Sigma, s' \in F \text{ s.t. } (s, a, s') \in \delta.$$

State 2 in Fig. 8a, and states 2 and 3 in Fig. 8b are yellow states (also shaded diagonally).

The green colour is used for 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. Formally,

$$\begin{aligned}&\mathit{colour}(s) = \text{green iff}\\ &\quad (\mathit{colour}(s) \neq \text{red}) \ \wedge \ (\mathit{colour}(s) \neq \text{yellow}) \wedge\\ &\quad (\exists a \in \Sigma, \ \exists s' \in S \text{ s.t. } (\mathit{colour}(s') = \text{yellow}) \ \wedge \ ((s', a, s) \in \delta)).\end{aligned}$$

State 1 in Fig. 8a is coloured green (also shaded vertically). States that are neither red, yellow or green are white.

Note that not all monitors have green states. For example, in $\mathtt{A}_2$ of Fig. 8b every yellow state (2 and 3) has outgoing transitions only to yellow or red states. Thus these states are "inescapable", and the monitor has no green states. A monitor with no green states is called a *safety* monitor. Otherwise, it is called a *mixed* monitor.

Definition 3.2 (Safety Monitor) *A monitor* $(S, \Sigma, \delta, I, F)$ *is a* safety monitor *if* $\forall s \in S \setminus F, \text{colour}(s) \in \{\mathit{white}, \mathit{yellow}\}$ *and* $\forall s \in F, \text{colour}(s) = \mathit{red}$.

Definition 3.3 (Mixed Monitor) *A monitor* $(S, \Sigma, \delta, I, F)$ *is a* mixed monitor *if* $\forall s \in S \backslash F, \text{colour}(s) \in \{white, yellow, green\}$ *and* $\forall s \in F, \text{colour}(s) = red$.

In this work, we assume that the set $\mathsf{A} = \{\mathsf{A}_1, ..., \mathsf{A}_n\}$ of monitors is provided by the application developers. However, we do use the BPEL application to suggest how to make some of the automata more permissive, to allow sequences of events to be recognized in any order. Recall that BPEL's <flow> captures the parallel composition of the enclosed activities, allowing their execution in any order. This allows us to flag the monitors that use these activities as events, build more permissive automata and suggest that the developers use them. For example, analyzing the `TAS` workflow in Fig. 4, we note that air (getFlight) and ground (getCar) transportation can be arranged in any order. Thus, if we had a monitor A_3 for a mixed property "if air transportation is arranged, then ground transportation should also be arranged", depicted in Fig. 8c, we would suggest to use the monitor in Fig. 8d, which allows processing of these events in any order.

In [38], we discuss property specification languages and property patterns for web service applications, as well as provide detailed algorithms for creating (regular, not more permissive) monitors from these specifications.

4 Preprocessing

The inputs to the Preprocessing stage of our framework are the BPEL program B and the set of properties expressed as monitoring automata (see the previous section). We begin by converting B into a Labelled Transition System and adding transitions on compensation actions (see Section 4.1). We then enrich the LTSs with state information coming from predicates in if statements and loops and service contracts, resulting in Doubly Labelled Transition Systems (L^2TSs). We use the state information in order to determine whether compensation is adequately specified (Section 4.2). Finally, in Section 4.3 we formalize change states and potential goal traces and provide an algorithm for computing these statically on the resulting L^2TS.

4.1 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 [21, 25, 37]. In this work, we build on Foster's [19, 20] approach of using an LTS as the underlying formalism.

4.1.1 Existing Translation

In [19, 20], Foster specified how to map all BPEL 1.1 activities into LTS. For example, Fig. 7b shows the translation of the <invoke> activity bf defined

in Fig. 7a, which returns a confirmation number. The activity is a sequence of two transitions: the actual service invocation (invoke_bf) and its return (receive_bf)[2].

Conditional activities like <while> and <if> are represented as states with two outgoing transitions, one for each valuation of the condition. The LTSs for these two activities are shown in Fig. 9a. Note that both LTSs have two transitions from state 1: $1 \xrightarrow{\mathsf{expr\_true}} 2$ and $1 \xrightarrow{\mathsf{expr\_false}} 3$. <pick> is also a conditional activity, but can have one or more outgoing transitions: one for each <onMessage> branch (there are two of these in the example in Fig. 9a). <sequence> and <flow> activities result in the sequential and the parallel composition of the enclosed activities, respectively (see Fig. 9b). Note that BPEL processes may have multiple <receive> activities. For such processes, the BPEL engine non-deterministically chooses a creation point in the execution. In his thesis, Foster assumes that processes have only one creation point, the first <receive> that appears in the process definition. We make the same assumption in this work.

Thus, we represent a BPEL program B by its LTS translation $\mathsf{L}(\mathsf{B})$. The set of labels Σ of $\mathsf{L}(\mathsf{B})$ is derived from the possible events in 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 S in $\mathsf{L}(\mathsf{B})$ consists of the states produced by the translation as well as a new state t, reachable from any state of S via a TER event: $\forall s \in S \setminus \{\mathsf{t}\}, (s, \mathsf{TER}, \mathsf{t}) \in \delta$.

4.1.2 Formalizing Compensation

In order to capture BPEL's compensation mechanism, we introduce additional, backwards transitions. For example, the compensation for bf, specified in Fig. 7a, is captured by adding the transition $3 \xrightarrow{\mathsf{invoke\_cancelF}} 1$ as shown in Fig. 7b. 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. 10a shows $\mathsf{L_C}(\mathtt{TAS})$. 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. 7b has two transitions between states 1 and 3: $1 \xrightarrow{\mathsf{bf}} 3$

[2] Foster's translation includes partner, activity and variable names in the labels, in order to include traceability information, but we omit these in this paper for simplicity.

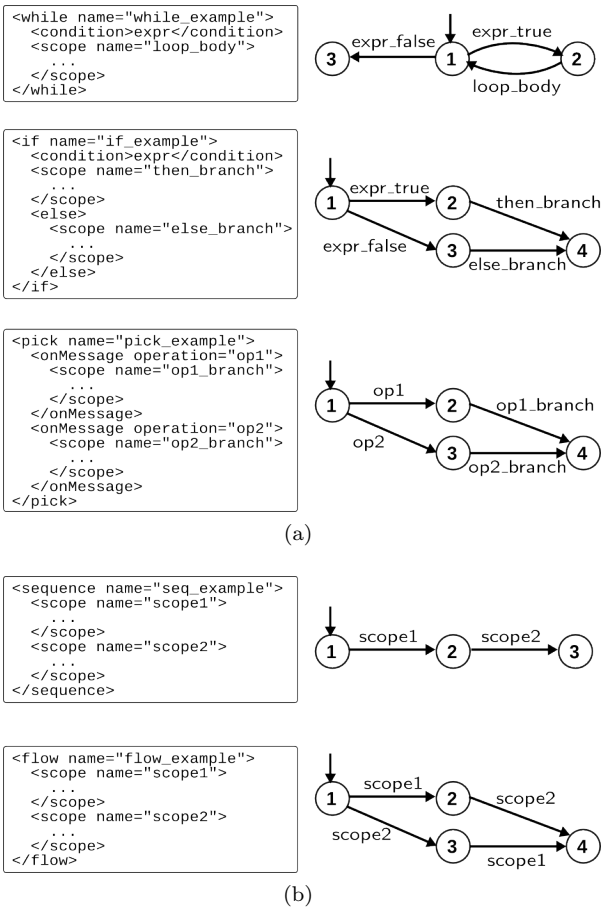


Fig. 9 (a) BPEL conditional activities and their corresponding LTSs; (b) BPEL structural activities and their corresponding LTSs.

and $3 \xrightarrow{\mathsf{cancelF}} 1$. 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. 10a represents two transitions: $(s, a, t) \in \delta$ and $(t, \bar{a}, s) \in \delta_c$.

The <pick> activity (◈ labeled ① in Fig. 4) corresponds to state 2 of Fig. 10a. The choice between onlyCar and carAndFlight is represented by two outgoing transitions from this state: $(2, \mathsf{onlyCar}, 3)$ and $(2, \mathsf{carAndFlight}, 6)$. Since these actions do not affect the state of the application, they are compensated by τ. The <flow> activity (⊕ labeled ② in Fig. 4) 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.

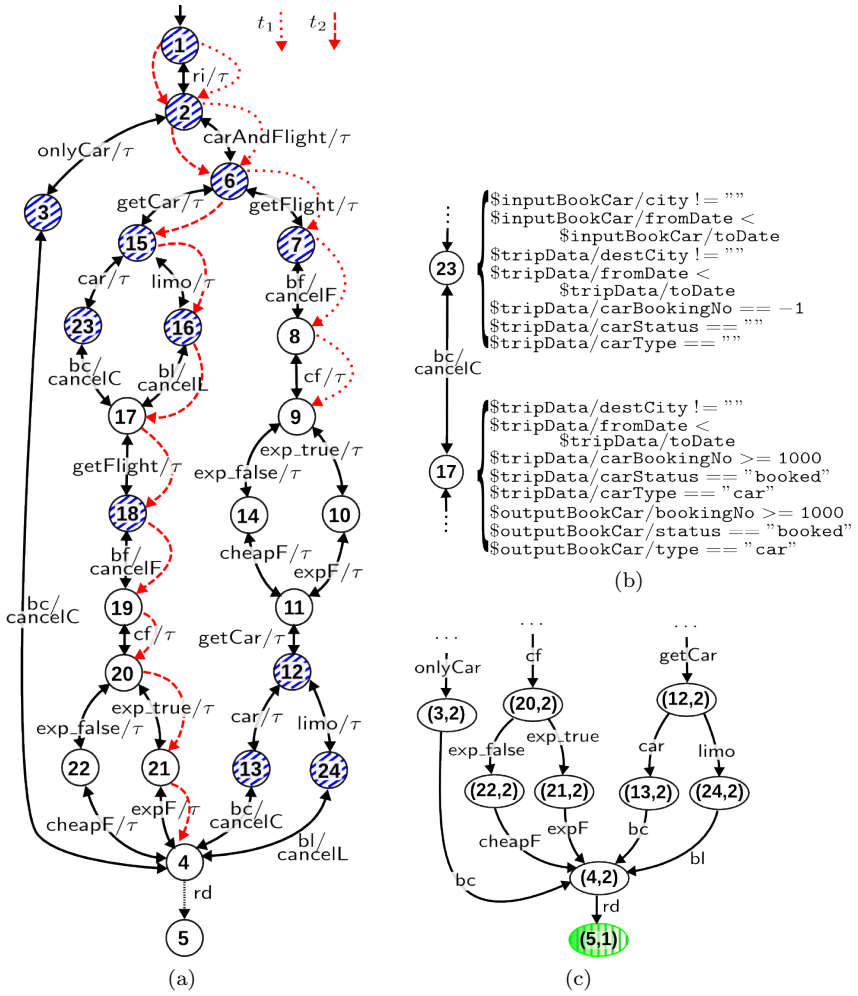


Fig. 10 (a) LTS $\mathsf{L}_{\mathsf{C}}(\mathtt{TAS})$: downward and upward arrows show forward and compensation logic, respectively. 1-step goal transitions are depicted by tiny-dashed transitions and change states are shaded diagonally in purple. Traces t_1 (dotted) and t_2 (dashed) are also shown; (b) A fragment $\mathrm{L}^2\mathrm{TS}$ $\mathsf{L}_{\mathsf{C}}(\mathtt{TAS})$, showing the predicates collected for states 17 and 23; (c) A fragment of $\mathsf{L}(\mathtt{TAS}) \times \mathtt{A}_1$.

4.2 Adequate Compensation

According to the BPEL standard, compensation for a service is well-defined if: 1) the compensation activity can be executed after the service has been invoked, and 2) the execution of a service's compensation activity leaves the application in a state where the service can be executed again. This corresponds to our translation of compensation, where compensation for an activity leaves the application back in its original state. However, we cannot guarantee that this condition holds at runtime, and thus, some of our recovery plans may fail.

In the future, we plan to improve this result by using information about the application's state and service contracts to statically check these conditions.

We begin by defining how we collect state information. Conditional activities like <if> and <while> *generate* predicates about the state of the system. Another source of predicates are service pre- and post-conditions. For example, the precondition for the booking service, `pre_bc`, defined in Section 3.1.3, generates two predicates: $\texttt{\$inputBookCar/fromDate} < \texttt{\$inputBookCar/toDate}$ and $\texttt{\$inputBookCar/city} \mathrel{!}= ""$.

Assignment activities can both *generate* and *kill* predicates. For example, in the state before executing bc, we know that $\texttt{\$tripData/carType} == ""$ is true (derived from the BPEL variable initialization). We also know that $\texttt{\$outputBookCar/type} == "\texttt{car}"$ should hold in the state after invoking bc, since it is guaranteed by its post-condition. Now, since the <assign> activity save_bc_output copies the value of $\texttt{\$outputBookCar/type}$ to $\texttt{\$tripData/carType}$, we can conclude that $\texttt{\$tripData/carType} == "\texttt{car}"$ holds after save_bc_output is executed (so $\texttt{\$tripData/carType} == ""$ is killed).

We use standard data-flow analysis techniques to propagate these predicates for the L^2TS $\mathsf{L}(\mathsf{B})$, constructing $\mathscr{L}(s)$ – the set of predicates that hold in state s. Fig. 10b shows the result of this analysis on states 17 and 23 of the LTS $\mathsf{L}(\texttt{TAS})$. In the rest of this paper, we will abuse the notation somewhat, referring to both the LTS and the L^2TS of a program B as $\mathsf{L}(\mathsf{B})$, and doing the same for the version with compensation, $\mathsf{L}_\mathsf{C}(\mathsf{B})$. We will explicate its type if it is not clear from the context.

Now we can define *adequate compensation*:

Definition 4.1 (Adequate Compensation) *Let* B *be a BPEL application and* $\mathsf{L}_\mathsf{C}(\mathsf{B}) = \{S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I, \mathscr{L}\}$ *be the* L^2*TS of* B *with compensation. A transition* $(s, a, s') \in \delta$ *has* adequate compensation *when used in* B *if there is a transition* $(s', a_C, s) \in \delta_C$*, where* a_C *is the compensation action for* a*, and both* $\{postcond(a) \wedge \mathscr{L}(s') \Rightarrow precond(a_C)\}$ *and* $\{postcond(a_C) \wedge \mathscr{L}(s) \Rightarrow precond(a)\}$ *hold. Actions compensated by* τ *(the empty action) are always adequately compensated.*

In the rest of this work, when we refer to the L^2TS $\mathsf{L}_\mathsf{C}(\mathsf{B})$, we assume that δ_c only contains adequate compensation transitions.

For example, according to Fig. 10b, cancelC is the compensation action for bc. Suppose the contract for cancelC is as follows:

$$\begin{aligned}&\texttt{pre\_cancelC} : \big((\texttt{\$inputCancelCar/bookingNo}) >= 1000\big)\\ &\texttt{post\_cancelC} : \big((\texttt{\$outputCancelCar/status}) == ""\\ &\quad \&\& \ (\texttt{\$outputCancelCar/type}) == ""\big),\end{aligned}$$

and suppose that the corresponding <assign> activities prep_cancelC_input and save_cancelC_output are similar to those defined for bc. Then the compensation is adequately defined since

$$\texttt{post\_bc} \wedge \mathscr{L}(17) \Rightarrow \texttt{pre\_cancelC}$$

and

$$\texttt{post\_cancelC} \wedge \mathscr{L}(23) \Rightarrow \texttt{pre\_bc}$$

We perform such checks using an SMT solver, such as MathSAT [16].

What happens if an SMT solver is unable to prove that the compensation is adequate? The reason might be an error in the specification or simply incomplete state information (since our predicate-collecting approach is pretty "light"). In such cases, we have to rely on user feedback – they can augment our automated analysis by either (1) confirming that the compensation is not adequate and removing it; (2) changing its contract, whereby we would again try to prove adequacy; or (3) stating that it is adequate, without proof, in which case we cannot guarantee that plans that use this activity will run to completion (i.e., the correctness claims we make in Sections 6.2 and 7.4).

4.3 Goal Traces and Change States

The last part of the preprocessing phase statically identifies key states and transitions of the application $\mathsf{L}(\mathsf{B})$, aimed to help find an efficient recovery plan when a violation is encountered (see Section 6 and Section 7).

4.3.1 Goal Traces

In order to find a good recovery plan, we first need to compute a set of *goal traces*, that is, traces which eventually result in the satisfaction of some properties. We compute these on a per-property basis. Further, recall that only mixed properties can be satisfied, which is indicated by the monitor reaching a green state; safety properties can only be violated. Thus, for each mixed monitor A, we are looking for traces in $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I, \mathscr{L})$ which make A enter its green state(s). We find those using the cross-product between the model and the automaton, $\mathsf{L}(\mathsf{B}) \times \mathsf{A}$. The traces can vary in length, using a parameter i: as i gets larger, the analysis gets more precise but more expensive (see the experiments in Section 9).

Definition 4.2 (i-Step Goal Trace) *$s_0 a_0 \ldots s_{i-1} a_{i-1} s_i$ is an* i-step goal trace *in $\mathsf{L}(\mathsf{B})$ iff $\exists q_0 a_0 \ldots q_{i-1} a_{i-1} q_i \in \mathsf{A}$ s.t. $\text{colour}(q_i) = \textit{green} \wedge \forall j, (0 \leq j < i) \Rightarrow ((s_j, q_j) \xrightarrow{a_j} (s_{j+1}, q_{j+1}) \in \delta_{\mathsf{L}(\mathsf{B})\times\mathsf{A}}) \wedge (\text{colour}(q_j) \neq \textit{green})$.*

That is, the last step of the trace, $s_{i-1} \xrightarrow{a_{i-1}} s_i$, corresponds to taking a transition on a_{i-1} into a green state of A. s_i is called a *goal state*. The resulting set of goal traces is denoted by $\mathsf{G_i}(\mathsf{B}, \mathsf{A})$. For example, consider the fragment of $\mathsf{L}(\texttt{TAS}) \times \texttt{A}_1$ shown in Fig. 10c. The green state of $\texttt{A}_1$ is state 1, with a transition on rd leading to it. In this example, the set of 1-step goal traces is $\mathsf{G}_1(\texttt{TAS}, \texttt{A}_1) = \{(4, \mathsf{rd}, 5)\}$ (depicted by tiny-dashed transitions in Fig. 10a). The set of two-step goal traces in our example is

$$\begin{aligned}\mathsf{G}_2(\texttt{TAS}, \texttt{A}_1) = \{&(3, \mathsf{bc}, 4, \mathsf{rd}, 5), (22, \mathsf{cheapF}, 4, \mathsf{rd}, 5), (21, \mathsf{expF}, 4, \mathsf{rd}, 5),\\ &(13, \mathsf{bc}, 4, \mathsf{rd}, 5), (24, \mathsf{bl}, 4, \mathsf{rd}, 5)\}.\end{aligned}$$

Note that while it is computed using $\mathsf{L}(\mathsf{B}) \times \mathsf{A}$, the set $\mathsf{G_i}(\mathsf{B},\mathsf{A})$ contains transitions only of $\mathsf{L}(\mathsf{B})$.

Given a set of goal traces $\mathsf{G_i}(\mathsf{B},\mathsf{A})$, let $\mathsf{G_i^s}(\mathsf{B},\mathsf{A}) = \{(s_0, \langle a_0, ..., a_{i-1}\rangle)\}$ s.t. $(s_0, a_0, \ldots, s_{i-1}, a_{i-1}, s_i) \in \mathsf{G_i}(\mathsf{B},\mathsf{A})$, i.e., we explicate a *source state*, i steps away from a goal state, and an i-step sequence of actions that can be executed to reach the goal state. In our example,

$$\mathsf{G_2^s}(\mathtt{TAS}, \mathtt{A_1})) = \{(3, \langle \mathsf{bc}, \mathsf{rd}\rangle), (13, \langle \mathsf{bc}, \mathsf{rd}\rangle), (21, \langle \mathsf{expF}, \mathsf{rd}\rangle), \\ (22, \langle \mathsf{cheapF}, \mathsf{rd}\rangle), (24, \langle \mathsf{bl}, \mathsf{rd}\rangle)\}.$$

When we compute recovery plans, we need to direct the application towards executing its goal traces, i.e., towards reaching source states and then executing the follow-on sequences of actions that would reach the goal state and move the corresponding monitor to the green state.

4.3.2 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 modelling 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". Services whose result is fully determined by the input are sometimes referred to as *idempotent* services, since multiple invocations of the same service yield the same results. We refer to all other services as *non-idempotent*. 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.

Definition 4.3 (Change State) *A state is a* change state *if it is identified by: 1) a <flow> activity, 2) a <pick> activity, or 3) a non-idempotent service call.*

We denote by $\mathsf{C}(\mathsf{B})$ the set of all change states in the L^2TS of the application B. For example, in the L^2TS in Fig. 10a, 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. 10a by purple diagonal shading.

A recovery plan should pass through at least one change state, to allow a change in the execution. Furthermore, if all paths from a change state lead to an error, such change states are not useful for recovery and should be removed from $\mathsf{C}(\mathsf{B})$ (the `TAS` example does not have such states). Candidates for removal can be easily identified using static analysis. In the remainder of the paper, we assume that $\mathsf{C}(\mathsf{B})$ has already been filtered using this technique.

5 Runtime Monitoring

The runtime monitoring phase uses the set of monitors to analyze the BPEL program B as it runs on a BPEL-specific Application Server. The runtime monitoring component of our recovery framework is based on that of [43], which has been implemented within the IBM WebSphere business integration products [26]. We capture events in Σ as they pass between the application server and the program, and use these events to update the state of the monitors and store them as part of the execution trace T. By default, service timeouts trigger application termination. Monitors can be dynamically enabled (e.g., to monitor new properties) and disabled (e.g., to reduce monitoring overhead). 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.

In our earlier work [43], we have used the *interception mechanism* based on *eavesdropping* – watching events as they pass between partners and updating monitors accordingly. While adequate for identifying and reporting property violations, it is insufficient for recovery. For example, we do not want to execute a TER event before knowing whether its execution causes any monitor violations, since we cannot reverse application termination. We also want to avoid executing other events that may directly lead to monitor violation, since these events will be inevitably compensated during recovery. Thus, instead of allowing all events to pass, our monitoring component delays the delivery of events that cause termination or property violation. If no violation is detected during analysis, the event is delivered and execution continues as usual. Otherwise, the event is not delivered and recovery is initiated – our framework computes and presents a set of recovery plans, and the BPEL engine attempts to execute the plan chosen by the user. Using a <collaboration> scope, the corresponding compensation handler is updated with the plan chosen by the user and then activated (<collaboration> scopes can be used to dynamically modify BPEL applications, see Section 8 for details).

During the execution of the application B, represented as $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I, \mathscr{L})$, we store its trace

$$\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n.$$

We say that T is a *successful* trace iff $\forall \mathsf{A_i} \in \mathsf{A}$, $a_0 a_1 ... a_{n-1}$ is rejected by $\mathsf{A_i}$. T is a *failure* (or an *error*) trace iff $\exists \mathsf{A_i} \in \mathsf{A}$ s.t. $a_0 a_1 ... a_{n-1}$ is accepted by $\mathsf{A_i}$. In such a case, state s_n is an *error state* of the application.

Recall that each state of the L^2TS $\mathsf{L(B)}$ had information about predicates harvested from control flow conditions and service contracts. At runtime, we maintain the exact values of variables over which these predicates were defined, which allows us to check correctness of service contracts. Specifically, if a step s_i of the trace T arrives at an invocation of some service sv_i, we check that sv_i's precondition holds. For example, ($\texttt{\$inputBookCar/city!} = \text{""}$) is part of bc's precondition (see Section 3.1.3). During the run, we keep the values $\texttt{\$inputBookCar/city}$ in each state and, if upon reaching state 17 (see Fig. 10a), its value is "", then the precondition is violated and we report an error. In a similar manner, we can check whether service sv_i leaves the application in the expected state by checking whether $\texttt{post}(sv_i)$ holds after executing sv_i. These checks are, in effect, application-independent safety properties (e.g., "the system does not have behaviours which violate stated pre- and postconditions for service invocations") that we check without constructing special-purpose automata. We discuss recovery from such violations in Section 6.

In addition to T, we also store traces $\mathsf{T}^{\mathsf{A}_1}...\mathsf{T}^{\mathsf{A}_n}$ that correspond to the executions of the *monitors* $\mathsf{A}_1...\mathsf{A}_n$, respectively. These are used in the recovery phase to reverse the state of the monitors. Note that all traces corresponding to a single execution differ in their *states* (e.g., application states are different from states of each monitor) but agree on the *events* which got executed. In what follows, traces corresponding to the application have no superscripts, whereas monitor traces are superscripted.

For example, 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 times out. This scenario corresponds to the trace t_1, depicted by dotted transitions in Fig. 10a. In addition to t_1, our tool stores $t_1^{\mathsf{A}_1}$ and $t_1^{\mathsf{A}_2}$ – the corresponding traces of the enabled monitors:

$$
\begin{aligned}
t_1 &= 1 \xrightarrow{\mathsf{ri}} 2 \xrightarrow{\mathsf{carAndFlight}} 6 \xrightarrow{\mathsf{getFlight}} 7 \xrightarrow{\mathsf{bf}} 8 \xrightarrow{\mathsf{cf}} 9,\\
t_1^{\mathsf{A}_1} &= 1 \xrightarrow{\mathsf{ri}} 2 \xrightarrow{\mathsf{carAndFlight}} 2, \xrightarrow{\mathsf{getFlight}} 2 \xrightarrow{\mathsf{bf}} 2 \xrightarrow{\mathsf{cf}} 2,\\
t_1^{\mathsf{A}_2} &= 1 \xrightarrow{\mathsf{ri}} 1 \xrightarrow{\mathsf{carAndFlight}} 1 \xrightarrow{\mathsf{getFlight}} 1 \xrightarrow{\mathsf{bf}} 1 \xrightarrow{\mathsf{cf}} 1.
\end{aligned}
$$

The application server detects that the cf invocation timed out, and sends a TER event (not shown in Fig. 10a) 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.

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. 10a (the trace corresponding to A_1 is omitted):

$$t_2 = 1 \xrightarrow{\mathsf{ri}} 2 \xrightarrow{\mathsf{carAndFlight}} 6 \xrightarrow{\mathsf{getCar}} 15 \xrightarrow{\mathsf{limo}} 16 \xrightarrow{\mathsf{bl}} 17 \xrightarrow{\mathsf{getFlight}} 18 \xrightarrow{\mathsf{bf}} 19 \xrightarrow{\mathsf{cf}} 20 \xrightarrow{\mathsf{exp\_true}} 21 \xrightarrow{\mathsf{expF}} 4.$$

$$t_1^{\mathsf{A}_2} = 1 \xrightarrow{\mathsf{ri}} 1 \xrightarrow{\mathsf{carAndFlight}} 1 \xrightarrow{\mathsf{getCar}} 1 \xrightarrow{\mathsf{limo}} 1 \xrightarrow{\mathsf{bl}} 3 \xrightarrow{\mathsf{getFlight}} 3 \xrightarrow{\mathsf{bf}} 3 \xrightarrow{\mathsf{cf}} 3 \xrightarrow{\mathsf{exp\_true}} 3 \xrightarrow{\mathsf{expF}} 4.$$

Before delivering an event to the application, our framework first checks whether doing so would cause an error. For example, during the execution of t_2, when the application is in state 21 and the monitor A_2 is in state 3, we note that the monitor has a transition on expF to its error state. Thus, our framework checks whether the next received event is expF. If so, t_2 is determined to be a failing trace, initiating recovery. Otherwise, the event is passed to the monitor and the application, and the monitoring continues.

6 Recovery Plans for 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 6.1 (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 mixed properties is described in Section 7.

6.1 Computing Plans

The recovery procedure for a safety property violation receives $\mathsf{L_C(B)}$ – the L^{2}TS of the running application B enriched with compensation (see Section 4.1), T – the executed trace ending in an error state e (see Section 5) and $\mathsf{C(B)}$ – the set of change states (see Section 4.3.2). It does not need to know which monitor discovered a violation and thus works the same way for violation of application-specific safety properties and service contracts.

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 Section 5 and shown in Fig. 10a. $\{1, 2, 6, 15, 16, 18\}$ are the change states seen along t_2. This leads to the recovery plans shown in Fig. 11a. We add state names between transitions for clarity and refer to plans as to mean "recovery to state s". A given plan

$r_{18} = 4 \xrightarrow{\tau} 21 \xrightarrow{\tau} 20 \xrightarrow{\tau} 19 \xrightarrow{\mathsf{cancelF}} 18$ $\quad r_6 = r_{15} \xrightarrow{\tau} 6$

$r_{16} = r_{18} \xrightarrow{\tau} 17 \xrightarrow{\mathsf{cancelL}} 16$ $\quad r_2 = r_6 \xrightarrow{\tau} 2$

$r_{15} = r_{16} \xrightarrow{\tau} 15$ $\quad r_1 = r_2 \xrightarrow{\tau} 1$

(a)

```
<sequence name="r18">
  <compensateScope target="expF" />
  <compensateScope target="exp_true" />
  <compensateScope target="cf" />
  <compensateScope target="bf" />
</sequence>
```

(b)

Fig. 11 (a) Plans for TAS for recovery from the safety violation of trace t_2; (b) XML version of recovery plan r_{18}.

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 A_2.

Computed plans are then converted to BPEL for presentation to the user. For example, plan r_{18} is shown in Fig. 11b. The chosen plan can then be applied (see Section 8), allowing the program to continue its execution from the resulting change state.

6.2 Analysis

Let $\mathsf{L}(\mathsf{B})$ and $\mathsf{L}_\mathsf{C}(\mathsf{B})$ be L^{2}TSs, as previously defined, and let $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ be an execution trace, where $s_n = e$ (the error state), and $r_l = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_l} s_l$ be a safety recovery plan for T that leaves the application in state s_l, where c_k compensates a_k and $s_l \in \mathsf{C}(\mathsf{B})$. Finally, let $\mathsf{C}_\mathsf{T}(\mathsf{B})$ be the set of change states that appear in T: $\mathsf{C}_\mathsf{T}(\mathsf{B}) = \{s | (s, a, s') \in \mathsf{T} \wedge s \in \mathsf{C}(\mathsf{B})\}$. Note that we cannot guarantee the correct execution of a recovery plan if a service that appears in the plan times out during its execution.

Definition 6.2 (Compensated Execution Trace) *Let* $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ *be an execution trace, where* $s_n = e$ *(the error state), and* $r_l = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_l} s_l$ *be a safety recovery plan for* T*. The* compensated execution trace T_{r_l} *is the result of applying* r_l *to* T*, assuming that none of the participating services timeout. In other words,* $\mathsf{T}_{r_l} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{i-1}} s_l$.

Proposition 6.1 *Let* $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ *be an execution trace, where* $s_n = e$ *(the error state), and* $r_l = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_l} s_l$ *be a*

safety recovery plan for T. *If compensation for actions* $a_l, ..., a_{n-1}$ *is adequate, and the default BPEL compensation order is observed at runtime, then the compensated execution trace* T_{r_l} *is the* l*-length prefix of* T*, i.e.,* $\mathsf{T}_{r_l} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{l-1}} s_l$*, where* $s_l \in \mathsf{C(B)}$.

This proposition follows from Definitions 4.1 and 6.2. Since the default BPEL compensation respects the forward execution order during compensation, compensating the tail of the execution trace leaves the application in the state from which the tail sequence of actions was executed.

The exact number of plans is determined by the number of change states encountered along the trace. 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 with the overall number of compensation actions fewer than 10").

In the worst case, the maximum number of plans and the maximum plan length are both at least n, $|\mathsf{C_T(B)}| = n - 1$, and each transition in T is compensatable. In other words, each non-error state in T is a change state, and each one is reachable from the current error state. According to our approach, we compute one recovery plan for each state in $\mathsf{C_T(B)}$, so computing recovery plans for safety violations is linear in the size of the error trace T.

In the average case, we expect that the maximum number of plans will be smaller than the size of the average execution trace, since execution traces contain many BPEL-induced actions that are not used to identify change states. We also expect that developers will set the maximum number of plans to be generated to a relatively small number (e.g., five) thus making recovery plan generation very feasible in practice.

7 Recovery Plans from Mixed Property Violations

Failure of a mixed 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, the monitor that identified the violation,
- $\mathsf{L_C(B)}$, the L^{2}TS of the application,
- $\mathsf{G_i(B, A)}$, the set of i-step goal traces corresponding to A (and its refactored version, explicating source states and sequences of actions, $\mathsf{G_i^s(B, A)}$),
- 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 behaviour to reach the goal (see Fig. 2 for a schematic view of the overall process). Our solution adapts techniques from the field of *planning* [18], described below.

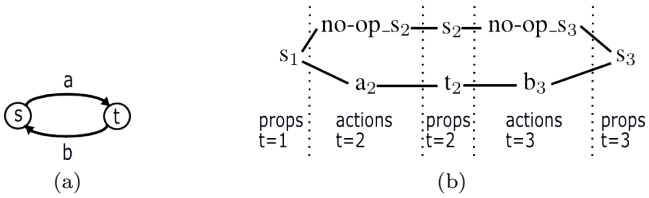


Fig. 12 (a) a simple LTS and (b) its encoding as a planning graph of size 3.

7.1 Recovery as a planning problem

A *planning problem* is a triple $P = (D, init, G)$, where D is the domain, *init* 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 *init* 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 $\mathsf{P}(\mathsf{B}, \mathsf{A}, \mathsf{T})$, we use $\mathsf{L_C}(\mathsf{B})$ as the domain and e as the initial state. The third component needed is a set of goal states. For that, we use source states from $\mathsf{G_i^s}(\mathsf{B}, \mathsf{A})$. I.e., we aim to compute a path (of length k) from the initial state to one of the states s s.t. $(s, \langle a_0, ..., a_{i-1}\rangle)$ in $\mathsf{G_i^s}(\mathsf{B}, \mathsf{A})$ and then extend it with $\langle a_0, ..., a_{i-1}\rangle$, resulting with a plan of length $k + i$.

For example, consider the trace t_1 of Fig. 10a, described in Section 5, in which monitor $\mathtt{A_1}$ fails and assume that we want to use goal traces of length 1. 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. 10a) and $\mathsf{G_1^s}(\mathtt{TAS}, \mathtt{A_1}) = \{(4, \langle \mathsf{rd}\rangle)\}$. The resulting plan p should than be extended to $p \xrightarrow{\mathsf{rd}} 5$.

Unfortunately, not every trace returned by solving $\mathsf{P}(\mathsf{B}, \mathsf{A}, \mathsf{T})$ is acceptable: the recovery plans for mixed violations should 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. 12a, 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. 12b and is read as follows: at time 1 we begin in state s_1. If action a occurs (modelled as a_2), then at time 2 we move to state t (modelled 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 (modelled 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 planner-specific treatment of a sequence of no-ops.

Several existing planners, such as BlackBox [28], translate the planning graph into a CNF formula and then use a SAT solver, such as SAT4J [8], 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. 12b is as follows:

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

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 $\mathtt{s}$, if one exists. The desired plan is extracted from $\mathtt{s}$ by taking propositions that correspond to actions and that are assigned positive values in $\mathtt{s}$. For the above example, these are a_2, b_3 and "do nothing".

Our approach has been inspired by existing work on a related problem – that of automatically creating new web service compositions that accomplish non-trivial tasks [29, 33, 30, 44]. In this case, the planning domain is the set of available web services, the goal is a specification of the desired behaviour, and plans are service compositions that accomplish the desired behaviour. Research in this area has focused on using different planning techniques to solve this problem in an efficient manner, dealing with the nondeterministic behaviour of web services, the partial observability of their internal status, and the specification of complex goals expressing temporal conditions and preference requirements. In this work, we do not use planning to generate new service compositions, but use it instead to explore existing applications. However, the approach presented here can be augmented with the work presented in [33, 30, 44], especially in the case when there is not enough compensation or redundancy in the application to permit the computation of recovery plans according to our approach.

In what follows, we first discuss how to produce a single recovery plan using a SAT-based approach (Section 7.2) and then show how to extend it to produce multiple plans (Section 7.3).

7.2 Producing a single recovery plan

Let $\mathsf{f_P}$ be the encoding of the planning problem $\mathsf{P(B, A, T)}$ produced by an existing planner. We augment $\mathsf{f_P}$ to follow our "compensate until a change state and then execute" 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 Section 7.5).

1. We want to make sure a recovery plan visits at least one of the change states encountered on the execution trace T. Let $\mathsf{S(T)}$ be the set of states on T. We define $\mathsf{C(T)} = \mathsf{S(T)} \cap \mathsf{C(B)}$ to be the change states that appear on T and denote by $c^1, ..., c^n$ the propositions that correspond to states in $\mathsf{C(T)}$. 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. 12 again. If x is a change state and $k = 3$, then propositions x_1, x_2, x_3 in $\mathsf{f_{lts}}$ correspond to expansions of x 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, $\mathsf{c} = (x_1 \vee x_2 \vee x_3)$. This formula is true when at least one of the change states in $\mathsf{C(T)}$ is part of the plan.

2. In order to further lead the planner towards the "compensate and execute" 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, to avoid having the compensations $\mathsf{cancelC}$ and $\mathsf{cancelF}$ in a plan of length 2, we would define a as

$$\mathsf{a} = \neg \mathsf{cancelC}^1 \wedge \neg \mathsf{cancelC}^2 \wedge \neg \mathsf{cancelF}^1 \wedge \neg \mathsf{cancelF}^2$$

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 $\mathsf{P(B, A, T)}$, 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.

7.3 Producing multiple recovery plans

Let π_0 be the plan produced for $\mathsf{R_0(f_P)}$ (see Section 7.2), leading to a state g (we called it the *source state*) s.t. $(g, \langle a_0, ... a_{i-1} \rangle) \in \mathsf{G_i^s(B, A)}$. 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 the set of desired goal states and repeat the process described in Section 7.2. The new plan 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. 12a, we use $\mathsf{R_0(f_{lts})}$ computed in Section 7.2 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 $\mathsf{R_j(f_P)}$ with the set of previously computed plans to get $\mathsf{R_{j+1}(f_P)}$, 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. 10a and ending in state 9. Looking for plans up to length 10 and using

$$p_0 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\tau} 6 \xrightarrow{\tau} 2 \xrightarrow{\text{onlyCar}} 3 \xrightarrow{\text{bc}} 4$$
$$p_1 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\text{bf}} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp\_true}} 10 \xrightarrow{\text{expF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{car}} 13 \xrightarrow{\text{bc}} 4$$
$$p_2 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\text{bf}} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp\_false}} 14 \xrightarrow{\text{cheapF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{car}} 13 \xrightarrow{\text{bc}} 4$$
$$p_3 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\text{bf}} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp\_false}} 14 \xrightarrow{\text{cheapF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{limo}} 24 \xrightarrow{\text{bl}} 4$$

(a)

```
<sequence name="p0">
  <compensateScope target="cf" />
  <compensateScope target="bf" />
  <compensateScope target="getFlight" />
  <compensateScope target="carAndFlight" />
  <pick name="transport" ... >
     <onMessage operation="onlyCar" ... >
        ...
     </onMessage>
  </pick>
  <invoke operation="bc" ... />
</sequence>
```

(b)

$$p'_1 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp\_true}} 10 \xrightarrow{\text{expF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{car}} 13 \xrightarrow{\text{bc}} 4$$
$$p'_2 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp\_false}} 14 \xrightarrow{\text{cheapF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{car}} 13 \xrightarrow{\text{bc}} 4$$
$$p'_3 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp\_false}} 14 \xrightarrow{\text{cheapF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{limo}} 24 \xrightarrow{\text{bl}} 4$$

(c)

Fig. 13 (a) Plans for `TAS` of length ≤ 10 for recovery from the mixed property violation of trace t_1; (b) XML version of recovery plan p_0; and (c) plans $p'_1 - p'_3$, the result of applying *plan-loop filtering* to plans $p_1 - p_3$.

goal traces of length 1, we get plans $p_0 - p_3$ shown in Fig. 13a. Each of these is extended with the last transition $4 \xrightarrow{\text{rd}} 5$.

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 through p_3 also cancel the flight (since 8 is not a change state whereas 7 is) and then proceed to re-book it and book a car (p_1, p_2) or a limo (p_3). 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 then ordered based on the length of the plan and the number of compensation actions in it. For example, plan p_0 is the shortest and has only one compensation action, for carAndFlight. Thus, it is the first plan to be presented.

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 some of the safety properties. These optimizations are described in detail in [41] and are omitted in this paper. Section 9 includes several small experiments to show their effectiveness.

Chosen plans are then converted to BPEL for execution. The compensation part of the plan is similar to the one shown in Fig. 11b, and the re-planning part

consists of a sequence of BPEL <invoke> operations. The XML translation of plan p_0 is shown in Fig. 13b.

7.4 Analysis

In the previous subsections, we described how to compute a plan p which first compensates the trace until a change state is reached and then computes an alternative path to a certain goal. Under which conditions can we guarantee that executing such a plan effectively leaves the system in a desired state?

In what follows, let B be the BPEL application, $\mathsf{L}(\mathsf{B}) = \{S, \Sigma, \delta, I, \mathscr{L}\}$ be the L^2TS that represents B, and $\mathsf{C}(\mathsf{B})$ be the set of change states of application B. Let $\mathsf{L}_\mathsf{C}(\mathsf{B}) = \{S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I, \mathscr{L}\}$ be the L^2TS of application B with compensation, where Σ_c is the set of compensation actions and δ_c the set of compensation transitions, computed as described in Section 4.1.

Let $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$, where $s_n = e$, be a trace of the program leading to an error, A be the mixed monitor that detected the violation and $\mathsf{G}^\mathsf{s}_\mathsf{i}(\mathsf{B}, \mathsf{A})$ be the set of (source state, action sequence) pairs corresponding to A. Let $p = (p_l, p_m, p_i)$ be a *mixed recovery plan* for T that tries to lead the application to the goal state s_m through the change state s_l. The first part of the plan, p_l, compensates trace T, leaving the application in state s_l. The second part, p_m, is a trace that leads to the state s_m such that $(s_m, \langle ... \rangle) \in \mathsf{G}^\mathsf{s}_\mathsf{i}(\mathsf{B}, \mathsf{A})$ when executed from s_l. That is, $p_l = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_l} s_l$, where c_k compensates a_k and $s_l \in \mathsf{C}(\mathsf{B})$, and $p_m = s_l \xrightarrow{b_0} s_j \xrightarrow{b_1} s_{j+1} \xrightarrow{b_2} ... \xrightarrow{b_{m-1}} s_m$. The third part, p_i, is a trace that executes the action sequence associated with state s_m in $\mathsf{G}^\mathsf{s}_\mathsf{i}(\mathsf{B}, \mathsf{A})$.

Let T_{p_l} be the compensated execution trace resulting from the application of p_l to T according to Definition 6.2. If we assume that compensation for actions $a_l, ..., a_{n-1}$ is adequate, and that the default BPEL compensation order is observed at runtime, then, according to Proposition 6.1, the execution of p_l leaves the application in state s_l. Again, we cannot guarantee the correct execution of a recovery plan if a service that appears in the plan times out during its execution.

Definition 7.1 (Updated Execution Trace) *Let T be an execution trace, $p = (p_l, p_m, p_i)$ be a mixed recovery plan and T_{p_l} be the compensated execution trace (as defined above). The* updated execution trace T_{p_i} *is the result of applying the sequence of actions associated with p_m, followed by the sequence of actions associated with p_i to the compensated execution trace T_{p_l}, assuming that none of the participating services timeout.*

In other words, the updated execution trace T_{p_i} is the result of executing the sequence $b_0 b_1 ... b_m$ from state s_l, followed by the i-step sequence associated with state $s_m \in \mathsf{G}^\mathsf{s}_\mathsf{i}(\mathsf{B}, \mathsf{A})$. Note that BPEL applications considered in this paper have several sources of non-determinism (from <pick> and <flow> activities). In these cases, we suggest a valid interleaving of events in the plan.

An incorrect interleaving is caught by the monitors during plan execution. In the proposition below, we define sufficient conditions under which the trace produced as a result of executing p_m followed by p_i reaches a goal state.

Proposition 7.1 *Let* T *be an execution trace,* $p = (p_l, p_m, p_i)$ *be a mixed recovery plan and* T_{p_i} *be the updated execution trace (as defined above). If compensation for actions* $a_l, ..., a_{n-1}$ *is adequate, the default BPEL compensation order is observed at runtime, the user acts as suggested by the plan in the case of external choices, and the suggested <flow> activity interleavings are executed, then the updated execution trace* T_{p_i} *is the result of compensating actions* $a_l a_{l+1} ... a_{n-1}$*, leaving the application in state* s_l*, then executing* p_m*, leaving the application in state* s_m*, and finally executing* p_i*, leaving the application in a goal state* g*, i.e.,* $\mathsf{T}_{p_m} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{l-1}} s_l \xrightarrow{b_0} ... \xrightarrow{b_{m-1}} s_m \xrightarrow{c_0} ... \xrightarrow{c_{i-1}} g$*, where* $s_l \in \mathsf{C}(\mathsf{B})$ *and* $(s_m, \langle c_0, ..., c_{i-1} \rangle) \in \mathsf{G}^{\mathsf{s}}_{\mathsf{i}}(\mathsf{B}, \mathsf{A})$.

This proposition follows from Definitions 4.1 and 6.2, as well as the fact that we expect both the user and the BPEL engine to execute the suggested actions. Of course, ensuring that a particular execution of the <flow> is chosen is difficult – and in fact, often unnecessary. Our approach, described in Section 3.2, of using events in the <flow> to make the corresponding monitors more permissive, allows the plan to execute successfully even when the BPEL engine produces events out of order.

Definition 7.2 (Successful Mixed Recovery Plan) *Let* T *be an execution trace,* $p = (p_l, p_m, p_i)$ *be a mixed recovery plan and* T_{p_i} *be the updated execution trace (as defined above). A mixed recovery plan* p *is* successful *on an execution trace* T *iff the execution of the updated execution trace* T_{p_i} *on the mixed monitor* A *(the monitor that detected the violation) leaves* A *in a state whose color is green.*

We now discuss the worst case analysis. SAT-based planning is an NP-hard problem [28]. However, due to advances in the SAT community, checking satisfiability has a good average case performance, allowing the solution of problem instances involving tens of thousands of variables and millions of constraints.

As with violations of safety properties, the maximum number of plans we compute for mixed violations is controlled by the user either directly or indirectly, by controlling the maximum plan length. In the latter case, the maximum number of plans is also indirectly determined by the number of change states encountered along the trace and the number of source states in the i-step goal traces reachable from these change states. If n_c change states and n_g source states are reachable from the error state within k steps, then the maximum number of plans of at most length k is $n_c \times n_g$. We check the satisfiability of an increasingly larger SAT instance in order to compute each new plan, since we add a set of constraints for each plan found. This process continues until all plans of length k are found, or the maximum number of plans is reached. In the average case, we expect that the maximum number of

plans will be much smaller than the number of application change states and source states of i-step goal traces. As in the case of safety property violations, we expect that the application developer will limit the maximum number of plans to a small number (≤ 5) to avoid overwhelming users with a large number of plans. The maximum plan length should also depend on how far apart the application goals are, since we want to ensure that at least one goal is reachable from each possible error state.

7.5 Controlling Unnecessary Compensations

Plans p_1, p_2 and p_3 (see Fig. 13a) 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 our approach first goes back to a change state and then goes forward to a goal. To circumvent this problem, we implemented *plan-loop filtering* which can remove the loop that goes from an application state to a change state and then back. Plan-loop filtering turns plans $p_1 - p_3$ into $p'_1 - p'_3$, respectively (see Fig. 13c). This is done semi-automatically: once plan-loops are identified, they are presented to the user, and, if the user requests it, the unnecessary compensations are removed, making the plans more usable but not affecting the overall number of computed plans.

This step cannot be done automatically because we do not know a priori how the output of non-idempotent service calls affects the rest of the execution. If, for example, the execution of cf failed because it received an invalid booking number from bf, then we must cancel the existing flight and book a new one, since cf would keep on failing otherwise.

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 $\mathrm{L^2TS}\ \mathsf{L_C}(\mathsf{TAS})$, shown in Fig. 10a, the plan may include booking a flight and then cancelling it several times (i.e., going between states 7 and 8 of $\mathsf{L_C}(\mathtt{TAS})$). Clearly, such situations should be avoided.

To circumvent such a problem, we implemented *compensation-loop filtering*. The idea is based on the notion that desired plans "go back" only once, and then start going forward. Thus, a plan where a compensatory action follows some non-compensatory one can be removed. Our compensation-loop filtering is implemented as a simple script. Instead, we could have encoded the negation of the above condition as a SAT formula, conjoining it with $\mathsf{R_0}(\mathsf{f_P})$, so plans with compensation loops are not generated at all. Since this encoding would mean propositionalizing two universal quantifiers (ranging over all compensatory and all non-compensatory actions in the system), it might make the SAT computation less efficient. We plan to experiment with it in the future.

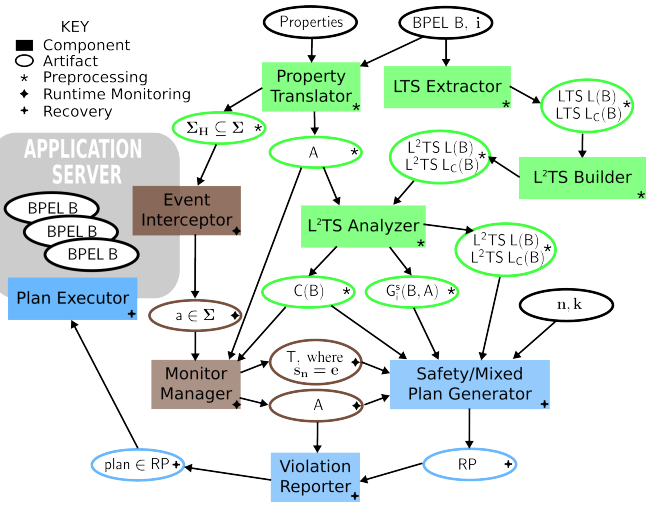


Fig. 14 Architecture of the framework.

8 Tool Support

In this section, we describe the implementation of the monitoring and recovery framework described in this paper. Our tool is called RuMoR, which stands for **RU**ntime **MO**nitoring and **R**ecovery. We have implemented RuMoR on top of the IBM WebSphere product suite, using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts. It 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. We discuss the architecture of our tool in this section, implementation details are available in [38]. When runtime violations are discovered, RuMoR automatically proposes recovery plans which users can then select for execution.

We show the architecture of our framework in Fig. 14. In this diagram, rectangles are components of our framework, and ovals are artifacts. We have also grouped the components and artifacts by phase: preprocessing – green, with a ✶ symbol; runtime monitoring – brown, with a ◆ symbol; and recovery – blue, with a ✚ symbol. Artifacts with a thick border are the initial inputs to our framework. The preprocessing and runtime monitoring phases of our framework are the same for both safety and mixed properties, but different components are required for generating plans from the two types of properties.

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 (this process is out of scope of this paper and is described in [38]). The *LTS Extractor* (LE) component extracts an LTS model from the BPEL pro-

gram and augments it with compensation links (see Section 4.1). The *L^2TS Builder* (LB) component collects predicate information and service contracts to augment the resulting LTS with state information, resulting in an L^2TS (see Section 4.2). The *L^2TS Analyzer* (LA) uses the SMT solver MathSAT to check which compensation links are adequate (see Section 4.2), and computes goal links and change states using the techniques described in Section 4.3.

During the execution of the application, the *Event Interceptor* (EI) component intercepts application events and sends them to the *Monitor Manager* (MM) for analysis (see Section 5 for details). MM stores the intercepted events for recovery, updates values of state variables of the application, checks pre- and post-conditions of service partners, and updates the state of each active monitor. This process continues until an error has been found, which activates the recovery state, or all partners terminate.

During the recovery phase, artifacts from both the preprocessing and the runtime monitoring phases are used to generate recovery plans. In the case of safety properties, the *Safety Plan Generator* generates recovery plans that can only compensate executed activities (see Section 6). For mixed properties, plans can compensate executed activities and execute new activities. In this case, the *Mixed Plan Generator* (MPG) first generates the corresponding planning problem and then modifies it in order to generate as many plans as required (see Section 7). The MPG also uses a simple script to identify plan-loops, as well as remove compensation loops from the computed plans (see Section 7.5).

In addition, we have implemented two heuristics aimed to reduce the number (and improve the quality) of generated plans. These are: 1) remove plans that require going through unnecessary change states, where re-executing the partner call cannot affect the (negative) outcome of the trace, and 2) remove plans that attempt to satisfy a mixed property at the expense of violating some safety properties. The details of these optimizations are available in [41].

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. These recovery plans may include plan-loops, depending on the user's choice. For example, the plans in the snippet in Fig. 15a generated for a violation of P_1 have plan-loops. Developers must include this snippet in the default error page, so that the computed recovery plans can be shown when an error is detected. Fig. 15b shows the (simplified) source code of such an error reporting page, where the bolded line has the instruction to include the snippet. After the recovery plans have been computed, the snippet is displayed as part of the application, and the user must pick a plan to continue execution. Fig. 15c shows a screen shot of `error.jsp` after recovery plans for P_1 have been computed. The PE receives as input a BPEL plan. Statically, we add a <collaboration> scope to each process before execution, and the BPEL plan chosen by the user is set as the logic of this scope. As the chosen plan gets executed, MM updates the states of the

snippet.jsp

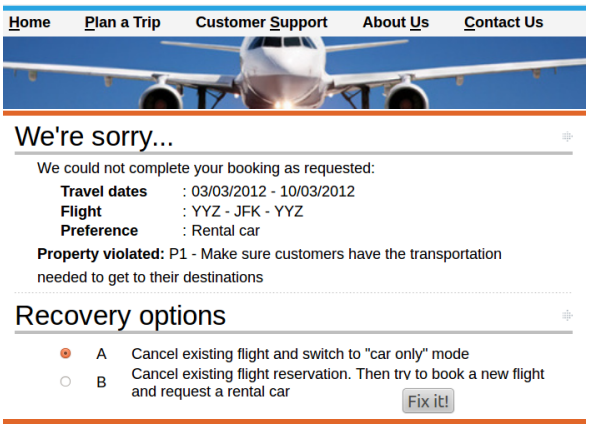
```
<p><strong>Property violated:</strong> P1 - Make sure customers have the transportation needed to get
            to their destinations.</p>

<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)



(c)

Fig. 15 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.

active monitors; if none are violated, the framework switches back to runtime monitoring.

Table 1 Results of adding filtering and the adequate compensation check to our framework.

Trace	k	$\lvert C_T(B)\rvert$	Baseline			Adequate Compensation (AC)			AC + Optimizations [41]		
			Plans	Time (s)	Plans (filtered)	Plans	Time (s)	Plans (filtered)	Plans	Time (s)	Plans (filtered)
t_1	6	4	1	0.01	1	1	0.15	1	1	0.27	1
	8	4	5	0.13	5	5	0.27	5	5	0.41	5
	13	4	13	0.27	13	13	0.41	13	13	0.54	13
t_2	6	2	2	0.01	2	2	0.15	2	2	0.27	2
	8	4	4	0.01	4	4	0.15	4	4	0.27	4
	13	6	6	0.01	6	6	0.15	6	6	0.27	6
t_3	5	2	2	0.01	2	2	0.55	2	0	0.81	0
	10	5	5	0.02	5	5	0.56	5	2	0.82	2
	15	8	8	0.02	8	8	0.56	8	5	0.82	5
	20	12	12	0.02	12	12	0.56	12	8	0.82	8
	25	13	13	0.02	13	12	0.56	12	8	0.82	8
	30	13	13	0.02	13	12	0.56	12	8	0.82	8
t_4	5	4	0	0.01	0	0	0.55	0	0	0.82	0
	10	7	2	0.14	2	2	0.68	2	2	1.13	2
	15	10	8	1.37	8	8	1.91	8	5	3.73	5
	20	10	18	4.72	11	13	3.95	11	8	6.84	8
	25	10	60	29.16	32	42	20.95	24	23	29.62	18
	30	10	68	61.34	38	42	38.43	24	23	47.08	18

9 Experience

We have applied our framework on various case studies with full results available in [38]. In this section, we focus on assessing the cost and effectiveness of the adequate compensation check introduced in Section 4.2. Since this check may remove compensation transitions from a L^2TS, we expect that including this step in our framework does not generate plans which are infeasible at runtime and thus result in fewer plans altogether. We are also interested in the overhead that this analysis adds in practice.

To study this, we compare the results of applying three different configurations of our framework to two of our case studies – `TAS` and the Travel Booking system (`TBS`). `TBS` was described in [38]. Its L^2TS model has 52 states (35 identified as change states) and 67 transitions, and $|\Sigma| = 33$. We ran these two case studies using three different framework configurations: 1) **Baseline** – 1-step goal traces, without the adequate compensation check and without extra optimizations, 2) **Adequate Compensation** (AC) – the basic framework with only the adequate compensation check, and 3) **AC + Optimizations** – includes both the adequate compensation check and extra optimizations, described in Section 8 and in [41].

Table 1 summarizes our results. Traces t_1 and t_2 are those described in Section 5; traces t_3 and t_4 correspond to the `TBS` system and are described in [38][3]. Traces t_1 and t_4 violate mixed properties, while t_2 and t_3 violate safety properties. In the case of safety property violations, no SAT instances are generated, and the running time for the plan generation is trivial. We report the number of plans computed for each framework configuration as well as the time it took to compute them. For `TAS`, we generated plans starting with length $k = 6$ and going to $k = 13$, while we started with length $k = 5$ and went to $k = 30$ for `TBS`. For example, at $k = 25$, a total of 13 plans were

[3] In [38], t_3 and t_4 are called $t_1^{\texttt{TBS}}$ and $t_2^{\texttt{TBS}}$, respectively.

initially generated for t_3; this total was reduced to 12 plans after we checked for adequate compensation, and if we also considered other optimizations to our framework, this total was further reduced to 8. In this case, our filtering techniques did not help reduce the number of plans. However, it was effective for t_4.

We have included the data for the `TAS` traces for completeness: since all of its compensation transitions are adequate and all its change states are relevant, neither of the new framework configurations produce a reduction in the number of computed plans. On the other hand, the adequate compensation check indicated that in `TBS`, the service used to hold a hotel reservation (`hh`) was not adequately compensated. Since `hh` appears in both t_3 and t_4, the **Adequate Compensation** framework configuration computes fewer recovery plans for both traces (in both cases, there are no new plans after $k = 25$). Also, since there is no compensation transition for `hh`, the third framework configuration results in a further reduction in the number of plans, since fewer change states are reachable from the error state.

The total time reported for both the second and third framework configurations includes the time it took to do the required static analyses. In practice, most of these analyses are only carried out once per L^2TS, so most of the extra analysis cost should be amortized as we increase k. For example, in the case of t_4, we see that after $k = 15$, the total time required by the second framework configuration is lower than the time required by the basic framework – even though the adequate compensation check is initially expensive, it effectively prevented the computation of about a third of the initially generated plans. The analyses included in the third framework configuration are a bit more expensive, but we see an even bigger reduction of number of plans for large values of k.

Our initial experience shows that the adequate compensation check is both feasible and effective, resulting in a relatively small number of relevant plans. This is important, since the plans are presented to the user who must pick one in order to continue execution.

We also studied the impact of varying i, the parameter in i-step goal traces used for computing plans (see Section 4.3.1). Specifically, we generated recovery plans of length 13 for trace t_1 of `TAS`, starting with $i = 1$ and going up to $i = 5$. For each goal trace, we measured the number of plans generated and the time it took to do so (using the **Baseline** framework configuration with filtering). Table 2 summarizes our results. For example, there were five 4-step goal traces (gt_{12}-gt_{16}). Using gt_{13}, we computed two plans for t_1, taking 0.548 s, and overall, we computed 11 plans of length 13 (e.g., using plans of length 9 followed by a four-event sequence), taking 1.194 s. Note that the time we report in the first row of Table 2 is larger than that for $k = 13$ in Table 1 since it also includes filtering.

Fig. 16 summarizes the results in the table. As expected, as i increases, the number of generated plans decreases since there are fewer valid "suffixes" for mixed recovery plans. However, as i increases, the time to compute mixed plans increases as well, due to the increased preprocessing, planning and fil-

Table 2 Results of varying i when generating recovery plans of length $k = 13$ for trace t_1.

i	Goal trace	Plans (filtered)	Time (s)	Total plans	Total time (s)
1	gt_1	13	0.364	13	0.364
2	gt_2	2	0.194		
	gt_3	4	0.254		
	gt_4	1	0.264		
	gt_5	2	0.260		
	gt_6	2	0.222	11	1.194
3	gt_7	2	0.248		
	gt_8	2	0.269		
	gt_9	1	0.277		
	gt_{10}	4	0.306		
	gt_{11}	2	0.318	11	1.418
4	gt_{12}	1	0.112		
	gt_{13}	2	0.548		
	gt_{14}	1	0.560		
	gt_{15}	3	0.352		
	gt_{16}	2	0.300	9	1.872
5	gt_{17}	2	0.564		
	gt_{18}	1	0.560		
	gt_{19}	2	0.218		
	gt_{20}	2	0.242		
	gt_{21}	2	0.248		
	gt_{22}	0	0.236	9	2.068

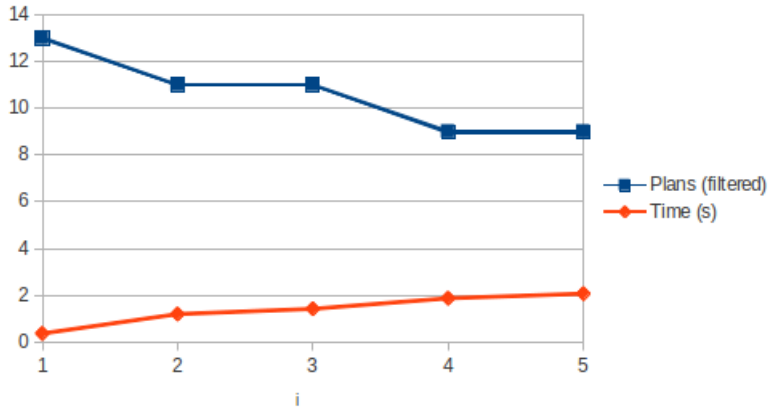


Fig. 16 Summary of the data in Table 2.

tering. Ideally, the value of i should be determined experimentally for each application. We plan to continue studying this issue in future case studies.

10 Related Work

In this section, we survey current work on error recovery and adaptation and compare it with our approach.

Existing infrastructures for web services, e.g., the BPEL engine [35], include mechanisms for fault definition, for specifying 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.

Since these standard error recovery mechanisms are statically defined, one relatively simple way of improving error recovery is to statically analyze the application and suggest changes that improve the application's fault tolerance. In [17], Dobson defined a library of fault tolerance patterns, which are used to transform the original BPEL process into a fault-tolerant one at compile time. This is done by adding redundant behaviour to the application, but this may result in a significantly bigger, and slower, program.

The work proposed by Baresi et al. [5] also enables recovery through the standard error recovery mechanisms, but by attaching BPEL exception handlers to properties that are checked at runtime. The advantage to this approach is that the new exceptions are triggered by the violation of high-level properties which can help debugging. If such an exception handler is not provided, execution terminates when a property is violated.

Several works have suggested self-healing mechanisms for web service applications. The Dynamo framework [4] uses *annotation rules* in BPEL in order to allow recovery once a fault has been detected. Such rules need to be statically defined by the developers before the system can function. Fugini and Mussi [22] propose a framework for self-healing web services, where all possible faults and their repair actions are pre-defined in a special registry. This approach relies on being able to identify and create recovery from all available faults.

The framework proposed by Carzaniga et al. [12] is the closest to ours in terms of error recovery, as the authors also attempt to automatically compute recovery plans for runtime errors. This framework exploits redundancy in web applications to find workarounds when errors occur, assuming that the application is given as a finite-state machine (without compensation), with an identified error state as well as the "fallback" state to which the application should return (one per error).

A workaround is a path in the finite-state machine starting at the error state and ending at a fallback state. Before computing all the possible workarounds for an error, the current error transitions are removed from the application model. In some cases, this makes the fallback state unreachable, and additional system events must be inserted to make the model connected again. The approach then exhaustively generates all possible workarounds, prioritizing them solely by length. Fallback states are manually identified by the developer, whereas our method attempts to compute an approximation of these states using user-specified properties of the system (goal transitions). Our framework also attempts to filter out unusable recovery plans (those that do not include change states) and ranks the remaining ones.

The work in [11] addresses some of the above limitations by generating workarounds using general code-rewriting rules specified by the API developers instead of the full model of the API behavior. The rules specify which

operations have no side-effects, which reverse effects of others, and which can serve as alternatives to others. Upon discovery of an API error, the framework generates workarounds that change the structure of the webpage being rendered, until one is accepted by the user. Our use of compensations and idempotent service calls is similar to rule-based specification of operations, but our framework does not use alternatives. Furthermore, the work in [11] does filtering based on different criteria from the ones presented here.

AI planning has been successfully used to automatically generate web service compositions (WSC) [29, 33, 30, 44]. Replanning when an error occurs can be expensive, so Yan et al. [47] propose a heuristic for repairing planning graphs, using service pre- and post-conditions to add new transitions to the existing planning graph. This heuristic stops when a new path to a goal state is found. This approach relies on the existence of detailed service contracts, does not consider the possibility of compensation, and limits repairs to service compositions.

Baresi and Pasquale [7] work at the requirements level, allowing the definition of adaptive goals and strategies which are used to provide support for adaptation at runtime. The adaptation goals are specified using a goal model, where LTL formulas and adaptation strategies are associated with individual goals. If a goal's LTL formula is violated at runtime, the associated adaptation strategy is triggered. These strategies include adding/removing goals from the goal model, modifing the LTL formula associated to a goal, adding/removing/modifying operations (which changes future goal operationalization). Like Yan et al. [47], this approach relies on the availability of detailed service contracts, and also does not explicitly consider compensation.

Charfi et al. [13] propose a plug-in architecture for self-adaptation, based on the aspect-oriented programming paradigm. The two types of aspects they allow to define are monitoring, used to detect problems, and adaptation, used to hot deploy (i.e., apply as the application runs) self-adaptation logic. This work extends the AO4BPEL [14] language, allowing dynamic generation, activation and deactivation of these aspects at runtime. It is the developer's responsibility to write the self-adaptation logic of the adaptation aspects, adding compensation manually if needed.

Unlike the above approaches, our framework does not explicitly check whether services comply with their specified contracts – we use these contracts to help establish our correctness guarantees. Our work does not require the definition of static recovery strategies; furthermore, plans are generated using the already specified behavior, i.e., we do not add transitions to our formal model. We also maintain an explicit relationship between actions and their compensations (unlike the AI-based systems) and take full advantage of the predefined recovery mechanism – compensation. Yet we enrich the standard compensation mechanism since runtime errors are defined w.r.t. high-level partner interaction properties, and we can prove that the suggested recovery from safety and bounded liveness property violations leaves the system in a desired state. Finally, some of the above approaches [47, 7] are complementary

to ours and can be applied in order to improve performance and/or quality of our plans.

11 Summary and Future Work

In this paper, we described our framework for runtime monitoring and recovery of web service conversations. The monitoring portion is non-intrusive, running in parallel with the monitored system and intercepting interaction events during runtime. It does not require any code instrumentation and enables reasoning about partners expressed in different languages. We have then used BPEL's compensation mechanism to define and implement an online system for suggesting, ranking and executing recovery plans. Furthermore, through collecting state information, we are able to check that partners fulfill their posted service contracts and that compensation is adequately defined. Finally, we are able to prove correctness of the generated plans. Our experience has shown that this approach computes a small number of highly relevant plans, doing so quickly and effectively, and in this paper we show that our framework remains feasible even with the additional static analysis that checks adequacy of compensation.

In what follows, we discuss limitations of our approach and venues for future research. A 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. 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. In such examples, we may not be able to prove adequacy of compensation because of insufficient state information. We intend to experiment with program analysis techniques such as abstraction refinement [3, 23] to try to derive additional information that should be kept in the state of our L^2TSs. To the best of our knowledge, such techniques have not been applied to web services.

In our framework, recovery is initiated only after an error has been discovered. An alternative approach might be to use the available properties to generate plans *a priori* and then re-plan if the user decides to deviate from the prescribed course of action or something in the environment changes, in the GPS-like plan recalculation. This should reduce the need for compensation and rework.

Finally, our work so far has assumed that all partners operate within the same process server and thus a centralized monitoring and recovery is a viable option. In practice, most web services are distributed, requiring distributed monitoring and recovery. Techniques for turning a centralized monitor into a set of distributed ones, running in different process servers, have been investigated by the DESERT project [27], but we leave the problem of distributed plan generation and execution for future work.

Acknowledgements Several people contributed to the ideas presented in this paper: Shiva Nejati, Yuan Gan and Jonathan Amir were involved in various aspects of defining and building the runtime monitoring framework. Various people at IBM CAS Toronto, specifically, Bill O'Farrell, Julie Watermark, Elena Litani and Leho Nigel, have been working with us over the years, and Bill is responsible for shaping this project into its present form, including the suggestion that we work on recovery from runtime failure. We are also grateful to members of the IFIP Working Group 2.9 for their helpful feedback on this work. This research has been funded by NSERC, IBM Toronto, MITACS Accelerate, UTFSM DGIP and by the Ontario Post-Doctoral Fellowship program. We also thank reviewers of this special issue for many helpful comments.

References

1. van der Aalst, W.M.P., Weske, M.: Case Handling: a New Paradigm for Business Process Support. Data Knowledge Engineering **53**(2), 129–162 (2005)
2. Alur, R., Henzinger, T.A.: Finitary Fairness. ACM Trans. Program. Lang. Syst. **20**, 1171–1194 (1998)
3. Ball, T., Rajamani, S.: The SLAM Toolkit. In: Proceedings of 13th International Conference on Computer-Aided Verification (CAV'01), *LNCS*, vol. 2102, pp. 260–264 (2001)
4. Baresi, L., Guinea, S.: Dynamo and Self-Healing BPEL Compositions (research demonstration). In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), pp. 69–70. IEEE Computer Society (2007). Companion Volume
5. Baresi, L., Guinea, S.: Self-Supervising BPEL Processes. IEEE Transactions on Software Engineering **37**(2), 247–263 (2011)
6. Baresi, L., Guinea, S., Plebani, P.: WS-Policy for Service Monitoring. In: 6th VLDB International Workshop on Technologies for E-Services, *LNCS*, vol. 3811, pp. 72–83. Springer (2006)
7. Baresi, L., Pasquale, L.: Adaptive Goals for Self-Adaptive Service Compositions. In: Proceedings of the 2010 IEEE International Conference on Web Services (ICWS '10), pp. 353–360. IEEE Computer Society, Washington, DC, USA (2010)
8. Berre, D.L., Parrain, A.: SAT4J. `http://www.sat4j.org/` (2012)
9. Biancullia, D., Ghezzi, C.: DynamoAOP - User Manual. `http://plastic.isti.cnr.it/download/tools/dynamo-aop/dynamoaop-manual.p%df` (2012)
10. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. **7**, 212–232 (2005)
11. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic Workarounds for Web Applications. In: Proceedings of Eighteenth International Symposium on the Foundations of Software Engineering (FSE'10), pp. 237–246 (2010)
12. Carzaniga, A., Gorla, A., Pezzè, M.: Healing Web Applications through Automatic Workarounds. Int. J. Softw. Tools Technol. Transf. **10**(6), 493–502 (2008)
13. Charfi, A., Dinkelaker, T., Mezini, M.: A Plug-in Architecture for Self-Adaptive Web Service Compositions. In: Proceedings of the 2009 IEEE International Conference on Web Services (ICWS '09), pp. 35–42. IEEE Computer Society, Washington, DC, USA (2009)
14. Charfi, A., Mezini, M.: AO4BPEL: An Aspect-oriented Extension to BPEL. World Wide Web **10**, 309–344 (2007)
15. Cheng, B.H.C., de Lemos, R., Garlan, D., Giese, H., Litoiu, M., Magee, J., Müller, H.A., Taylor, R.: SEAMS 2009: Software Engineering for Adaptive and Self-Managing Systems. In: 31st International Conference on Software Engineering, (ICSE'09), Companion Volume, pp. 463–464 (2009)
16. DISI-UniTN/FBK-IRST: The MathSAT 5 SMT Solver. `http://mathsat.fbk.eu` (2012)
17. Dobson, G.: Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06), pp. 126–133 (2006)
18. Fikes, R., Nilsson, N.J.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Journal of Artificial Intelligence **2**(3/4), 189–208 (1971)

19. Foster, H.: A Rigorous Approach to Engineering Web Service Compositions. Ph.D. thesis, Imperial College (2006)
20. Foster, H., Emmerich, W., Kramer, J., Magee, J., Rosenblum, D., Uchitel, S.: Model Checking Service Compositions under Resource Constraints. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07), pp. 225–234. ACM (2007)
21. Fu, X., Bultan, T., Su, J.: Analysis of Interacting BPEL Web Services. In: Proceedings of the 13th International Conference on World Wide Web (WWW'04), pp. 621–630 (2004)
22. Fugini, M.G., Mussi, E.: Recovery of Faulty Web Applications through Service Discovery. In: Proceedings of the 1st SMR-VLDB Workshop, Matchmaking and Approximate Semantic-based Retrieval: Issues and Perspectives, 32nd International Conference on Very Large Databases, pp. 67–80 (2006)
23. Gurfinkel, A., Wei, O., Chechik, M.: YASM: A Software Model-Checker for Verification and Refutation. In: Proceedings of 18th International Conference on Computer-Aided Verification (CAV'06), *LNCS*, vol. 170-174, p. 4144. Springer, Seattle, WA (2006)
24. Heckel, R., Lohmann, M.: Towards Contract-based Testing of Web Services. Electronic Notes in Theoretical Compututer Science **116**, 145–156 (2005)
25. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri Nets. In: Proceedings of the 3rd International Conference on Business Process Management (BPM'05), *LNCS*, vol. 3649, pp. 220–235 (2005)
26. IBM: WebSphere Integration Developer. `http://www-01.ibm.com/software/integration/wid/` (2012)
27. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: Synthesis of Correct and Distributed Adaptors for Component-Based Systems: an Automatic Approach. In: Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05), pp. 405–409. ACM (2005)
28. Kautz, H.A., Selman, B.: Unifying SAT-based and Graph-based Planning. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99), pp. 318–325 (1999)
29. McDermott, D.V.: Estimated-Regression Planning for Interactions with Web Services. In: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS '02), pp. 204–211. AAAI (2002)
30. McIlraith, S.A., Son, T.C.: Adapting Golog for Composition of Semantic Web Services. In: Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR '02), pp. 482–496. Morgan Kaufmann (2002)
31. Meyer, B.: Applying "design by contract". Computer **25**, 40–51 (1992)
32. Milner, R.: Communication and Concurrency. Prentice-Hall, New York (1989)
33. Narayanan, S., McIlraith, S.A.: Simulation, Verification and Automated Composition of Web Services. In: Proceedings of the 11th International Conference on World Wide Web (WWW '02), pp. 77–88. ACM (2002)
34. Nicola, R.D., Vaandrager, F.: Three Logics for Branching Bisimulation. Journal of the ACM (JACM) **42**(2), 458–487 (1995)
35. OASIS: Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html` (2012)
36. Oracle: Welcome to NetBeans. `http://netbeans.org` (2012)
37. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal Semantics and Analysis of Control Flow in WS-BPEL. Science of Computer Programming **67**(2-3), 162–198 (2007)
38. Simmonds, J.: Dynamic Analysis of Web Services. Ph.D. thesis, University of Toronto, Toronto (2011)
39. Simmonds, J., Ben-David, S., Chechik, M.: Guided Recovery for Web Service Applications. In: Proceedings of Eighteenth International Symposium on the Foundations of Software Engineering (FSE'10), pp. 247–256 (2010)
40. Simmonds, J., Ben-David, S., Chechik, M.: Monitoring and Recovery of Web Service Applications. In: J.W. Ng, M. Chignell, J.R. Cordy (eds.) Smart Internet, Lecture Notes in Computer Science, pp. 250–288. Springer (2010)

41. Simmonds, J., Ben-David, S., Chechik, M.: Optimizing Computation of Recovery Plans for BPEL Applications. In: Proceedings of 2010 Workshop on Testing, Analysis and Verification of Web Software (TAV-WEB'10), pp. 3–14 (2010)
42. Simmonds, J., Chechik, M.: RuMoR: Monitoring and Recovery of BPEL Applications. In: Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), pp. 345–346 (2010)
43. Simmonds, J., Gan, Y., Chechik, M., Nejati, S., O'Farrell, B., Litani, E., Waterhouse, J.: Runtime Monitoring of Web Service Conversations. IEEE Transactions on Service Computing **2**(3), 223–244 (2009)
44. Traverso, P., Pistore, M.: Automated Composition of Semantic Web Services into Executable Processes. In: Proceedings of the International Semantic Web Conference (ISWC '04), pp. 380–394 (2004)
45. W3C: Semantic Web. `http://www.w3.org/standards/semanticweb/` (2012)
46. W3C: Web Services Description Language (WSDL). `http://www.w3.org/TR/wsdl` (2012)
47. Yan, Y., Poizat, P., Zhao, L.: Self-Adaptive Service Composition Through Graphplan Repair. In: IEEE International Conference on Web Services (ICWS '10), pp. 624–627 (2010)

A Appendix

This appendix gives additional background information on BPEL's activities and variables. We also provide an overview of the WSCoL subset used in this article.

A.1 BPEL activities and variables

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. Conditional activities are used to define the control flow of the application: <while>, <if> and <pick>. The <while> and <if> activities model internal choice, as conditions are expressions over process variables. The <pick> activity is used to model external choice: the application waits for the first occurrence of one of several possible events (specified using <onMessage>), executing the activities associated to the corresponding event. The <pick> activity completes once these activities are completed.

The structural activities <sequence> and <flow> are used to specify sequential and parallel composition of the enclosed activities, respectively. <scope> is used to define nested activities. In IBM WebSphere Integration Developer v7, developers can also add <collaboration> scopes, inspired by the work on dynamic workflows [1], which can be used to alter the application logic at runtime.

The <assign> activity is used to update the values of variables. Assignment activities consist of multiple <copy> rules, each with their own <from> and <to> parts, representing the source and destination data, respectively. <copy> rules can include data modification expressions, e.g., <copy> <from> `$i + 1` </from> <to> `$i` </to> </copy> increments the value of `$i` by one, but can also be simple assignments.

BPEL has both global and local variables. Global variables are available throughout the process, while local variables are only available within the <scope> in which they are defined. Variable types can be simple or complex XSD (XML Schema Definition) types, schema elements, or WSDL message types, declared using the keywords `type`, `element` and `messageType`, respectively. Multiple schema elements can refer to the same complex XSD type.

Fig. 17a shows the definition of the simple and complex XSD types used by the `TAS` application. For example, `fromDate` and `sourceCity` are of the simple XSD types `xsd : int` and `xsd : string`, respectively. On the other hand, `tTripData` is a complex XSD type and includes a reference to every simple type declared in the same figure. These type definitions

```
<!-- definition of simple elements -->
<element name="fromDate" type="xsd:int"/>
<element name="toDate" type="xsd:int"/>
<element name="sourceCity" type="xsd:string"/>
<element name="destCity" type="xsd:string"/>
<element name="flightBookingNo" type="xsd:int"/>
<element name="flightNo" type="xsd:string"/>
<element name="flightStatus" type="xsd:string"/>
<element name="carBookingNo" type="xsd:int"/>
<element name="carStatus" type="xsd:string"/>
<element name="carType" type="xsd:string"/>
<element name="expensive" type="xsd:string"/>

<!-- definition of complex elements -->
<element name="tTripData">
   <complexType>
      <sequence>
         <element ref="fromDate"/>
         <element ref="toDate"/>
         <element ref="sourceCity"/>
         ...
         <element ref="carType"/>
         <element ref="expensive"/>
      </sequence>
   </complexType>
</element>
```

(a)

```
<message name="in_bc" xmlns:x="tas.xsd"/>
   <part name="city" element="x:destCity"/>
   <part name="fromDate" element="x:fromDate"/>
   <part name="toDate" element="x:toDate"/>
</message>

<message name="out_bc" xmlns:x="tas.xsd"/>
   <part name="bookingNo" element=
      "x:carBookingNo"/>
   <part name="status" element="x:carStatus"/>
   <part name="type" element="x:carType"/>
</message>
```

(b)

```
<variables>
   <variable name="tripData" element=
      "x:tTripData"/>
   <variable name="inputBookCar" messageType=
      "x:in_bc"/>
   <variable name="outputBookCar" messageType=
      "x:out_bc"/>
      ...
</variables>
```

(c)

Fig. 17 (a) Part of the schema file `tas.xsd`, showing the definition of the `TAS` simple and complex XSD types; (b) part of the WSDL file `tas.wsdl`, showing the definition of the WSDL message types for the `bc` service; and (c) the declaration of various BPEL global variables.

are then available from the BPEL process by importing the XSD file in which they are defined.

WSDL message types define abstract messages that can serve as the input or output of an operation, and are thus defined in the corresponding WSDL file. WSDL messages consist of one or more <part> elements, where each <part> is associated with an existing XSD element. For example, as shown in Fig. 17b (copy of Fig. 5a), the message type `in_bc`, which is the input message for the `bc` service, consists of three parts: `city`, `fromDate` and `toDate`.

Once the necessary variable types are defined and imported, we can declare the required BPEL variables. Fig. 17c (copy of Fig. 5b) shows the definition of three variables of the `TAS` system: `tripData`, `inputBookCar` and `outputBookCar`. The first variable is used to maintain the state of the application, while the second two are the input and output variables of the

```
          rules ::= sub_rule ((==>|<==|<==>) sub_rule)*
       sub_rule ::= and_expr (||  and_expr)*
       and_expr ::= equals_expr (&& equals_expr)*
    equals_expr ::= relational_expr ((==|!=) relational_expr)?
relational_expr ::= op_expr ((>|>=|<|<=) op_expr)?
        op_expr ::= basic_expr ((+ | − | * | / | %) basic_expr)*
     basic_expr ::= variable | quant_expr | true | false | number | string_value
     quant_expr ::= ( quantifier $ identifier in variable ; sub_rule )
     quantifier ::= \forall | \exists | \avg | \sum | \product | \min | \max
       variable ::= ((ivar | evar)) | (ivar | evar)
           ivar ::= $ identifier xpath_expr?
           evar ::= return_type (string_value, string_value, string_value, xpath_expr)
    return_type ::= \returnInt | \returnBool | \returnString
   string_value ::= substring_value (+ substring_value)*
substring_value ::= identifier | literal | variable
        literal ::= LIT
         number ::= NUM
```

Fig. 18 Grammar of the subset of the WSCoL language used in this work.

bc service. We have defined similar input and output variables for the rest of the services used by the TAS application.

A.2 WSCoL

The grammar of the WSCoL subset used in this article is given in Fig. 18, where identifier represents a valid BPEL variable identifier and xpath_expr represents a valid XPath expression. *LIT* and *NUM* are sets of BPEL literals and numbers, respectively, while **true**, **false**, **\forall**, **\exists**, **\avg**, **\sum**, **\product**, **\min**, **\max**, **\returnInt**, **\returnBool**, and **\returnString** are terminal symbols. The full WSCoL grammar is available in [9].

In WSCoL, we can write predicates about the state of a BPEL process by accessing its internal variables using XPath expressions. For example, the expression $tripData/carBookingNo retrieves the car booking number for the current trip, where tripData is the associated BPEL identifier and /carBookingNo is the XPath expression pointing to the XSD type to be queried.

External data is accessed by invoking other web services. For example, the expression

```
\returnInt('WSDL','subtractDates',
  ($tripData/fromDate, $tripData/toDate),
  '//parameters/subtractDatesResult/days')
```

returns the difference between the trip start and end dates as an integer. This is done by invoking the subtractDates service with parameters $tripData/fromDate and $tripData/toDate, corresponding to the trip start and end dates, respectively. The difference is stored in the variable //parameters/subtractDatesResult/days, and the subtractDates service definition is available in a preconfigured WSDL file.

The typical Boolean, relational and mathematical operators, as well as some pre-defined functions, are used to build more complex expressions. For example, the assertion

```
($tripData/carBookingNo) >= 1000
  && $tripData/fromDate < $tripData/toDate
```

checks that the car booking number is at least four digits long and that the trip end date occurs after the trip start date.

Predicates on sets of variables are defined through the use of general quantifiers of the form (`Q $V in R; C`), where `Q` is one of the quantifiers listed in Fig. 18, `R` is an XPath expression that represents a set of finite nodes, `$V` is the identifier used to access each node, and `C` is a WSCoL predicate.

For example, the predicate

```
(\forall $L in ($outputGF/flights/flight/);
  ($L/destCity) == ($tripData/destCity))
```

checks that all the flights returned by a getFlights service have the same destination city as the one stored in `$tripData`.