

Monitoring and Recovery of Web Service Applications

Jocelyn Simmonds, Shoham Ben-David, Marsha Chechik

Department of Computer Science, University of Toronto
{jsimmond, shoham, chechik}@cs.toronto.edu

Abstract. For a system of distributed processes, correctness can be ensured by (statically) checking whether their composition satisfies properties of interest. However, web services are distributed processes that dynamically discover properties of other web services. Since the overall system may not be available statically and since each business process is supposed to be relatively simple, we propose to use (on-line) runtime monitoring of conversations between partners as a means of checking behavioural correctness of the entire web service system. Our framework allows application developers to specify behavioural correctness properties. By transforming these properties to finite-state automata, we enable conformance checking of finite execution traces of web services described in BPEL against the specification. Moreover, when violations are discovered at runtime, we automatically propose and rank recovery plans which users of the system can then select for execution. For some of the violations, such plans essentially involve “going back” – compensating the occurred actions until an alternative behaviour of the application is possible. For other violations, such plans include both “going back” and “re-planning” – guiding the application towards a desired behaviour. We report on the implementation and experience with our monitoring and recovery system, and discuss the implications that the move to “smart internet” [39] may have on our approach.

1 Introduction

Recent years have seen an emergence of the field of web services, which use Service-Oriented Architectures (SOA) to dynamically discover and bind to services in order to increase the flexibility of business interactions. Each service consists of *components* and can discover other components using published interfaces. An SOA component can be written in a traditional compiled language such as JavaTM, or in an XML-centric language such as BPEL [40]. An SOA *module* is made up of multiple SOA components which are commonly referred to as web services.

Since each web service is a relatively simple process, analysis can concentrate on the message exchange between partners – their *conversations*. For a classical system of distributed processes, correctness can be ensured by statically checking their composition against properties of interest. The same approach has been taken by several researchers in the context of web services as well, e.g., [2, 17, 19, 20, 29]. While static analysis is very appealing – errors are discovered ahead of time and without the need to exercise the system, this approach has several major limitations:

- Web services are distributed systems, where partners are dynamically discovered and are going on- and off-line as the application runs.

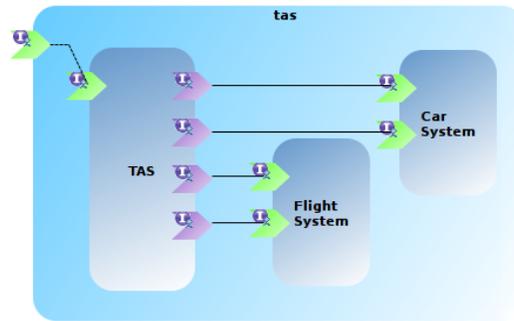


Fig. 1. Assembly diagram describing interactions between the main TAS process and its partners.

- Web services typically communicate via infinite-length channels, so the problem is decidable only under certain conditions [23].
- Web applications usually interact with web services developed by partners. Partners are only required to make web service interfaces public, not the code.
- Realistic web services exchange many types of messages: some synchronous, some asynchronous, and some with acknowledgements and priorities.
- Web services are typically heterogeneous, i.e., each component can be implemented in a different programming language.

Instead, we concentrate on the dynamic analysis via *runtime monitoring*, which tries to ensure the quality of an application through the analysis of runtime events. *Online monitoring* – during the execution of the application – concentrates on monitoring *pre-defined properties*, collecting just those events which are related to the given properties. Moreover, monitoring as the system runs provides a chance to recover from an error once a problem has been detected.

This chapter describes a user-guided runtime monitoring and recovery framework for web services expressed in BPEL. Our motivation was the traditional web services model, where services reside on the server and communicate with other partners or with the user. We discuss the implications that the move towards *smart internet*, described in [39], has on our approach at the end of the paper, in Section 10.

In the “traditional” web service model, properties describe behavior, specifically, interactions between service partners. Such properties are effectively scenarios that the system should exhibit and those that the system should not exhibit. Such desired and forbidden behaviors can come from use-cases, global invariants, simulation, or a variety of other sources. In this chapter, we express behavioural correctness properties using the Specification Pattern System (SPS) [13], converting the high-level patterns into quantified regular expressions (QRE) and then to finite-state automata. We then use the automata to enable conformance checking of finite execution traces and recovery, should a violation be detected.

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 thus books it), or via an air/ground transportation combination, combining the

flight with either a rental car from the airport or a limo. The requirement of the system is to make sure the customer has the transportation needed to get to her destination (this is a desired behavior which we refer to as P_1) while keeping the costs down, i.e., she is not allowed by her company to reserve an expensive flight and a limo (this is a forbidden behavior which we refer to as P_2).

Figure 1 presents an assembly diagram depicting interactions between the main TAS process and its partners – 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). This is depicted in Figure 1 by two sets of connections between TAS and each of the Flight and the Car components. Since the TAS system is a composition of several distributed business processes, its correctness depends on the correctness of its partners and their interactions. For example, the Car system can go down while the user attempts to book ground transportation, thus preventing the entire system from getting the user to her destination.

2 Overview of the Approach

The overview of the approach is given in Figure 2.

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

In our approach, developers supply a BPEL program and a set of behavioral correctness properties (expressed using property patterns) 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.

In the Preprocessing phase, the correctness properties are turned into finite-state automata (monitors), and the BPEL program is turned into a labeled transition system. These are then passed to the Runtime monitoring phase which runs the monitors in parallel with the BPEL application, stopping when one of the monitors is about to enter its error state. The use of high-level properties allows us to detect the violation, and our event interception mechanism allows us to stop the application *right before* the violation occurs and begin the Recovery phase.

In the Recovery phase, we identify and optionally rank 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. For violations of properties capturing undesired behavior, such plans use compensation actions to allow the application to “go back” to an earlier state at which an alternative path that potentially avoids the fault is available. We call such states “change states”; these include user choices and certain partner calls.

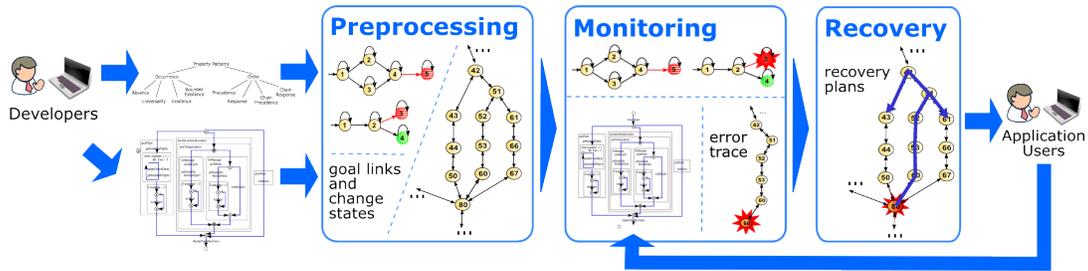


Fig. 2. Overview of our approach.

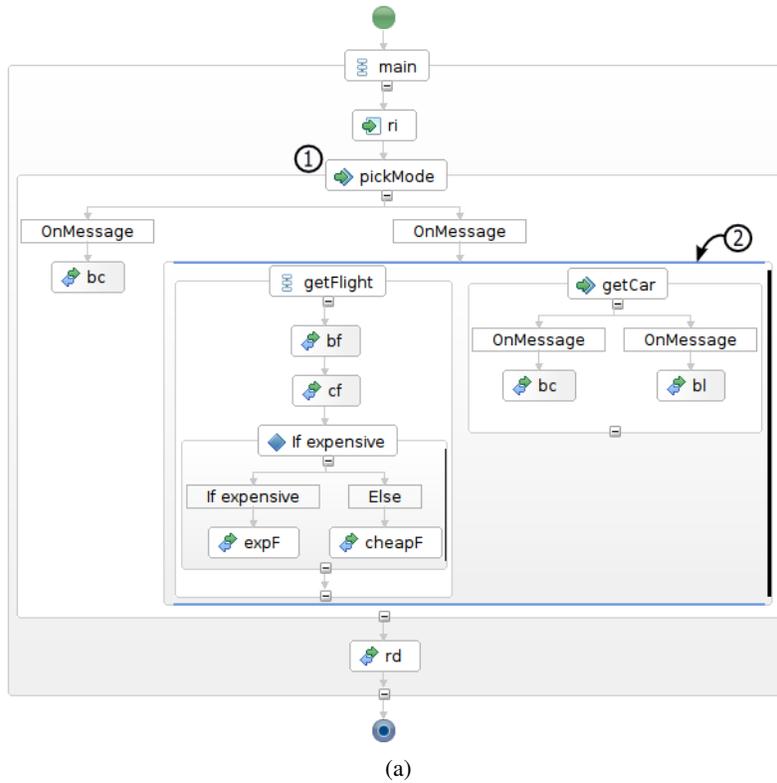
For example, if the TAS system described in Section 1 produces an itinerary that is too expensive, a potential recovery plan might be to undo the limo reservation (so that a car can now be booked) or to undo the flight reservation and see if a cheaper one can be found.

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

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

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

In the rest of this chapter, we further describe and evaluate the above approach. Specifically, we describe inputs to our system, BPEL models and correctness properties, in Section 3. We define the representation of BPEL models as Labeled Transition Systems (LTS) and show how to use these representations for *static* identification of change states and goal transitions in Section 4. In this section, we also discuss how to convert behavioral correctness properties into finite-state automata. We discuss runtime monitoring in Section 5 and describe recovery from violations of behavioral properties in Section 6. We report on our implementation (Section 7) and use it to compute recovery plans for several web service examples (Section 8). We then compare our work with related approaches in Section 9. Finally, in Section 10, we summarize the chapter, give suggestions for future work, and discuss the relationship between our approach and the smart internet vision articulated in [39].



```

<scope name="bf">
  <invoke ... operation = "bf" ... outputVariable = "flightConf" />
  <compensationHandler cost = "g">
    <invoke ... operation = "cancelF" inputVariable = "flightConf"/>
  </compensationHandler>
</scope>

```

(b)

Fig. 3. (a) Workflow of TAS; (b) Compensation for booking a flight (bf).

3 Input

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

3.1 BPEL Programs

BPEL [40] is a standard for implementing orchestrations of web services (provided by partners) by specifying an executable workflow using predefined activities. The basic BPEL activities for interacting with partner web services are <receive>, <invoke> and <reply>, which are used to receive messages, execute web services and return values,

respectively. 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 one of several possible messages (specified using `<onMessage>`) to occur, executing the associated child activity. The `<pick>` activity completes when the child activity completes.

The structural activities `<sequence>` and `<flow>` are used to specify sequential and parallel composition of the enclosed activities, respectively. The `<scope>` activity is used to define a nested activity. In IBM WebSphere Integration Developer v7, developers can also add `<collaboration>` scopes, inspired by the work on dynamic workflows [52], which can be used to alter the application logic at runtime.

Figure 3a shows the BPEL-expressed workflow of the Trip Advisor System (TAS), introduced in Section 1. We use the Eclipse BPEL Project notation¹. TAS interacts with four external services: 1) book a rental car (bc), 2) book a limo (bl), 3) book a flight (bf), and 4) check price of the flight (cf). The result of cf is then passed to local services to determine whether it is expensive (expF) or cheap (cheapF). Service interactions are preceded by a  symbol.

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

Compensation. BPEL's *compensation* mechanism allows the definition of the application-specific reversal of completed activities. For example, the compensation for booking a flight (bf) is to cancel the booking (`cancelF`). This is described in BPEL as shown in Figure 3b: the `<invoke>` and its compensation are enclosed in a named `<scope>` (the scope's name is later used to execute compensation).

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

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

An attempt to compensate a scope for which the CH either has not been installed, or has been installed and executed, is treated as executing an `<empty>` activity (we denote these by τ).

¹ <http://www.eclipse.org/bpel/index.php>.

Absence	An event does not occur within a given scope;
Existence	An event must occur within a given scope;
Bounded Existence	An event can occur at most a certain number of times within a given scope;
Universality	An event must occur throughout a given scope;
Response	An event must always be followed by another within a scope;
Response Chain	A chain of events must always be followed by another chain of events within a scope;
Precedence	An event must always be preceded by another within a scope;
Precedence Chain	A chain of events must always be preceded by another chain of events within a scope.

Table 1. SPS patterns.

We further extended BPEL to allow application developers to associate compensations with different costs, e.g., to indicate that canceling a flight might be significantly more expensive than canceling a car. We do this by adding an extra attribute cost to the definition of `<compensationHandler>`. For example, the flight booking compensation defined in Figure 3b has been assigned a cost of 9 (out of 10), indicating that this is an expensive compensation and should be avoided if possible.

3.2 Specifying Properties

The second input to our system is 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. We assume that the properties are specified using the *Specification Pattern System* (SPS). This system (described below), has been advocated as a standard tool for measuring the practical usefulness and expressive power of specification languages, e.g., [1, 53].

Our framework also includes an (optional) ranking of the properties in the order of importance. 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).

Specification Patterns. The *Specification Pattern System* (SPS), proposed by Dwyer et al. [14], is a pattern-based approach to the presentation, codification, and reuse of property specifications. The system allows patterns like “event P is absent between events Q and S ” or “ S precedes P between Q and R ” to be easily expressed in and translated between linear-time temporal logic (LTL) [44], computational tree logic (CTL) [11], quantified regular expressions (QRE) [41] and other state-based and event-based formalisms.

The property patterns are organized into a hierarchy based on the kinds of system behaviors they describe (see Figure 4a): **Occurrence** patterns talk about the occurrence of a given event/state during system execution, and **Order** patterns specify relative order in which multiple events/states occur during system execution. The patterns are described in Table 1.

Each pattern is associated with *scopes* – the regions of interest over which the pattern must hold. There are five basic kinds of scopes: **Global**, **Before**, **After**, **Between**

Global	The entire program execution;
Before R	The execution up to event R ;
After Q	The execution after event Q ;
Between Q and R	All parts of the execution between events Q and R ;
After Q until R	Similar to Between , except that the designated part of the execution continues even if the second event does not occur.

Table 2. SPS scopes.

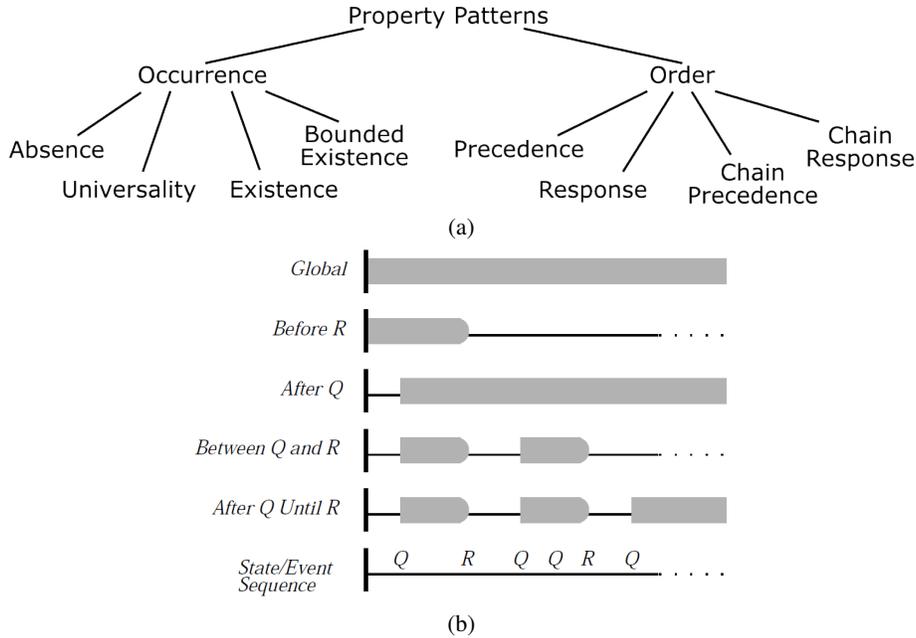


Fig. 4. Specification property system: (a) a pattern hierarchy and (b) pattern scopes.

and **After-Until**. Definitions of these are given in Table 2 and pictorially described in Figure 4b.

Using the Patterns System. To use the pattern system, the specifier begins with a property of interest expressed in natural language. She then identifies atomic actions in the property, then determines a pattern and a scope and chooses a desired output language. For example, the two requirements of the TAS system are to make sure that the customer has the transportation needed to get to her destination, while keeping the costs down. More formally, they become “ P_1 : if requested (ri), TAS will guarantee that the transportation booked reaches the customer’s destination (rd), regardless of the type of transportation chosen”. This is the **Response** pattern in a **Global** scope. We now look up the pattern and the scope in the table. We use the QRE encoding that is easily translatable into monitors as shown in Section 4.2. For our example, it resulting QRE

property is

$$P_1 = [-ri] * \cdot (ri \cdot [-rd] * \cdot rd \cdot [-ri] *) *$$

Formalizing the second requirement, we get “ P_2 : the user cannot book both a limousine (bl) and an expensive flight (expF)”. To express this using patterns, we use two instances of the **Absence** pattern in the **After** scope: A limousine should never be booked (bl) after an expensive flight has been booked (expF) and vice versa. In QRE, we get a pair of properties:

$$P_{2a} = [-bl] * \cdot (bl \cdot [-expF] *) ?$$

$$P_{2b} = [-expF] * \cdot (expF \cdot [-bl] *) ?$$

When monitoring the application, we need to make sure that both P_{2a} and P_{2b} hold in order to comply with the requirement P_2 .

Since we use specifications to establish runtime correctness of a set of conversations between BPEL-expressed partners, we need to determine it using finite traces. Thus, not all patterns are appropriate for this view, since some use future events as preconditions. For example, the scope **Before R** requires that the pattern holds only if **R** eventually occurs on the evaluated path. The same problem occurs with the scope **Between Q and R**. In our setting, the monitoring is performed on the current execution, without looking ahead, and thus those scopes are not supported.

Positive and Negative Behaviors. We differentiate between properties describing *negative* behaviors (that should not appear in the application), and properties describing *positive* behaviors (that the system must possess). For example, property P_1 above 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. In many cases, properties may have both a negative and a positive component, and thus we refer to such properties as *mixed*².

4 Preprocessing

Inputs to the preprocessing stage are the BPEL program B and the set of properties written in QRE. We begin with converting B into a formal representation, $L(B)$, which is a labeled transition system (LTS). We then enrich it with transitions on compensation actions to get $L_C(B)$ (see Section 4.1). In Section 4.2, we discuss the translation of a given

² Formally, mixed properties are either finitary liveness properties or a mixture of finitary liveness and safety properties.

QRE specification into a monitor. Finally, in Section 4.3 we formalize *change states* and potential *goal* transitions and provide an algorithm for computing these statically on $L_C(B)$.

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 [20, 25, 42]. In this work, we build on Foster’s [16] approach of using an LTS as the underlying formalism.

Definition 1 (Labeled Transition Systems) A Labeled Transition System *LTS* is a quadruple (S, Σ, δ, I) , where S is a set of states, Σ is a set of labels, $\delta \subseteq S \times \Sigma \times S$ is a transition relation, and $I \subseteq S$ is a set of initial states.

Effectively, LTSs are state machine models, where transitions are labeled whereas states are not. We often use the notation $s \xrightarrow{a} s'$ to stand for $(s, a, s') \in \delta$. An *execution*, or a *trace*, of an LTS is a sequence $T = s_0 a_0 s_1 a_1 s_2 \dots a_{n-1} s_n$ such that $\forall i, 0 \leq i < n$, $s_i \in S$, $a_i \in \Sigma$ and $s_i \xrightarrow{a_i} s_{i+1}$.

Existing Translation. [16] specifies mapping of all BPEL 1.0 activities into LTS. For example, Figure 5c shows the translation of the `<invoke>` activity `bf` which returns a confirmation number. The activity is a sequence of two transitions: the actual service invocation (`invoke_bf`) and its return (`receive_bf`)³. Conditional activities like `<while>` 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 Figure 5a. Note that both LTSs have two transitions from state 1: $1 \xrightarrow{\text{expr.true}} 2$ and $1 \xrightarrow{\text{expr.false}} 3$. `<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 Figure 5a). `<sequence>` and `<flow>` activities result in the sequential and the parallel composition of the enclosed activities, respectively (see Figure 5b).

Thus, formally, we are going from a BPEL program B to its LTS translation $L(B)$. The set of labels Σ of $L(B)$ 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 $L(B)$ consists of the states produced by the translation as well as a new state `t`. This state is reached from any state of S via a `TER` event: $\forall s \in S \setminus \{t\}, (s, \text{TER}, t) \in \delta$.

Formalizing Compensation. In order to capture BPEL’s compensation mechanism, we introduce additional, backwards transitions. For example, the compensation for `bf`, specified in Figure 3b, is captured by adding the transition $3 \xrightarrow{\text{invoke_cancelF}} 1$ as shown in Figure 5c. Taking this transition effectively leaves the application in a state where

³ Foster’s translation uses names to include traceability information to the BPEL’s scopes. We omit these in this chapter for simplicity.

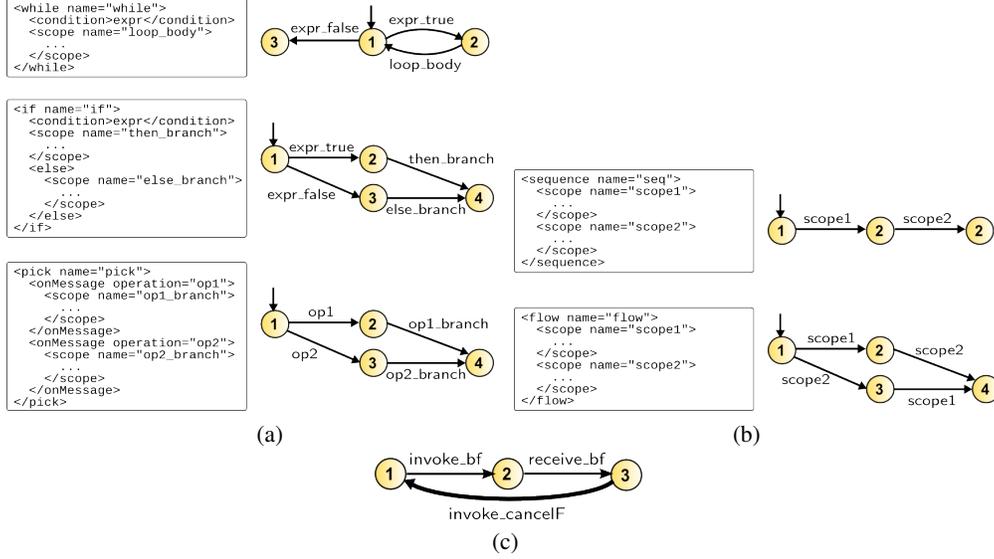


Fig. 5. (a) BPEL conditional activities and their corresponding LTSs; (b) BPEL structural activities and their corresponding LTSs; (c) LTS translation of the `<invoke>` activity bf and its compensation (bold).

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 $L(B) = (S, \Sigma, \delta, I)$, we translate B with compensation into an LTS $L_C(B) = (S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I)$, where Σ_c is the set of compensation actions (including τ) and δ_c is the set of compensation transitions.

Figure 6a shows $L_C(\text{TAS})$. 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 Figure 5c has two transitions: $1 \xrightarrow{\text{bf}} 3$ and $3 \xrightarrow{\text{cancelF}} 1$. This allows us to visually combine an action and its compensation into one transition, labeled in the form a/\bar{a} , where a is the application activity and \bar{a} is its compensation. In other words, each transition $s \xleftrightarrow{a/\bar{a}} t$ in Figure 6a represents two transitions: $(s, a, t) \in \delta$ and $(t, \bar{a}, s) \in \delta_c$.

The `<pick>` activity (labeled ① in Figure 3a) corresponds to state 2 of Figure 6a. The choice between `onlyCar` and `carAndFlight` is represented by two outgoing transitions from this state: $(2, \text{onlyCar}, 3)$ and $(2, \text{carAndFlight}, 6)$. Since these actions do not affect the state of the application, they are compensated by τ . The `<flow>` activity (scope enclosed in bold, blue lines — labeled ② in Figure 3a) 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 .

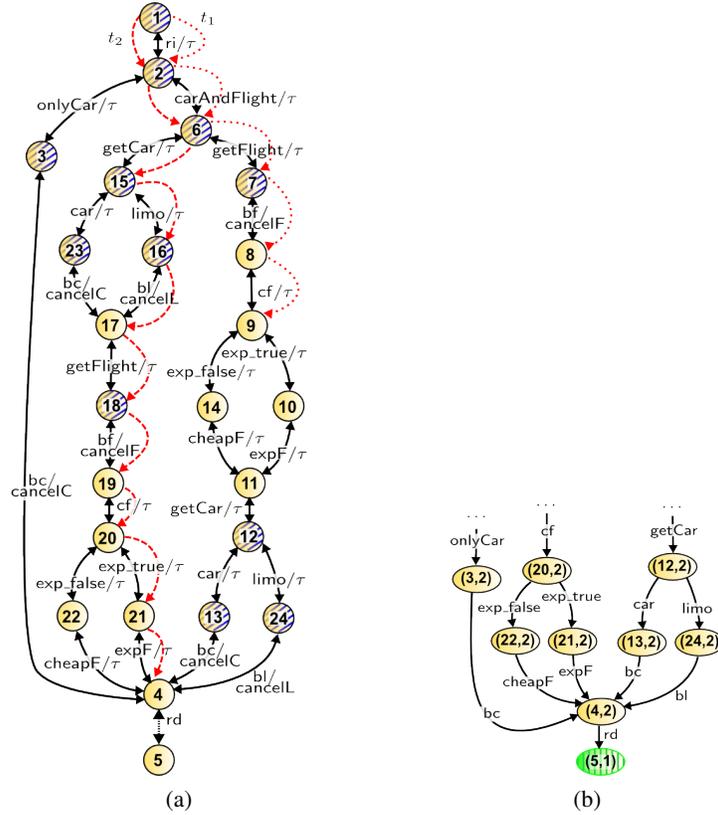


Fig. 6. (a) LTS $L_C(TAS)$, showing traces t_1 (dotted) and t_2 (dashed); (b) a fragment of $L(TAS) \times A_1$.

4.2 From Properties to Monitors

In order to be verified, properties are translated into deterministic finite state machines (FSMs) that we call “monitors”. Different algorithms to perform such a translation from a QRE formula exist in the literature [26]. The translation we use generates a monitor that accepts the *bad* computations of the application – those on which the property fails to hold.

For example, Figure 7c shows the monitor built for the property pattern: “ s responds to p after q until r ”. State 4 of the monitor (colored red and shaded horizontally) is an accepting state, since if we reach it, a violation has been seen: there was a q and later a p (bringing the monitor to state 3), but this p was not followed by s either because r appeared first, or because the application terminated. State 2 (colored green and shaded vertically) is a good state: if we reach it after p was seen, it means that a response by s occurred as needed. Σ is the alphabet of the monitor, i.e., it includes every event occurring in the application, as defined in Section 4.1.

Similar monitors are built for our example properties P_1 and P_2 . Monitor A_1 in Figure 7a represents P_1 : if the application terminates before rd appears, the monitor moves to the (error) state 3. State 1 is a good state since the monitor enters it once the

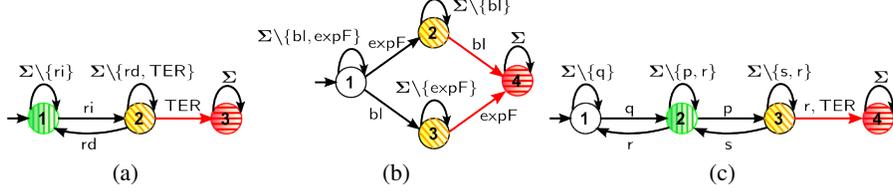


Fig. 7. Monitors: (a) A_1 , (b) A_2 , and (c) for a property pattern “s responds to p after q until r”. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

booked transportations reach the destination (rd). Monitor A_2 in Figure 7b represents both P_{2a} and P_{2b} (defined in Section 3.2). It enters its error state (4) when either a limo was booked and later an expensive flight (corresponding to the violation of P_{2a}), or an expensive flight was booked first and then a limo (violating P_{2b}). We formalize (colored) monitors below.

Definition 2 (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.

We say that A accepts a word $a_0a_1a_2\dots a_{n-1} \in \Sigma^*$ iff there exists an execution $s_0a_0s_1a_1s_2\dots a_{n-1}s_n$ of A such that $a_0 \in I$ 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 colors to states in S . Accepting states are colored red, signaling violation of the property. State 3 of Figure 7a and state 4 in Figures 7b and 7c are red states. Yellow states are those from which a red state can be reached through a single transition. Formally, for a state $s \in S$,

$$color(s) = \text{yellow if } \exists a \in \Sigma, s' \in F \cdot (s, a, s') \in \delta.$$

In addition, we also color yellow those states whose successors are all yellow.

$$color(s) = \text{yellow if } \forall a \in \Sigma, \forall s' \in S \cdot (s, a, s') \in \delta \Rightarrow color(s') = \text{yellow}.$$

State 2 in Figure 7a, states 2 and 3 in Figure 7b and state 3 in Figure 7c are yellow states.

The green color 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,

$$color(s) = \text{green iff } (color(s) \neq \text{red}) \wedge (color(s) \neq \text{yellow}) \wedge (\exists a \in \Sigma, \exists s' \in S \cdot (color(s') = \text{yellow}) \wedge ((s', a, s) \in \delta)).$$

State 1 in Figure 7a, as well as state 2 in Figure 7c are colored green. Note that not all monitors have green states. For example, in A_2 of Figure 7b 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.

Given specification properties $\Phi_1 - \Phi_n$, we translate them to a set $A = \{A_1, \dots, A_n\}$ of monitors, denoting by A_S the subset of A that includes all safety monitors.

4.3 Identifying Change States and Goal Transitions

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

Goal Transitions. In order to find a good recovery plan, we first need to compute a set of *goal transitions*, that is, transitions taken by the application which (immediately) result in some properties reaching a green state. We compute these on a per-property basis. Recall that not all monitors have green states; thus, we define goal transitions only for monitors that do include green states. Let $A_i = (S_i, \Sigma, \delta_i, I_i, F_i)$ be such a monitor. We are looking for transitions in $L(B) = (S, \Sigma, \delta, I)$ which correspond to A_i moving from a yellow state and entering a green state. To find those, we compute the cross-product $L(B) \times A_i$. $(s, a, s') \in \delta$ is a *goal transition* iff $\exists q, q' \in S_i \cdot (s, q) \xrightarrow{a} (s', q') \wedge \text{color}(q) = \text{yellow} \wedge \text{color}(q') = \text{green}$. That is, $s \xrightarrow{a} s'$ corresponds to taking a transition on a from a yellow state into a green state of A_i . The resulting set of goal transitions is denoted by $G(B, A_i)$.

For example, consider a fragment of $L(\text{TAS}) \times A_1$ shown in Figure 6b. The green state of A_1 is state 1, with a transition on rd leading to it from state 2, which is a yellow state. The only transition in $L(\text{TAS}) \times A_1$ satisfying the above definition is $(4, 2) \xrightarrow{\text{rd}} (5, 1)$, and thus $G(\text{TAS}, A_1) = \{(4, \text{rd}, 5)\}$ (depicted by tiny-dashed transitions in Figure 6a).

When computing recovery plans, we need to direct the application towards taking its goal transitions. While the process of identification of goal transitions requires a cross-product computation between the system and each monitor, this computation is done off-line and thus does not affect performance of the overall recovery framework.

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 $L(B)$ that have actions executing which can potentially produce a branch in control flow of the application are called *change states*.

Flow-changing actions are user choices, states modeling the $\langle \text{flow} \rangle$ activity (since each pass through this state may produce a different interleaving of actions), and those service calls whose outcomes are not completely determined by their input parameters but instead depend on the implicit state “of the world”. This characteristics of services is sometimes referred to as *idempotence*, since multiple invocations of the same service yield the same results. Thus, non-idempotent service calls also identify change states. For example, cheapF is a call to determine whether a given flight is cheap and, unless the specification of what cheap means changes, returns the same answer for a given

flight. On the other hand, bf books an available flight, and each successive call to this service can produce different results. Non-idempotent service calls are identified by the BPEL developer as XML attributes in the BPEL program.

Let $C(B)$ denote the set of all change states of the LTS of the application B. For example, in the LTS in Figure 6a, state 6 corresponds to the $\langle \text{flow} \rangle$ 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

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

identified in Figure 6a by shading.

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

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 [50], which has been implemented within the IBM WebSphere business integration products⁴. 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. 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.

The *interception mechanism* used in [50] has been *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 (see Section 7 for details).

For the LTS of the application $L(B) = (S, \Sigma, \delta, I)$, we store the trace of the execution:

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

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

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

a case, state s_n is an *error state* of the application. In addition to T , we also store traces $T^{A_1} \dots T^{A_n}$ that correspond to the executions of the *monitors* $A_1 \dots A_n$, respectively. These are used in the recovery phase (see Section 6). 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 time outs. This scenario corresponds to the trace t_1 , depicted by dotted transitions in Figure 6a. In addition to t_1 , our tool stores $t_1^{A_1}$ and $t_1^{A_2}$ – the corresponding traces of the enabled monitors:

$$\begin{aligned} t_1 &= 1 \xrightarrow{\text{ri}} 2 \xrightarrow{\text{carAndFlight}} 6 \xrightarrow{\text{getFlight}} 7 \xrightarrow{\text{bf}} 8 \xrightarrow{\text{cf}} 9, \\ t_1^{A_1} &= 1 \xrightarrow{\text{ri}} 2 \xrightarrow{\text{carAndFlight}} 2, \xrightarrow{\text{getFlight}} 2 \xrightarrow{\text{bf}} 2 \xrightarrow{\text{cf}} 2, \\ t_1^{A_2} &= 1 \xrightarrow{\text{ri}} 1 \xrightarrow{\text{carAndFlight}} 1 \xrightarrow{\text{getFlight}} 1 \xrightarrow{\text{bf}} 1 \xrightarrow{\text{cf}} 1. \end{aligned}$$

The application server detects that the *cf* invocation timed out, and sends a TER event (not shown in Figure 6a) to the application. Our framework intercepts this TER event and determines that executing it turns t_1 into a failing trace, because the monitor 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 Figure 6a (the traces of the monitors are omitted):

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

As the monitor A_2 has a transition on *expF* to an error state, our framework delays the execution of this event from application state 21. In this example, executing *expF* will make A_2 enter its error state 4, so t_2 is also a failing trace. The *expF* event is not delivered, and the recovery phase is activated.

6 Recovery from 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 3 (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 which failed. In Section 6.1, we discuss recovery from violations of a safety monitor (i.e., the one without a green state), and in Section 6.2, we consider mixed monitors.

$$\begin{array}{l}
r_{18} = 4 \xrightarrow{\tau} 21 \xrightarrow{\tau} 20 \xrightarrow{\tau} 19 \xrightarrow{\text{cancelF}} 18 \quad r_6 = r_{15} \xrightarrow{\tau} 6 \\
\text{(a) } r_{16} = r_{18} \xrightarrow{\tau} 17 \xrightarrow{\text{cancell}} 16 \quad r_2 = r_6 \xrightarrow{\tau} 2 \\
r_{15} = r_{16} \xrightarrow{\tau} 15 \quad r_1 = r_2 \xrightarrow{\tau} 1 \\
\\
p_0 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\tau} 6 \xrightarrow{\tau} 2 \xrightarrow{\text{onlyCar}} 3 \xrightarrow{\text{bc}} 4 \\
\text{(b) } p_1 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\text{bf}} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp.true}} 10 \xrightarrow{\text{expF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{car}} 13 \xrightarrow{\text{bc}} 4 \\
p_2 = 9 \xrightarrow{\tau} 8 \xrightarrow{\text{cancelF}} 7 \xrightarrow{\text{bf}} 8 \xrightarrow{\text{cf}} 9 \xrightarrow{\text{exp.false}} 14 \xrightarrow{\text{cheapF}} 11 \xrightarrow{\text{getCar}} 12 \xrightarrow{\text{car}} 13 \xrightarrow{\text{bc}} 4
\end{array}$$

Fig. 8. Recovery plans for TAS: (a) plans for the safety violation of trace t_2 ; (b) plans of length ≤ 10 for recovery from the mixed property violation of trace t_1 .

<pre> <sequence name="r18"> <compensateScope target="expF" /> <compensateScope target="exp_true" /> <compensateScope target="cf" /> <compensateScope target="bf" /> </sequence> </pre>	<pre> <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> </pre>
(a)	(b)

Fig. 9. XML versions of recovery plans in Figure 8: (a) for r_{18} ; (b) for p_0 .

6.1 Recovery from Safety Monitor Violations

The recovery procedure for a safety property violation receives $L_C(B)$ – the LTS of the running application B with compensations (see Section 4.1), T – the executed trace ending in an error state e (see Section 5) and $C(B)$ – the set of change states (see Section 4.3).

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 $C(B)$ or the initial state of $L_C(B)$. 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 Figure 6a. $\{1, 2, 6, 15, 16, 18\}$ are the change states seen along t_2 . This leads to the recovery plans shown in Figure 8a. We add state names between transitions for clarity and refer to plans p_s to mean “recovery to state s ”. A given plan can also become a prefix for the follow-on one. This is indicated by using the former’s name as part of definition of the latter. For example, recovery to state 16 starts with recovery to state 18 and then includes two more backward transitions, the last one with a non-empty compensation. Plan r_{18} can avoid the error if, after its application, the user chooses a cheap flight instead of an expensive one. Executing plan r_{15} gives the user the option of changing the limousine to a rental car, and plan r_2 – the option of changing from an air/ground combination to just renting a car. Both of these behaviours do not cause the violation of A_2 .

Computed plans are then converted to BPEL for presentation to the user. For example, plan r_{18} is shown in Figure 9a. The chosen plan can then be applied (see Section 7), allowing the program to continue its execution from the resulting change state.

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

Discussion. Note that plan r_{16} which cancels the limo, would lead to rebooking it right away which may still leave the possibility of booking an expensive flight and violating the property P_2 . The reason why this plan might not be as useful as others is that computation of change states in Section 4.3 treats all non-idempotent service calls as the same, whereas not all might be *relevant* to the satisfaction of properties of interest. See [49] for a description of computation and evaluation of effectiveness of relevant change states.

6.2 Recovery from Mixed Monitor Violations

When a monitor with a green state detects a violation during execution (i.e., it is in a red state), we attempt to direct the execution towards one of its green states.

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

A recovery plan effectively “undoes” actions along T , starting with e and ending in a change state (otherwise, the plan would not be executable!) and then “re-plans” the behavior to reach the goal while avoiding redundant loops (i.e., when some actions are executed and then immediately compensated). Our solution adapts techniques from the field of *planning* [24], described below.

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

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

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

$$P(B, A, T) = (L_C(B), e, G_s(B, A)).$$

Note that when a plan p to a goal state s is computed, we need to extend it with an additional transition, $p \xrightarrow{a} s'$ to account for $(s, a, s') \in G(B, A)$. For example, consider

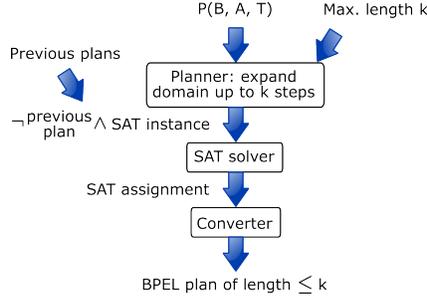


Fig. 10. Planning Diagram

the trace t_1 of Figure 6a, described in Section 5, in which monitor A_1 fails. We define the planning problem $P(\text{TAS}, A_1, t_1) = (L_C(\text{TAS}), 9, \{4\})$, where 9 is the initial state (see Figure 6a) and $G_s(\text{TAS}, A_1) = \{4\}$ (see Section 4.3), and its result, p , should be expanded to $p \xrightarrow{\text{rd}} 5$.

Unfortunately, we cannot simply use a planner as a “black box”, for two main reasons. First, not every trace returned by solving $P(B, A, T)$ is acceptable: our recovery plans must visit change states. Second, we may want to produce multiple recovery plans to let the user select the best – a requirement that is not supported by planners.

Instead, we look at how planners encode the planning graph and then manipulate the produced encoding directly, to add additional constraints. Several existing planners, such as BlackBox [28], translate the planning graph into a CNF formula and then use a SAT solver, such as SAT4J⁵, to find a satisfying assignment for it. Such an assignment, if found, represents a plan. We use a planner to translate a planning problem into a SAT instance, and then modify this instance for our needs.

Figure 10 gives a schematic overview of our method. The planning domain as well as the maximum length k of required plans are fed into a planner, which we use only for producing a CNF formula. We then modify the CNF formula to account for change states, and iteratively restrict it further to disallow already seen plans. The modified formula is then given as input to a SAT solver, and the satisfying assignment it produces is converted back into a BPEL plan. A detailed explanation of the modifications made to the CNF formula can be found in [48].

For example, consider the TAS problem and the error trace t_1 shown in Figure 6a (ending in state 9). Looking for plans up to length 10, we get plans p_0 , p_1 and p_2 shown in Figure 8b. And, as mentioned earlier, each plan is extended with the last goal 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 and p_2 also cancel the flight (since 8 is not a change state whereas 7 is) and then proceed to re-book it and then book the car, regardless of the flight’s cost. Increasing the plan length, we also get the option of taking the getCar transition out of state 6, book the car and then the flight.

⁵ <http://www.sat4j.org/>

The produced plans are then ranked based on the length of the plan and the cost of compensation actions in it:

$$\text{cost}(p) = c_1 \times \text{length}(p) + c_2 \times \sum_{i=1}^{\text{length}(p)} \text{compensation-cost}_i(p),$$

where c_1 and c_2 are constants, $\text{length}(p)$ is the number of events in the plan p and $\text{compensation-cost}_i$ is a cost of the i th compensation action. For example:

$$\text{cost}(p_0) = 8 + 6 = 14,$$

assuming $c_1 = c_2 = 1$, and $\text{cost}(p_1) = \text{cost}(p_2) = 17$. Thus, p_0 is ranked the highest and presented first. Of course, the cost function does not take into account the time the user will spend driving rather than flying should she select plan p_0 , so she may choose one of the alternative plans instead.

Chosen plans are then converted to BPEL for execution. The compensation part of the plan is similar to the one shown in Figure 9a, and the re-planning part consists of a sequence of BPEL `<invoke>` and `<pick>` operations (see the XML translation of plan p_0 in Figure 9b).

In addition, we can aim to limit the number of recovery plans computed by taking two issues into consideration: (a) making sure that the plan goes through only “relevant” change states, i.e., those that affect the computation of the violating trace, and (b) removing those plans that result in the violation of one of the safety properties (see [49]).

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

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

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

7 Tool Support

We have implemented the process described in this chapter using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts. The preprocessing and runtime monitoring phases of our framework are the same for both safety

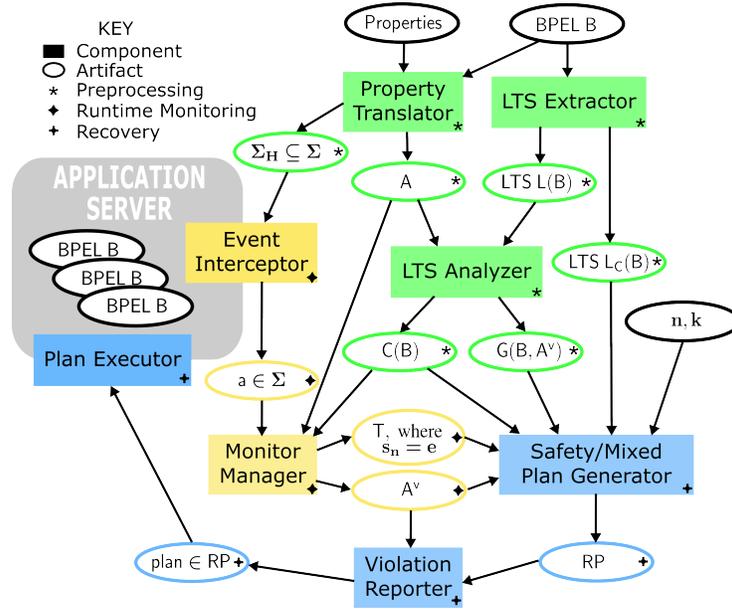


Fig. 11. Architecture of the framework.

and mixed properties, but different components are required for generating plans from the two types of properties. We show the architecture of our framework in Figure 11. 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 \star symbol; runtime monitoring – yellow, with a \blacklozenge symbol; and recovery – blue, with a \blackplus symbol. Artifacts with a black border are the initial inputs to our framework.

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 (as described in Section 4.2). The *LTS Extractor* (LE) component extracts an LTS model from the BPEL program and creates a second LTS model with compensation (both processes are described in Section 4.1). The *LTS Analyzer* (LA) 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 updates the state of each active monitor, until an error has been found (which activates the recovery phase) or all partners terminate. MM also stores the intercepted events for recovery.

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 6.1). For mixed properties, plans can compensate executed activities and execute new activities. In this case, the *Mixed Plan Generator* first generates

the corresponding planning problem and then modifies it in order to generate as many plans as required (see Section 6.2).

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). If no monitor is violated during the execution of the chosen plan (MM updates the states of the active monitors during the plan execution), the framework switches back to runtime monitoring. We describe these components in more detail below.

7.1 Preprocessing

As the developer is responsible for the preprocessing phase, we have implemented this part of our framework as a WebSphere Integration Developer plugin.

Property Translator provides a graphical interface for specifying properties using property patterns and application events. Properties are translated into QREs, from which we generate a set of monitors A , as explained in Section 4.2. These monitors are stored in the Aldebaran [6] format for use by the rest of the components.

LTS Extractor receives as input a BPEL program B in the BPEL4WS XML format. We use the WS-Engineer extension [18] to LTSA [36] to translate B into an LTS $L(B)$ and then export it in the Aldebaran format [6], with an `.aut` extension. Since WS-Engineer does not support the full handling of BPEL compensations, we built our own `.aut-to-.aut` Python script (`add_comp.py`) which uses B and $L(B)$ to produce $L_C(B)$ as described in Section 4.1. Traceability between the BPEL and the resulting LTS is established by the WS-Engineer's encoding of BPEL scopes into names of LTS actions. This traceability allows us to convert the computed plans back to BPEL.

LTS Analyzer receives as input the application monitors and LTS $L(B)$ (both in the Aldebaran format). We wrote a script `compute_cp.py` that computes the cross-product between the application and each A_i in $A \setminus A_S$ and uses these cross-products to identify goal links, as described in Section 4.3. This component also checks which service invocations of B have been marked as non-idempotent, and uses this information to identify the application change states (as described in Section 4.3).

7.2 Runtime Monitoring

The monitoring phase is implemented on top of the IBM WebSphere Process Server, a BPEL-compliant process engine for executing BPEL processes and a built-in Service Component Architecture (SCA), which is a particular instantiation of SOA.

The *Event Interceptor* (EI) is deployed on the process server and establishes a bridge through which our runtime monitoring framework communicates with the server to obtain information about the web service execution. On the process server, SCA is responsible for the invocation of native SCA service components and for the binding and interaction with external services. EI monitors interactions within the SCA application server runtime environment, and is responsible for observing and routing these invocation requests and responses to MM. If EI observes an event that may cause termination or a property violation (in other words, a monitor currently in a yellow state transitions

to a red state on that event), the event is first forwarded to MM for analysis. If no violation is detected, the execution continues as normal. Otherwise, EI stops forwarding events to the corresponding application instance until a recovery plan is executed.

Monitor Manager receives A – the set of monitors produced by the PT component. During execution, monitors can be enabled/disabled through MM. A new copy of each active monitor is created for each new instance of the application. The MM component registers itself as a listener to EI, updating the state of all active monitors when a new event is received. MM also stores the current execution trace for each application instance. In the case of a monitor violation, MM broadcasts the violated monitor A^v and the corresponding error trace T , initiating recovery.

7.3 Recovery

The recovery phase is also implemented on top of the IBM WebSphere Process Server. This allows us to avoid recomputing recovery plans by keeping a centralized hash of property violations and computed recovery plans. The maximum plan length (k) and the maximum number of plans (n) are the configuration parameters of the framework.

Safety Plan Generator receives as input k , n , $L_C(B)$, $C(B)$, and T . We use our own script (`gen_safe_plan.py`) to determine which visited change states are reachable from the error state e on $L_C(B)$, within the maximum plan length, and the set of recovery plans RP is produced as a by-product of this check.

Mixed Plan Generator receives as input k , n , $L_C(B)$, $C(B)$, $G(B, A_v)$, A_v , and T . We use our own script (`gen_plan_prob.py`) to translate $L_C(B)$ into a planning problem $(L_C(B), e, G_s(B, A^v))$ (see Section 6.2). The planning problem is expressed in STRIPS [15] – an input language to the planner Blackbox [28] which we use to convert it into a SAT instance. The maximum plan length is used to limit the size of the planning graph generated by Blackbox, effectively limiting the size of the plans that can be produced. We use another new script (`GenPlans.java`) to successively modify the initial SAT instance in order to produce alternative plans. It calls the satisfiability solver SAT4J, extracts plans from the satisfying assignments produced by SAT4J, ranks them and converts them to the BPEL4WS XML format for displaying and execution. SAT4J is an *incremental* SAT solver, i.e., it saves results from one search and uses them for the next. For our method of generating multiple plans, where each SAT instance is more restricted than the previous one, this is particularly useful, leading to efficient analysis.

Violation Reporter (VR) receives as input A_v and a list of BPEL plans RP . VR generates a web page snippet with violation information, as well as a form for selecting a recovery plan. A snippet generated for a violation of P_1 is shown in Figure 12a. Developers must include this snippet in the default error page, so that the computed recovery plans can be shown when an error is detected. Figure 12b 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 (r in the case of safety properties, p otherwise). Figure 12c shows a screen shot of `error.jsp` after recovery plans for P_1 have been computed.

Plan Executor 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

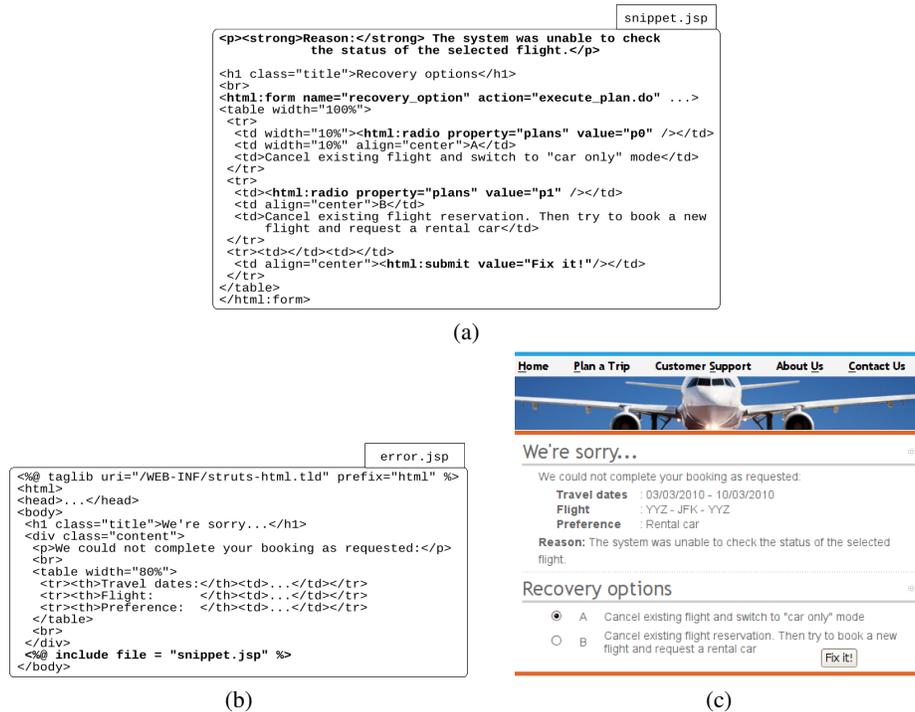


Fig. 12. Violation reporting: (a) `snippet.jsp`, automatically generated snippet that contains recovery plans; (b) `error.jsp`, the application error handling page; (c) `error.jsp` displayed on a browser.

as the logic of this scope. EI also intercepts application events during the execution of the recovery plan, and a new recovery plan must be chosen if the current one causes a monitor violation.

Because web services are distributed and allow asynchronous message communication, messages may get delivered and received out of order. To handle out-of-order events, we annotate each event with two timestamps: one at invocation and one at reception. When events arrive at the message queue of MM, these timestamps are used to check if the invocation ordering is consistent with the reception ordering. If the orderings are not consistent, detected errors may be caused by network delays rather than incorrect conversations. Currently, all timestamps are generated by the same WebSphere Process Server.

8 Case Studies

We have applied our framework to several web service applications and report on our experience on recovery from property violations. In the first case study (see Section 8.1), we show how to recover from multiple problems seeded in the Travel Book-

ing System (TBS) (adapted from [22]). Experience with this larger, more complex case study shows that our approach is both effective and scalable.

In Section 8.2 we report on an experiment running our method on the *Flickr* system (see Carzaniga et al. [8]). In [8], several aspects of this system are modeled as finite-state machines, and the paper shows how to use redundancies in the system in order to “work around” some vulnerabilities. We reverse-engineered BPEL applications from the finite-state models presented in [8], and we compare our recovery plans to the method presented in [8].

8.1 Travel Booking System (TBS)

The Travel Booking system (TBS) provides travel booking services over the web. In a typical scenario, a customer enters the expected travel dates, the destination city and the rental car location – airport or hotel. The system searches for the available flights, hotel rooms and rental cars, placing holds on the resources that best satisfy the customer preferences. If the customer chooses to rent a car at the hotel, the system also books the shuttle between the airport and the hotel. If the customer likes the itinerary presented to him/her, the holds are turned into bookings; otherwise, the holds are released. Figure 13 shows the BPEL implementation of this system.

Implementation. TBS interacts with three partners (FlightSystem, HotelSystem and CarSystem), each offering the services to find an available resource (flight, hotel room, car and shuttle), place a hold on it, release a hold on it, book it and cancel it. Booking a resource is compensated by canceling it (at a cost of 8 out of 10), and placing a hold is compensated by a release (at a cost of 2). All external service calls are non-idempotent.

The workflow begins by `<receive>`ing input (`receiveInput`), followed by `<flow>` with two branches, as the flight and hotel reservations can be made independently. The branches are labeled ① and ②: ①) find and place a hold on a flight, ②) place a hold on a hotel room (this branch has been simplified in this case study). If there are no flights available on the given dates, the system will prompt the user for new dates and then search again (up to three tries). After making the hotel and flight reservations, the system tries to arrange transportation (see the `<pick>` activity labeled ③): the user `<pick>`’s a rental location (`pickAirport` or `pickHotel`) and the system tries to place holds on the required resources (car at airport, or car at hotel and a shuttle between the airport and hotel).

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

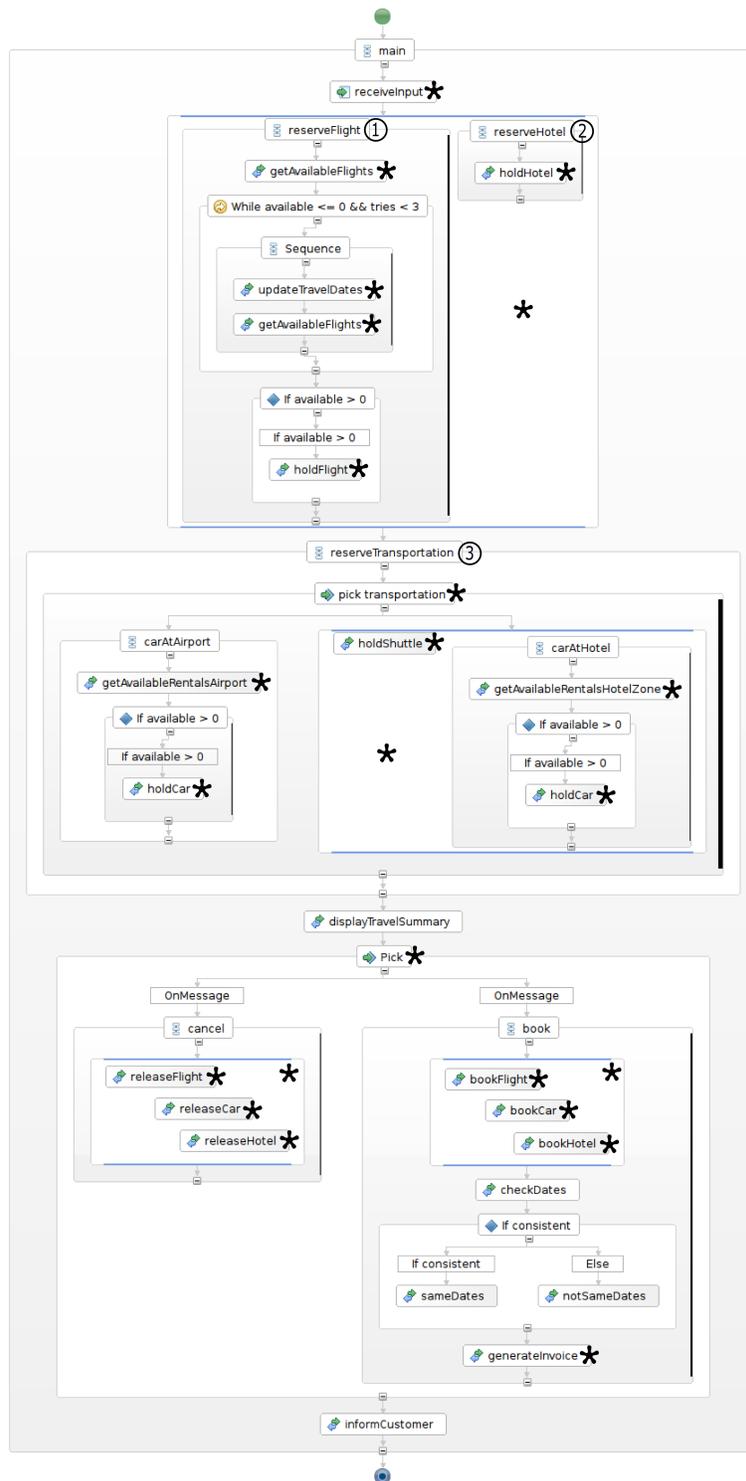


Fig. 13. BPEL implementation of the Travel Booking System.

Properties. Some behavioural correctness properties of TBS are P_3 : “there shouldn’t be a mismatch between flight and hotel dates”, and P_4 : “a car reservation request will be fulfilled regardless of the location (i.e., airport or hotel) chosen”. We can express these properties using patterns:

P_3 : **Absence** of a date mismatch event (notSameDate) **After** both a flight and hotel have been booked (bookFlight and bookHotel, in any order),

and

P_4 : **Globally** place a hold on a car (holdCar) in **Response** to a rental location selection (pickHotel or pickAirport).

Note that P_3 is a safety property, describing a forbidden scenario, while P_4 has also a desired component and it is thus a mixed property.

Preprocessing. We translated the two properties into monitoring automata: P_3 (P_4) has 5 (3) states and 10 (6) transitions. P_3 (P_4) has 0 (1) green, 1 (1) yellow, and 1 (1) red states, so we could compute safety plans for P_3 and mixed plans for P_4 . The LTS $L(TBS)$ has 52 states and 67 transitions, and $|\Sigma| = 33$, which makes TBS double the size of the TAS example. 20 of the BPEL activities (highlighted with a \star symbol in Figure 13) yield a total of 35 change states in the LTS. P_4 is a mixed property, with three goal links corresponding to it.

Experience: Recovery from a safety property violation. We generated a recovery plan for the following scenario (called trace t_3 , of length 21) which violates property P_3 : The application first makes a hotel reservation (holdHotel) and then prompts the user for new travel dates (updateTravelDates), since there were no flights available on the current travel dates. The car rental location is the airport (pickAirport). The system displays the itinerary (displayTravelSummary) but the user does not notice the date inconsistency and decides to book it. The TBS makes the bookings (bookFlight, bookHotel and bookCar) and then checks for date consistency (checkDates). In this case, the dates are not the same (notSameDates), which allows us to detect the violation of P_3 and initiate recovery.

We generated plans starting with length $k = 5$ and going to $k = 30$ in increments of 5. In order to generate all possible plans for each k , we chose n – the maximum number of plans generated – to be MAX_INT. Table 3 summarizes the results. A total of 13 plans were generated, and the longest plan, which reaches the initial state, is of length 21 (and thus the rows corresponding to $k = 25$ and $k = 30$ contain identical information). Since t_3 violates a safety property, no SAT instances were generated, and the running time of the plan generation is trivial.

The following plans turn t_3 into a successful trace: p_A^3 – cancel the flight reservation and pick a new flight using the original travel dates, and p_B^3 – cancel the hotel reservation and pick a new hotel room for the new travel dates. Our tool generated both of these plans, but ranked them 11th and 12th (out of 13), respectively. They were assigned a low rank due to the interplay between the following two characteristics of our

case study: (i) the actual error occurs at the beginning of the scenario (in the flight and hotel reservation <flow>), but the property violation was only detected near the end of the workflow (in the book flow), and (ii) t_3 passes through a relatively large number of change states, and thus many recovery plans are possible.

The first of these causes could be potentially fixed by writing “better” properties – the ones that allows us to catch an error as soon as it occurs. We recognize, of course, that this can be difficult to do. The second stems from the fact that not all service calls marked as non-idempotent are relevant to P_3 or its violation. In the future, we intend to explore this direction by trying to rank change states w.r.t. their relevance to the property, with the hope of reducing the occurrences of cases like this.

Experience: Recovery from a mixed property violation. The following scenario (we call it trace t_4 , with length 14), violates property P_4 . Consider an execution where the user has chosen to rent the car at the hotel (pickHotel), but no cars are available at that hotel. TBS makes flight, hotel and shuttle reservations (holdFlight and holdHotel), but never makes a car reservation (holdCar). The user does not notice the missing reservation in the displayed itinerary (displayTravelSummary) and decides to book it. The TBS tries to complete the bookings, first booking the hotel (bookHotel) and then the car (bookCar). When the application attempts to invoke bookCar, the BPEL engine detects that the application tries to access a non-initialized process variable (since there is no car reservation), and issues a TER event. Rather than delivering the TER event to the application, we initiate recovery.

We are again using $n = \text{MAX_INT}$ and varying k between 5 and 30, in increments of 5, summarizing the results in Table 3. The first thing to note is that our approach generated a relatively large number of plans (over 60) as k approached 30. While in general, the further we move away from a goal link, the more alternative paths lead back to it, this was especially true for TBS which had a number of <flow> activities. The second is that our analysis remained tractable even as the length of the plan and the number of plans generated grow (around 1 min for the most expensive configuration).

Executing one of the following plans would leave TBS in a desired state: p_A^4 – attempt the car rental at the hotel again, and p_B^4 – cancel the shuttle from the airport to the hotel and attempt to rent a car at the airport. Unlike t_3 , the error in this scenario was discovered soon after its occurrence, so plans p_A^4 and p_B^4 are the first ones generated by our approach. p_A^4 actually corresponds to two plans, since the application logic for reserving a car at a hotel is in a <flow> activity, enabling two ways of reaching the same goal link. Plan p_B^4 was the 3rd plan generated.

The rest of the plans we generated compensate various parts of t_4 , and then try to reach one of the three goal links. While these longer plans include more compensations and are ranked lower than p_A^4 and p_B^4 , we still feel that it may be difficult to the user to sift through all of them. We are currently actively pursuing the problem of reducing the number of plans generated to recover from mixed properties. One of the approaches is to remove those plans that (necessarily) lead to violations of safety properties. For now, this remains “work in progress” and is not presented in this chapter.

8.2 Comparisons with a Related Approach

While in the Travel Booking System, we were comparing the effectiveness of the generated plans to our expectations, in the set of examples that follow, we compare the effectiveness of the plans we generate with a related approach – that of Carzaniga et al [8]. In each case, we describe the example from [8], discuss our experience translating it to BPEL and expressing correctness properties and then report on the plans we generate and the relevant statistics.

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

We created the Flickr visibility system (FV) by reverse-engineering the behavioral model in Figure 14a (given in [8]) and expressing it in BPEL (see Figure 14b). The behavioral model has four states: `notOnFlickr`, `public`, `private` and `family`. `notOnFlickr` is the initial state, executing the `upload()` operation (with a visibility parameter) from this state leads to one of the three other states (visibility states). The BPEL model FV, consists of 20 activities (6 with explicit compensations).

In Figure 14b, the transitions from the initial state are modeled in the `<scope>` called `upload` (labeled ①). In this scope, we call three different upload services depending on the upload visibility: `uploadPub`, `uploadPriv` and `uploadFam` (equivalent to `upload()`, `upload(isPublic_OFF)` and `upload(isFamily_ON)`, respectively).

The transition relation between the visibility states specifies valid changes in the photo visibility. This has been modeled using case statements in the `<scope>` called `changePerm` (labeled ②). In this scope, `setPermPub`, `setPermPriv` and `setPermFam` are equivalent to `setPerm(isPublic_ON)`, `{setPerm(isPublic_OFF), setPerm(isFamily_OFF)}` and `setPerm(isFamily_ON)`, respectively.

We also defined compensation for FV. Since there were no transitions back to state `notOnFlickr`, we assumed that the upload services do not have compensation. However, compensation for the `setPerm` services is obvious – reverse the permission setting, e.g., `setPerm(isPublic_ON)` is compensated by `setPerm(isPublic_OFF)`. The `upload` and `setPerm` service calls are non-idempotent.

Converted to LTS, the resulting model has 28 states and 37 transitions. $L(FV)$ is larger than the original behavioral model since the LTS includes BPEL-induced actions such as entering scopes, and we used case statements to model operation parameters.

We then expressed properties of the FV system: “If a user tries to set a photo’s visibility to X , Flickr will guarantee that the photo will have the visibility X ”, where X is each of the possible visibilities. These became separate properties expressed using the **Response** pattern. An instance with $X = \textit{family}$ will “catch” the identified vulnerability in the case where a photo is initially loaded as *private*.

Flickr comments. Flickr lets users comment on uploaded photos. While any user can add a comment to a *public* photo, only authorized users can comment on *private* and

App.	Our approach					[8]	
	k	vars	clauses	plans	time (s)	length	plans
TAS	6	135	254	1	0.01	-	-
	8	798	10,355	5	0.13	-	-
	13	1,398	25,023	13	0.27	-	-
TBS t_3	5	-	-	2	0.01	-	-
	10	-	-	5	0.02	-	-
	15	-	-	8	0.02	-	-
	20	-	-	12	0.02	-	-
	25	-	-	13	0.02	-	-
	30	-	-	13	0.02	-	-
TBS t_4	5	108	464	0	0.01	-	-
	10	883	30,524	2	0.14	-	-
	15	1,456	74,932	8	1.37	-	-
	20	2,141	135,047	18	4.72	-	-
	25	3,298	246,210	60	29.16	-	-
	30	5,288	464,654	68	61.34	-	-
FV	15	797	16,198	2	0.04	≤ 2	1
	22	1,436	33,954	4	0.74	≤ 3	5
	26	1,804	44,262	8	1.14	≤ 4	13
	42	3,276	85,494	40	3.12	≤ 8	412
FC	4	42	159	1	0.01	≤ 1	0
	6	95	592	2	0.02	≤ 2	2
	12	321	3,248	4	0.15	≤ 3	8
	16	554	7,393	5	0.27	≤ 4	22
	20	856	14,427	13	1.38	≤ 8	484

Table 3. Plan generation data. “-” mark cases which are not applicable, such as references to SAT for recovery from safety property violations.

LTS model has 18 states and 22 transitions. We expressed FC’s property “if a user adds a comment to a *public* photo that has comments enabled, the comment should be successfully added to the photo’s comments” using the **Response** pattern.

Experience. The number of recovery plans generated for failed traces of FV and FC is shown in Table 3. For example, for the plan length up to 26, we have generated 8 plans for FV. The longest plan was of length 42. We now look at the effectiveness of the plan generation process. For FV, one of the plans we generate for $k = 22$ is “compensate changes in visibility until the photo becomes *private* again, set the photo visibility to *public* and change visibility to *family*”, which corresponds to the workaround plan chosen by [8]. For FC, the plan corresponding to the chosen workaround is “delete the problematic comment, toggle the comments permission and then try to add the comment again”, generated when $k = 12$.

To compare the precision of our approach, i.e., the number of plans generated, we look at the list of workaround sequences computed by [8] (see Table 3). The work in [8] modeled the Flickr behavior directly and the model did not include BPEL-induced actions such as entering scopes. Further, the workaround sequences did not include the

“going back” part – they were plans on how to execute a task starting from the initial state. Thus, the plans we generate are somewhat longer. For example, the workaround sequences of length ≤ 2 correspond to our plans of length $k = 15$. With this adjustment, Table 3 shows that we generate significantly fewer plans of the corresponding length. We also generate every plan marked by [8] as desired.

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

8.3 Scalability

To check whether SAT-solving done as part of the planning is the bottleneck of our approach, we measured sizes of SAT problems for FV, FC, TBS, and our running example, TAS, listing them in Table 3. For all four systems, the number of variables and the number of clauses grows linearly with the length of the plan, as expected, and the running time of the SAT solver remains in seconds.

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

9 Related Work

Our work deals with monitoring web applications and, when violations found, uses planning techniques to propose recovery measures. This work is different from the approach of monitoring web services for quality assurance (e.g., [30, 45, 54]). The reason is that we rely on an implicit model for behavioural correctness (expressed using properties) and do our checking only w.r.t. the behaviour rather than other attributes such as the mean response time of external services.

In Section 9.1, we survey other (behavioural) monitoring frameworks, and in Section 9.2, we compare our method to other self-healing approaches for web applications. Note that the area of monitoring web services for quality assurance has been

9.1 Runtime monitoring of web services

Monitoring techniques for web services can be roughly divided into *offline* techniques (for example, [37, 38, 51]), that analyze system events *after* execution, and *online* techniques [3, 4, 32, 34, 35, 43] that, like in our method, monitor the system as it executes. These techniques differ in the types of properties they can handle. *Global properties* allow the analysis of orchestrated obligations. These obligations are expressed from the point of view of the orchestrating service, but also include events from the other services involved in the conversation being monitored. *Local properties* are restricted to monitoring the events of a single service. Furthermore, some techniques concentrate on *state* properties, whereas others allow the developer to express *sequences* of events.

The approach introduced by Pistore et al. [43] is the closest to our monitoring framework. It can be used to check global properties that are specified in LTL, and thus it is somewhat more expressive than our input properties (although may prove more difficult to use [13]). However, the main interest of [43] is the synthesis of a BPEL composition, and it does not deal with recovery.

The frameworks described in [3,4,32,34,35] are restricted to local properties. Li et al. [34] specify properties using Interaction Constraints (IC) [33] – a language based, like our method, on Dwyer’s Specification Pattern System [13]. Unlike our automata though, IC does not allow pattern nesting. Thus, new events must be introduced in order to reason about sequences of events. The rest of the local property frameworks check state formulas, specified using simple predicate logic. Specifically, Baresi et al. [3,4] and Lohmann et al. [35] check service pre- and postconditions associated to external service invocations, while Lazovik et al. [32] check local assertions.

Offline techniques can handle both global and local properties. In Mahbub et al. [37,38], properties are expressed using event calculus [46]. Van der Aalst et al. [51] introduce DerSecFlow, a graphical language that can be used to express properties similar to our patterns, but without pattern nesting.

Various techniques are used for *checking* properties. [32] and [43] rely on planning techniques to create service compositions. [43] analyzes the application once the composition has been obtained, by instrumenting the system to include Java code that checks LTL monitors during runtime. [32] iteratively replaces the violated service with another one, with weaker assertions, continuing the process until there are no more violations, or the composition is not possible.

In the case of service pre- and postconditions, [3,4] modify the original BPEL diagram, introducing new BPEL activities that check the contract during external service calls. [35] proposes a similar, but more intrusive framework, as JML contracts are integrated at the source code level. [37,38] use temporal deductive databases to store and reason about events generated during runtime, while [51] analyzes low-level event logs using an LTL checker.

Techniques used in the work of Li et al. [34] are the closest to ours. Like us, they take an automata-based approach for monitoring communications between partners and enable graphical display of violations.

9.2 Recovery and Self-Healing

The advantage of online techniques is that it is possible for the system to react once a problem has been detected. Existing infrastructures for web services, e.g., the BPEL engine [40], include mechanisms for fault definition, for specification of compensation actions, and for dealing with termination. When an error is detected at runtime, they typically try to compensate all completed activities for which compensations are defined, with the default compensation being the reversal of the most recently completed action. Instead, our approach allows for a *guided* recovery. While using the compensation mechanism to reverse activities, we also direct the application forward, towards a goal state.

In [3,4], BPEL exception handlers can be attached to the properties being checked. If such an exception handler is not provided, the execution terminates when a violation

occurs. As [35, 43] are Java-based, they can use Java’s exception handling mechanism for recovery actions; however, this approach is highly intrusive.

Several works have suggested “self-healing” mechanisms for web-service applications. The Dynamo framework [5] uses *annotation rules* in BPEL in order to allow recovery once a fault has been detected. Such rules need to be installed by the developers before the system can function. In contrast, our work uses an existing compensation mechanism and requires no extra effort from developers.

[21] proposes 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. Our approach uses compensations for *individual* actions and can dynamically recover from errors as they are detected.

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

An emerging research area in recent years is that of *self-adaptive* and *self-managed* systems (see [7, 9, 10, 31] for a partial list). A system is considered self-adaptive if it is capable of adjusting itself in response to a changing environment. This approach is different from ours, since in our framework no change is made to the system itself, and recovery plans are discovered and executed using the original application. However, some similarities do exist. For example, a major issue in our approach is the identification of a goal state, which should become the target of the recovery plan. The problem of finding a *desired* or *correct* state [10] to which a system should evolve, is a concern in the field of self-adaptive system as well.

The work of Carzaniga et al. [8] is the closest to ours in spirit. It exploits redundancy in web applications to find workarounds when errors occur, assuming that the application is given as a finite-state machine, with an identified error state as well as the “fallback” state to which the application should return. The approach generates all possible recovery plans, without prioritizing them. In contrast, our framework not only detects runtime errors but also calculates goal and change states and in addition automatically filters out unusable recovery plans (those that do not include change states) and ranks the remaining ones. See Section 8 for a detailed comparison.

10 Conclusion, Discussion and Future Work

In this chapter, 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, does not significantly affect the performance of the monitored system, 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. Our experience has shown that this approach computes a small number of highly relevant plans, doing so quickly and effectively. In what follows, we discuss limitations of our approach and

venues for future research. We also speculate about how the move towards the “smart” internet – the vision articulated in [39] – will affect our approach.

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

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

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

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.

Monitoring and Recovery for Smart Web Service Interactions. In this chapter, we have described how to monitor and recover from violations in the traditional web service model, where applications are predefined and are deployed on the server.

The emerging paradigm of smart internet and thus smart interactions, described elsewhere in the current volume [39], would shift the emphasis towards the user, who maintains a list of personal goals (defined in [39] as *matters of concern* (MOC)) which persists between individual sessions with various services.

The move to smart internet would affect our proposed framework as follows:

1. User MOCs are obvious candidates for liveness properties for our framework. They essentially describe user desires to get something accomplished, e.g., making a purchase of a great gift. In addition, users operate under a variety of constraints, such as making sure that they stay within their budget or that the gift’s arrival day

is before Christmas. Thus, we feel that user properties in the smart internet model are effectively MOCs subject to constraints. In our model, constraints are described using safety properties and MOCs – using liveness properties.

However, our approach, as presented, has a limitation: the properties need to be expressible by end users. While the pattern-based approach certainly makes property expression easier than the traditional, logic-based approach, it still may not be appropriate for the end users.

2. In our approach, compensation and its cost is defined statically in BPEL. In order to move our approach to the smart internet model, compensation and its cost should be user-specified (e.g., to account for cases where some users pay smaller fees for a transaction cancellation, be that for a stop payment or for canceling a flight). Unfortunately, we are not aware of existing technology which allow such dynamic, user-centered compensation definition and configuration.
3. Finally, in the traditional model of internet, applications are created and tested by software developers. In the smart internet domain, the standard notion of testing as means of quality assurance cannot be applied, since each user has her own version of the application, with its own MOC, constraints and compensation. Thus, monitoring is the only way to ensure correctness of such applications. Furthermore, monitoring and recovery has to be conducted on the user side rather than on the central server.

Overall, while there are a number of hurdles to overcome to make behavioural monitoring and recovery truly usable for the smart internet paradigm, we feel that this approach is a promising way (perhaps, the only way!) of ensuring quality of user-centric web systems where the level of customization does not allow effective testing. Thus, we intend to join forces with the other groups who contributed to this volume to make the smart internet vision a reality, allowing users to engage in non-trivial, meaningful and non error-prone interactions with the web.

Acknowledgements

Several people contributed to the ideas presented in this chapter: 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 and Alessandra Gorla for sharing her examples and data with us. This research has been funded by NSERC, IBM Toronto, MITACS Accelerate, and by the Ontario Post-Doctoral Fellowship program.

References

1. Marco Autili, Paola Inverardi, and Patrizio Pelliccione. A Scenario Based Notation for Specifying Temporal Properties. In *Proceedings of the 2006 ICSE International Workshop on*

- Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06)*, pages 21–28, 2006.
2. Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Claudio Schifanella. Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step. In *Proceedings of International Workshop on Web Services and Formal Methods (WS-FM'05)*, volume 3670 of *LNCS*, pages 257–271, 2005.
 3. Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart Monitors for Composed Services. In *Proceedings of 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 193–202, November 2004.
 4. Luciano Baresi and Sam Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In *Proceedings of 3rd International Conference on Service Oriented Computing (ICSOC'05)*, pages 269–282, 2005.
 5. Luciano Baresi and Sam Guinea. Dynamo and Self-Healing BPEL Compositions (research demonstration). In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 69–70. IEEE Computer Society, 2007. Companion Volume.
 6. Marius Bozga, Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Protocol Verification with the ALDÉBARAN Toolset. *International Journal on Software Tools for Technology Transfer*, 1(1-2):166–184, 1997.
 7. Yuriy Brun and Nenad Medvidovic. Fault and Adversary Tolerance as an Emergent Property of Distributed Systems' Software Architectures. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems, (EFTS'07)*, pages 1–7, September 2007.
 8. Antonio Carzaniga, Alessandra Gorla, and Mauro Pezze. Healing Web Applications through Automatic Workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, 2008.
 9. Betty H. C. Cheng, Rogério de Lemos, David Garlan, Holger Giese, Marin Litoiu, Jeff Magee, Hausi A. Müller, and Richard Taylor. Seams 2009: Software engineering for adaptive and self-managing systems. In *31st International Conference on Software Engineering, (ICSE'09), Companion Volume*, pages 463–464, May 2009.
 10. Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaella Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.
 11. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
 12. Glen Dobson. Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)*, pages 126–133, August, 2006.
 13. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, pages 411–420, May 1999.
 14. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, March 1998.
 15. Richard Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Journal of Artificial Intelligence*, 2(3/4):189–208, 1971.
 16. Howard Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, Imperial College, 2006.

17. Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based Verification of Web Service Compositions. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 152–163. IEEE Computer Society, 2003.
18. Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. LTSA-WS: a Tool for Model-Based Verification of Web Service Compositions and Choreography. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 771–774, May 2006.
19. Xiang Fu, Tevfik Bultan, and Jianwen Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. In *Proceedings of the Eighth International Conference on Implementation and Application of Automata (CIAA'03)*, pages 188–200, July 2003.
20. Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *Proceedings of the 13th international conference on World Wide Web (WWW'04)*, pages 621–630, May 2004.
21. Maria Grazia Fugini and Enrico Mussi. 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*, pages 67–80, September 2006.
22. Yuan Gan. Runtime Monitoring of Web Service Conversations. Master's thesis, University of Toronto, Department of Computer Science, March 2007.
23. N. Ghafari, A. Gurfinkel, N. Klarlund, and Richard Trefler. Algorithmic Analysis of Piecewise FIFO Systems. In *Proceedings of 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 45–52, November 2007.
24. Fausto Giunchiglia and Paolo Traverso. Planning as Model Checking. In *Proceedings of the 5th European Conference on Planning (ECP'99)*, pages 1–20, September 1999.
25. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In *Proceedings of the 3rd International Conference on Business Process Management (BPM'05)*, volume 3649 of *LNCS*, pages 220–235, September 2005.
26. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
27. P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. 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)*, pages 405–409, 2005.
28. Henry A. Kautz and Bart Selman. Unifying SAT-based and Graph-based Planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 318–325, 1999.
29. Raman Kazhamiakin and Marco Pistore. A Parametric Communication Model for the Verification of BPEL4WS Compositions. In *Proceedings of International Workshop on Web Services and Formal Methods (WS-FM'05)*, volume 3670 of *LNCS*, pages 318–332, 2005.
30. Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.
31. Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *The ICSE'07 Workshop on the Future of Software Engineering (FOSE'07)*, pages 259–268, May 2007.
32. Alexander Lazovik, Marco Aiello, and Mike P. Papazoglou. Associating Assertions with Business Processes and Monitoring Their Execution. In *Proceedings of 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 94–104, November 2004.

33. Zheng Li, Jun Han, and Yan Jin. Pattern-Based Specification and Validation of Web Services Interaction Properties. In *Proceedings of 3rd International Conference on Service Oriented Computing (ICSOC'05)*, pages 73–86, 2005.
34. Zheng Li, Yan Jin, and Jun Han. A Runtime Monitoring and Validation Framework for Web Service Interactions. In *Proceedings of the 17th Australian Software Engineering Conference (ASWEC'06)*, pages 70–79. IEEE Computer Society, 2006.
35. Marc Lohmann, Leonardo Mariani, and Reiko Heckel. A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services. In *Test and Analysis of Web Services*, pages 173–204. Springer, 2007.
36. Jeff Magee and Jeff Kramer. *Concurrency - State Models and Java Programs*. John Wiley, 1999.
37. Khaled Mahbub and George Spanoudakis. A Framework for Requirements Monitoring of Service Based Systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 84–93, New York, NY, USA, 2004. ACM.
38. Khaled Mahbub and George Spanoudakis. Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In *Proceedings of International Conference on Web Services (ICWS'05)*, pages 257–265, July 2005.
39. Joanna W. Ng, Mark Chignell, and James R. Cordy. The Smart Internet: Transforming the Web for the User. In *CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 285–296, New York, NY, USA, 2009.
40. OASIS. Web Services Business Process Execution Language Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, Accessed January 2009.
41. Kurt M. Olender and Leon J. Osterweil. “Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation”. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.
42. Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.
43. Marco Pistore and Paolo Traverso. Assumption-Based Composition and Monitoring of Web Services. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 307–335. Springer, 2007.
44. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.
45. Akhil Sahai, Vijay Machiraju, and Klaus Wursterl. Monitoring and Controlling Internet Based E-Services. In *Proceedings of the Second IEEE Workshop on Internet Applications (WIAPP '01)*, page 41, Washington, DC, USA, 2001. IEEE Computer Society.
46. Murray Shanahan. The Event Calculus Explained. In *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *LNCS*, pages 409–430. Springer, 1999.
47. Jocelyn Simmonds. *Dynamic Analysis of Web Services*. PhD thesis, Department of Computer Science, University of Toronto, 2010. (in preparation).
48. Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. Guided Recovery for Web Service Applications. In *Proceedings of Eighteenth International Symposium on the Foundations of Software Engineering (FSE'10)*, 2010. To appear.
49. Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. Optimizing Computation of Recovery Plans for BPEL Applications, 2010. Submitted.
50. Jocelyn Simmonds, Yuan Gan, Marsha Chechik, Shiva Nejati, Bill O'Farrell, Elena Litani, and Julie Waterhouse. Runtime Monitoring of Web Service Conversations. *IEEE Transactions on Service Computing*, 2009.

51. Wil M. P. van der Aalst and Maja Pestic. Specifying and Monitoring Service Flows: Making Web Services Process-Aware. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 11–55. Springer, 2007.
52. Wil M. P. van der Aalst and Mathias Weske. Case Handling: a New Paradigm for Business Process Support. *Data Knowledge Engineering*, 53(2):129–162, 2005.
53. Jian Yu, Tan Phan Manh, Jun Han, Yan Jin, Yanbo Han, and Jianwu Wang. Pattern Based Property Specification and Verification for Service Composition. In *Proceedings of 7th International Conference on Web Information Systems Engineering (WISE'06)*, pages 156–168, 2006.
54. Farhana H. Zulkernine, Patrick Martin, and Kirk Wilson. A Middleware Solution to Monitoring Composite Web Services-Based Processes. In *Proceedings of the 2008 IEEE Congress on Services Part II (SERVICES-2 '08)*, pages 149–156, Washington, DC, USA, 2008. IEEE Computer Society.