Dynamic Analysis of Web Services

by

Jocelyn Simmonds

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Dynamic Analysis of Web Services

Jocelyn Simmonds

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2011

Orchestrated web service applications are highly distributed applications that accomplish business goals by executing services offered by partners. This dependance on partner services allows the development of more flexible, modular applications. For a classical distributed system, correctness can be ensured by statically checking the composition of the components that make up the system against properties of interest. However, in the case of web service applications, there are various conditions that make this type of analysis insufficient. For example, partners can be dynamically discovered, which means that we cannot create a definitive model of the system to analyze. Web service applications can also display new behaviour at execution time, so statically checked properties of the system may not hold throughout the system's lifetime.

Due to these limitations of static analysis, this thesis concentrates on the dynamic analysis of web service applications, specifically, by monitoring runtime events. The goal of runtime monitoring is to check whether an application violates a given specification of its behaviour during its execution. The behaviour of the system can be specified in a number of ways, e.g., as a set of temporal properties, assertions or even scenarios. During execution, application events are intercepted and used to determine if the system is violating its specification. Moreover, monitoring the system as it runs provides a chance to recover from an error once a problem has been detected. This is critical in the domain of web service applications, as bugs are potentially exposed to millions of users before they are found/fixed. We present techniques to address several major challenges facing the creation of an industrial-strength runtime monitoring and recovery framework for web service applications.

The first milestone for achieving this goal is the creation of an adequate property specification language. This language must be expressive enough to capture the distributed, interactive, and message-driven nature of web service applications, but must also be amenable to efficient runtime monitoring. We propose Web Sequence Diagrams (W-SD), a language that, we feel, meets these criteria. Specifications expressed in W-SD permit the analysis of orchestrations involving multiple partners, from the point of view of the orchestrating service.

The second contribution of this thesis is the creation of an industrial-strength online runtime monitoring and recovery framework that is non-intrusive, supports the dynamic discovery of web services, deals with synchronous and asynchronous communication, as well as partner services implemented in different languages. Developers using this framework can specify and efficiently monitor a variety of temporal behaviour. If recovery is enabled, properties are monitored proactively, so this framework allows developers to effortlessly enable error recovery in applications being monitored.

The last contribution of this thesis is the development of recovery plans from runtime errors. Given an application path which led to a failure and a monitor which detected it, we have developed various techniques and optimizations that make recovery plan generation feasible in practice. 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.

Acknowledgements

I want to take this opportunity to thank all the people that had an influence in the work presented in this thesis. First and foremost, none of this work would have been possible without Marsha Chechik's support. I am extremely grateful for the time she dedicated to my projects, as well as her inspiration (and her formidable knowledge of Formal Methods!). I would also like to thank my committee members, Luciano Baresi, Eric Hehner and Sheila McIlraith, for their insigtful comments and suggestions about my work. I also have to thank the people at the IBM Toronto Lab, Bill O'Farrell, Elena Litani and Leho Nigul, who inspired various aspects of the problems studied in this thesis. I also need to thank Cecilia Bastarrica, who (strongly) encouraged me to apply to the PhD program.

On a personal note, I would like to also thank my family and friends. My fiancé Juan Pablo has been extremely patient and understanding during this whole process, his love and support has made the experience easier. My parents, George and Jennifer, who always knew when I needed some extra encouragement. My siblings and their families, who always open their homes to me. To my Toronto roommates, Mihaela Bobaru, and later Alivia Dey and Maria Modanu, who were always checking whether I was getting enough sleep. A special thanks to all my Toronto friends, those that guided me as a new PhD student, and those that later were under my wing (you know which group you belong to!): Mehrdad Sabetzadeh, Shiva Nejati, Mihaela Bobaru, Jorge Baier, Daniela Nuñez, Andres Lagar-Cavilla, Claudia Garcia, Yiqiao Wang, Anya Tafliovich, Golnaz Elahi, Justin Ward, Amy Miller, Jennifer Horkoff, Alicia Grubb, Jessica Davis, Maryam Fazel, Elizabeth Lam, Aws Albarghouthi and Michalis Famelis (hopefully I did not forgot anybody). I would also like to thank the gals from the Latinas in Computing group, I really appreciated your support.

Contents

Chapter 1

Introduction

1.1 Motivation

Recent years have seen an emergence of service-oriented applications, where applications are created using existing services. Companies with large legacy back-end systems, like Deutsche Post AG and Credit Suisse, have used this approach to modernize existing infrastructure, repackaging legacy applications as services, leading to an overall reduction in maintenance costs [58]. On the other hand, Web 2.0 companies like Google and Amazon offer services that grant access to data that was (possibly) expensive to collect, allowing 3rd party developers to create new applications using this data. These services are commonly referred to as *web services*, as they are usually available via the web.

As the individual services are developed on a wide variety of platforms, there is a need for a flexible architecture that standardizes how these services interact. The Service-Oriented Architecture (SOA) is such an architecture. Partners develop services and make them available by publishing service interfaces. Web services can be written in a traditional compiled language such as Java^{TM}, or in an XML-centric language such as BPEL [76], in which predefined activities are used to specify executable workflows. *Web service* applications are created by specifying how these partner services interact

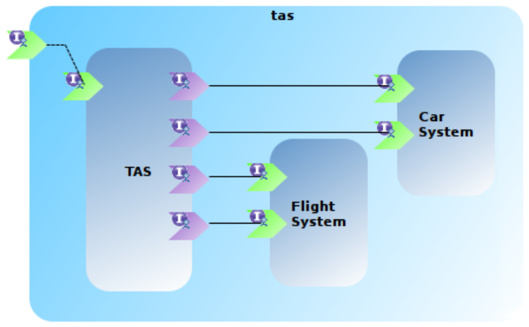


Figure 1.1: Assembly diagram describing interactions between the main TAS process and its partners.

to achieve stakeholder requirements. The main process of a web service application is usually referred to as the *main* (or *orchestrating*) web service. Service-orientation is not new concept, as the SOA standard is a generalization of existing web technologies like Remote Method Invocations (RMI) and Common Object Request Broker Architecture (CORBA).

1.1.1 Example: the Trip Advisor System

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 behaviour) 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 behaviour).

Figure 1.1 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 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 1.1 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 offline while the user attempts to book ground transportation, thus preventing the entire system from getting the user to her destination.

1.1.2 Developing Quality Web Service Applications

This dependance on partner services allows the development of more flexible, modular applications, but also introduces various problems. Since the SOA standard allows dynamic service discovery and binding, web service applications can display new behaviour at execution time. Changes to partner services during execution can also lead to new runtime behaviour. Applications without adequate exception handling can crash if a partner service is not available, e.g., due to network problems. Web service applications can also fail because there are bugs in the orchestrating service, e.g., due to faulty logic and bad data manipulation, or because of the incorrect invocation of services.

Another serious problem with web service applications is that bugs are now potentially exposed to millions of users before they are found/fixed. For example, Amazon.com suffered a three hour outage on June 29, 2010, displaying blank or partial pages instead of product listings. No official reason has been given for this service outage, but at an annual revenue of nearly \$27 billion, Amazon faces a potential loss of an average of \$51,400 a minute when its site is offline [24].

Since runtime failures of web services are inevitable, frameworks 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 in the TAS example would be to cancel the booking. These error recovery mechanisms can be used to minimize the impact of runtime bugs, but the developer must anticipate possible runtime errors since these mechanisms are statically defined. Also, the application must be restarted once the bug has been fixed, which can affect existing user sessions.

The combination of these issues makes the development and maintenance of quality web service applications quite a challenge in practice, leading us to formulate the following thesis statement:

> The main objective of this work is to enable the efficient analysis of web service applications (with dynamic service binding). Also, due to the possibly large impact of bugs, another goal of this work is to attempt to automate some level of error recovery.

To accomplish the first goal, we must focus on how to specify and check the correctness of web service applications. For a classical distributed system, correctness can be ensured by statically checking the composition of the components that make up the system against properties of interest. This approach has been taken by several researchers in the context of web services, e.g., [34, 35, 55, 8, 32]. While static analysis is very appealing – errors are discovered ahead of time and without the need to exercise the system, this approach has three major limitations:

1. As discussed in the beginning of this section, partners are dynamically discovered, which means that we cannot create a definitive model of the system to analyze.

2. Another issue is that web services typically communicate via infinite-length channels, so the problem is decidable only under certain conditions [39].

3. Finally, since web service applications can exhibit new behaviour at runtime, statically checked properties of the system may not hold throughout the system's lifetime.

Due to these limitations of static analysis, this thesis concentrates on the dynamic analysis of web service applications, specifically, by monitoring runtime events. Moreover, monitoring the system as it runs provides a chance to recover from an error once a problem has been detected. In the rest of this section, we give a brief overview of dynamic analysis techniques for web service applications, as well as existing work on error recovery, and outline some of the challenges of applying these techniques to web service applications.

1.2 Dynamic Analysis of Web Service Applications

A goal of dynamic analysis is to check whether an application violates a given specification of its behaviour during its execution. The behaviour of the system can be specified in a number of ways, e.g., as a set of temporal properties, assertions or even scenarios. During execution, application events are intercepted and used to determine if the system is violating its specification. This is commonly referred to as *runtime monitoring*. Reported violations can be used to debug both the application and its specification.

Runtime monitoring is a very flexible technique: applications can be monitored for both functional (e.g., assertions) and non-functional requirements (e.g., performance), and the analysis can be done while the application is running, or after the fact. The complexity of the analysis is determined by the amount and type of information collected at runtime. There are also various practical considerations that must be taken into account when applying these techniques. The language used to specify properties must be expressive enough for users to specify properties of interest, but resulting specifications must also be efficiently monitorable. Another important factor is the intrusiveness of the approach – the amount of information available at runtime affects the precision of the analysis. In the following section, we briefly discuss a classification of dynamic analysis frameworks.

1.2.1 A Classification of Runtime Monitoring Frameworks

The following axes can be used to classify dynamic analysis frameworks – 1) *online* vs. *offline*: runtime events can be analyzed during or after execution, 2) *global* vs. *local* properties: frameworks can check properties of the whole application or of the individual services that make up the application (or both), and 3) *passive* vs. *active*: passive frameworks just report property violations, while active frameworks attempt to react to violations, offering some form of error recovery. Frameworks that check local properties usually insert assertions in the appropriate application locations, while those that check global properties usually convert the properties into monitors that are updated as the application runs. Online frameworks can be passive or active, but offline approaches can only be passive.

1. **Online vs. Offline.** Online frameworks monitor predefined properties, collecting just those events which are related to these properties. While expressing properties beforehand may be non-trivial, the collected data is guaranteed to be both small and sufficient to check these properties; they also serve as an additional, and very valuable, *documentation* of the desired behaviour of the system. Offline frameworks analyze event logs, allowing the expression of free-form queries over all stored events. However, since these queries are not necessarily known a priori, the runtime data collected might not be sufficient to answer the relevant questions, or, on the other extreme, the amount of data collected may become excessive and hard to manage, leading to intractable analysis.

2. **Global vs. Local properties.** Global properties allow the analysis of orchestrated obligations, i.e., desired or forbidden scenarios (*conversations*) involving collaborating partner services. These obligations are expressed from the point of view of the monitored application, but can 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, expressed either as a property of the current state, or as a desired/forbidden sequence of events.

3. **Passive vs. Active.** Passive frameworks let the monitored application continue execution undisturbed when a property violation is detected. Active frameworks attempt to maintain behavioural correctness, for example, by trying to return the application to a stable state after a violation is detected. Error recovery mechanisms are discussed in the next section.

1.2.2 Error Recovery

The advantage of online frameworks 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 [76], include mechanisms for fault definition, for specifying compensation actions, and for dealing with termination. When an error is detected at runtime, they typically try to compensate all completed activities for which compensations are defined, with the default compensation being the reversal of the most recently completed action.

Since these standard error recovery mechanisms are statically defined, one relatively simple manner of improving error recovery is to statically analyze the application and suggest changes that improve the application's fault tolerance. In [25], Dobson defined a library of fault tolerance patterns, which are used to transform the original BPEL process into a fault-tolerant one at compile time. This is done by adding redundant behaviour to the application, but this may result in a significantly bigger, and slower, program.

The work proposed by Baresi et al. [9, 10] also enables recovery through the standard error recovery mechanisms, but by attaching BPEL exception handlers to properties that are checked at runtime. The advantage to this approach is that the new exceptions are triggered by the violation of high-level properties, which can help debugging. If such an exception handler is not provided, execution terminates when a property is violated.

An emerging research area in recent years is that of *self-healing* systems (see [14, 59, 19, 18] for a partial list). A system is considered self-healing if it is capable of detecting failures and diagnosing faults, and can adjust itself in response. Error recovery frameworks omit the diagnosis phase, and can thus be classified as simple self-healing systems.

Several works have suggested self-healing mechanisms for web service applications. The Dynamo framework [11] uses *annotation rules* in BPEL in order to allow recovery once a fault has been detected. Such rules need to be statically defined by the developers before the system can function. Fugini and Mussi [36] propose a framework for self-healing web services, where all possible faults and their repair actions are pre-defined in a special registry. This approach relies on being able to identify and create recovery from all available faults.

Carzaniga et al. [17] exploit 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. This approach generates all possible recovery plans, prioritizing them by length.

1.3 Challenges

There are still many challenges in the development of an online monitoring and error recovery framework for web service applications. In this section, we outline some of those challenges (in no specific order).

1. **Specification.** As with any formal analysis technique, the correct formalization of properties of the system is a major challenge. Formal languages like temporal logic are hard to use by practitioners, and do not capture the characteristics of web service applications. A property specification language for web service applications must be able to capture the distributed, interactive, and message-driven nature of

business processes. Such a language should enable specifying a variety of properties, allowing the analysis of orchestrations involving multiple partners, from the point of view of the orchestrating service. Support for data should also be considered. Also, in order to improve the usability of such a specification language, we believe that this language should be visual. To address this problem, we have adopted a subset of UML 2.0 Sequence Diagrams (SD) as a specification language. SDs are used to capture interactions in the form of message passing between objects. They have been widely adopted by industry as a suitable language for describing and documenting scenario-based requirements specifications. We aim to show that this language is sufficiently expressive to capture a wide variety of frequently used properties, as well as study its formalization, so as to enable monitoring.

2. **Runtime monitoring.** A problem closely related to the Specification challenge is that runtime monitoring brings a sense of false security – if an application did not violate any properties during testing, then there are no bugs! But since we are dealing with incomplete behavioural specifications and a finite set of execution traces, obviously there may be many more undiscovered bugs. Moreover, because of the dynamic nature of web service applications, properties that went unviolated for large periods of time may suddenly be violated. Another problem is that no matter how expressive the property specification language is, it must still be amenable to efficient runtime monitoring. Each web service application can be executed by thousands of clients simultaneously, and thus monitoring all client interactions with the application would mean a significant monitoring overhead unless distributed to the individual clients. Also, not all individual violations are interesting – once a violation has been discovered, its corresponding monitor can be switched off until a patch is introduced. To address this problem, we must develop a configurable, non-intrusive runtime monitoring framework and we must also adapt/develop various case studies in order to evaluate the practical aspects of such a framework.

3. **Error recovery.** A major limitation of most existing error recovery frameworks is that they assume that the orchestrating service is internally consistent and that errors only appear during interactions with partner services. Another limitation of these frameworks is that possible recovery plans must be specified at design time by the application developers. Currently, there is a push towards automated recovery plan generation, but existing proposals do not scale (both with respect to the application size, and to the number of plans generated). When considering automated plan generation, an important question is what can be considered a recovery action, as well as who defines these actions and their cost. Another question is how to evaluate how "good" a plan is (which enables plan ranking). To address this problem, new recovery plan generation techniques must be created, incorporating domain information existing in the orchestrating web service and its properties.

1.4 Contributions

The main contribution of this thesis is to address some of the challenges facing runtime monitoring and error recovery described in the previous section. We address the "Specification" challenge by proposing a scenario-based property specification language for web service applications. We address the "Runtime Monitoring" challenge by proposing a non-intrusive, online runtime monitoring framework that combines and extends existing runtime verification techniques, as part of the IBM WebSphere product suite [50]. We also propose various optimization techniques that improve the effectiveness of our framework in practice. Finally, we address the "Error recovery" challenge by developing new property-guided, SAT-based, recovery plan generation techniques. We have explored these challenges in the specific context of BPEL, but the ideas presented in this thesis can be applied to other web service application development frameworks that allow user-defined compensation. In the rest of this section, we give a more detailed overview of these contributions.

Property Specification Language for Web Service Applications. The first contribution of this thesis is Web Sequence Diagrams (W-SD), a property specification language for web service applications. This language is a subset of UML 2.0 Sequence Diagrams (SD), a feature-rich language without a formal semantics. In this thesis, we describe the semantics of our chosen subset of SDs, and show how to translate properties specified in W-SD into automata, enabling runtime monitoring. We also show that this language is sufficiently expressive to capture a wide variety of frequently used properties, captured and catalogued in the Specification Pattern System (SPS) [28].

Runtime Monitoring and Recovery. The second contribution of this thesis is the creation of an industrial-strength online runtime monitoring and recovery framework that is non-intrusive, supports the dynamic discovery of web services, deals with synchronous and asynchronous communication, as well as partner services implemented in different languages. Developers using this framework can specify and efficiently monitor a variety of temporal behaviour. If recovery is enabled, properties are monitored proactively, so this framework allows developers to effortlessly enable error recovery in applications being monitored.

Property-driven Error Recovery. The third contribution of this thesis is the development of recovery plans from runtime errors. Given an application path which led to a failure and a monitor which detected it, we have developed various techniques and optimizations that make recovery plan generation feasible in practice. For violations of safety properties, these recovery plans attempt to return the application to an earlier state at which an alternative path that potentially avoids the fault is available. For violations of (bounded) liveness properties, merely going back is insufficient to ensure that the system can produce the desired behaviour. In this

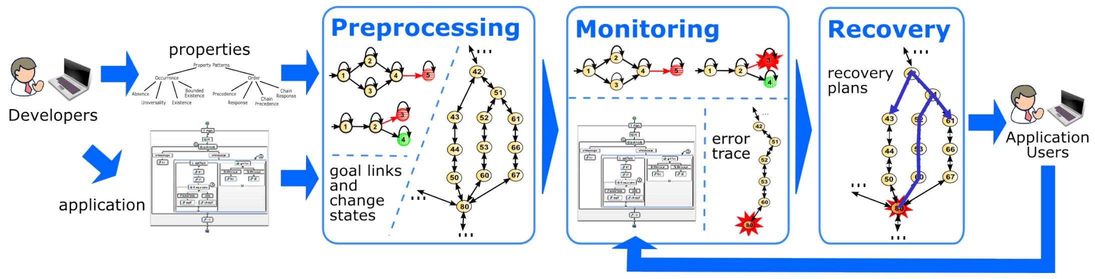


Figure 1.2: Overview of our approach.

case, we compute plans that attempt to redirect the application towards executing new activities that may lead to the satisfaction of the property in question. We rely on BPEL's compensation mechanism to "undo" application actions.

The work presented in this thesis is based upon, and extends, several papers and reports that have been published in the last three years ([88, 89, 84, 87, 85, 86]). Work on tool support for the approach was carried out as part of various internships at the IBM Toronto Lab. This thesis should be regarded as the definitive account of this work.

1.5 Our Approach

Figure 1.2 shows a schematic view of our approach to runtime monitoring and error recovery. In our approach, developers supply a BPEL program and a set of behavioural correctness properties that need to be maintained by the program as it runs. These properties can be visually specified using our subset of UML 2.0 Sequence Diagrams, but can also be directly specified using the Specification Pattern System (SPS). In this thesis, we show examples of both types of specifications, but only formalize error recovery for the second type of specification. The BPEL program is enriched (by its developers) with the compensation mechanism which allows us to compensate 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 labelled transition system.

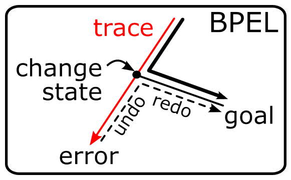


Figure 1.3: A schematic view on plan generation.

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 recovery 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 behaviour, 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. For example, if the `TAS` system described in Section 1.1.1 produces an itinerary that is too expensive, a potential recovery plan might be to cancel the limo reservation (so that a car can now be booked) or to cancel 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 behaviour. Thus, in order to satisfy (bounded) liveness properties, 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 cancelling 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 (error) state in the system, find a plan to achieve the goal that goes through a change state.

This process is shown schematically in Figure 1.3. 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.

1.6 Organization

The rest of this thesis is structured as follows. In Chapter 2, we give an overview of the web technologies used in the rest of this thesis, as well as fix our notation. In Chapter 3, we present Web Sequence Diagrams (W-SD), a property specification language for web service applications (reported in [89]), as well as discuss its formalization and property templates. Chapter 4 describes our approach to runtime monitoring and develops the connection between user-defined properties and automated recovery plan generation, reported in [89, 84]. Chapter 5 describes RuMoR, our runtime monitoring and recovery framework (reported in [87, 85]), that implements the techniques presented in Chapter 4. In Chapter 6, we present the case studies used to evaluate RuMoR, as well as present a couple of optimizations to the recovery plans generation process (reported in [86]). Finally, we conclude in Chapter 7 with a summary of this thesis and an outline of future research directions.

Chapter 2

Preliminaries

This thesis focuses on the analysis of service-based applications. There are many different proposals for how such applications should be built. In Section 2.1, we give an overview of two standards we used in our work: the Service-Oriented Architecture (SOA) [92] framework and the Business Process Execution Language (BPEL) [76]. A SOA-based application is an orchestration of services offered by (possibly third-party) components written in possibly different languages. BPEL is a standard for implementing orchestrations of web services (provided by partners) by specifying an executable workflow using predefined activities.

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. In this work, we extend the approach described in Howard Fosters Ph.D. thesis [30]. This approach uses Labelled Transition Systems (LTS) [70] as the underlying formalism. In Section 2.2, we give an overview of LTS, as well as Non-deterministic Finite Automata (NFA) and Quantified Regular Expressions (QRE) – low-level action-based modelling formalisms used in this thesis. We also give an overview of the high-level behavioural specification formalisms used in this thesis: UML Sequence Diagrams (SD) [77] and the Specification Pattern System (SPS) [26]. Finally, in Section 2.3, we present Foster's BPEL to LTS translation.

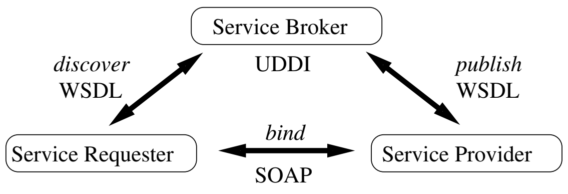


Figure 2.1: SOA infrastructure.

2.1 Web Technologies

This section presents a brief overview of the SOA and BPEL standards.

2.1.1 SOA

SOA [92] provides a general architecture for building service-based applications. SOA-based applications can dynamically discover and bind to services in order to provide aggregate services. Figure 2.1 shows the three types of partners required to build a SOA-based application: Service Brokers, Service Providers and Service Requesters. Service Providers create services, which are made available to Service Requesters through a Service Broker.

The following three standards define how SOA partners communicate:

UDDI (Universal Description, Discovery and Integration) [75]: This is an XML-based standard that provides a platform-independent manner for Service Providers to list their services and for Service Requesters to query existing services.

WSDL (Web Service Description Language) [102]: This is an XML-based language that provides a model for describing Web services.

SOAP [101]: This once stood for Simple Object Access Protocol, but this acronym has been dropped, as it was misleading. This is a protocol for exchanging XML-based messages over computer networks, normally using HTTP/HTTPS.

Service Brokers are UDDI-driven servers. Service Providers (Service Requesters) use

WSDL to describe the services they wish to publish (discover). Once a Service Requester has found a suitable service, it establishes a connection to the corresponding Service Provider, and exchanges XML messages using the SOAP protocol.

Through this architecture, Service Providers are required to make public only service interfaces. The underlying implementation is hidden; so, unless there are direct changes to a service's interface, changes in the internal logic are transparent to Service Requesters. This makes the task of service maintenance easier for Service Providers, since there is no need to worry about client dependencies with respect to internal data structures. However, since the semantics of the operations offered by a service can be modified without changing the service's interface, Service Requesters cannot assume that any services will continue to behave according to their published specifications. This means that maintenance from the Service Requester's point of view can be tricky.

2.1.2 BPEL

BPEL [76] is a XML-centric language for describing the behaviour of a business process based on its interactions with its partner services (both synchronous and asynchronous interactions are allowed). An executable BPEL process specifies how multiple service interactions coordinate to achieve a business goal, as well as the state and the logic necessary for this coordination. The BPEL standard also allows the definition of *abstract* processes, which are not executable because they are not completely specified. For example, abstract processes allow non-deterministic data assignments, while executable processes do not. Since we focus on runtime monitoring in this work, we have omitted the presentation of abstract BPEL processes from this thesis, the reader can consult [76] for details. In the rest of this work, the words "process" and "application" refer to executable BPEL processes. A process must be deployed to a BPEL engine, e.g., the IBM WebSphere Process Server [51] or ActiveBPEL [1], for execution.

BPEL also makes available various common mechanisms for dealing with business

```
<invoke operation="op1" ...
  inputVariable="inVar"
  outputVariable="outVar">
  ...
</invoke>
```

(a)

```
<if name="if_example">
  <condition>expr</condition>
  <scope name="then_branch">
    ...
  </scope>
  <else>
    <scope name="else_branch">
      ...
    </scope>
  </else>
</if>
```

(b)

```
<while name="while_example">
  <condition>expr</condition>
  <scope name="loop_body">
    ...
  </scope>
</while>
```

(c)

```
<pick name="pick_example">
  <onMessage operation="op1">
    <scope name="op1_branch">
      ...
    </scope>
  </onMessage>
  <onMessage operation="op2">
    <scope name="op2_branch">
      ...
    </scope>
  </onMessage>
</pick>
```

(d)

Figure 2.2: Basic BPEL activities: (a) <invoke>; (b) <if>; (c) <while>; and (d) <pick>.

exceptions and processing faults, like fault and termination handlers. Moreover, BPEL introduces *compensation*, a mechanism used to define how individual or composite activities within a process are to be compensated in cases where exceptions occur. The rest of this section presents an overview of the BPEL language, exemplified by the Trip Advisor System (`TAS`) introduced in Section 1.1.1.

2.1.2.1 Basic and Structural 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 non-deterministic 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.

```
<flow name="flow_example">
  <scope name="scope1">
    ...
  </scope>
  <scope name="scope2">
    ...
  </scope>
</flow>
```

(a)

```
<sequence name="seq_example">
  <scope name="scope1">
    ...
  </scope>
  <scope name="scope2">
    ...
  </scope>
</sequence>
```

(b)

```
<scope name="scope">
  <!-- scope activities
  ...
  -->
  <eventHandlers>
    ...
  </eventHandlers>
  <faultHandlers>
    ...
  </faultHandlers>
  <compensationHandler>
    ...
  </compensationHandler>
</scope>
```

(c)

Figure 2.3: Structural BPEL activities: (a) <flow>; (b) <sequence>; and (c) <scope>.

If multiple <onMessage> branches are simultaneously activated, the BPEL engine non-deterministically chooses which one should be executed. Figure 2.2 shows XML declarations of these activities (<receive> and <reply> are similar to <invoke>, and have been omitted). BPEL also provides the <empty> activity, a "no-op" activity which does nothing when executed.

The structural activities <sequence> and <flow> are used to specify sequential and parallel composition of the enclosed activities, respectively. In the case of the <flow> activity, the BPEL engine non-deterministically chooses the order in which to execute the enclosed activities. The <scope> activity is used to define named logical units of activities (with individual fault, termination and compensation handlers). Figure 2.3 shows XML declarations of these activities.

Figure 2.4 shows the workflow of the Trip Advisor System, expressed using the Eclipse BPEL Project notation [90]. 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

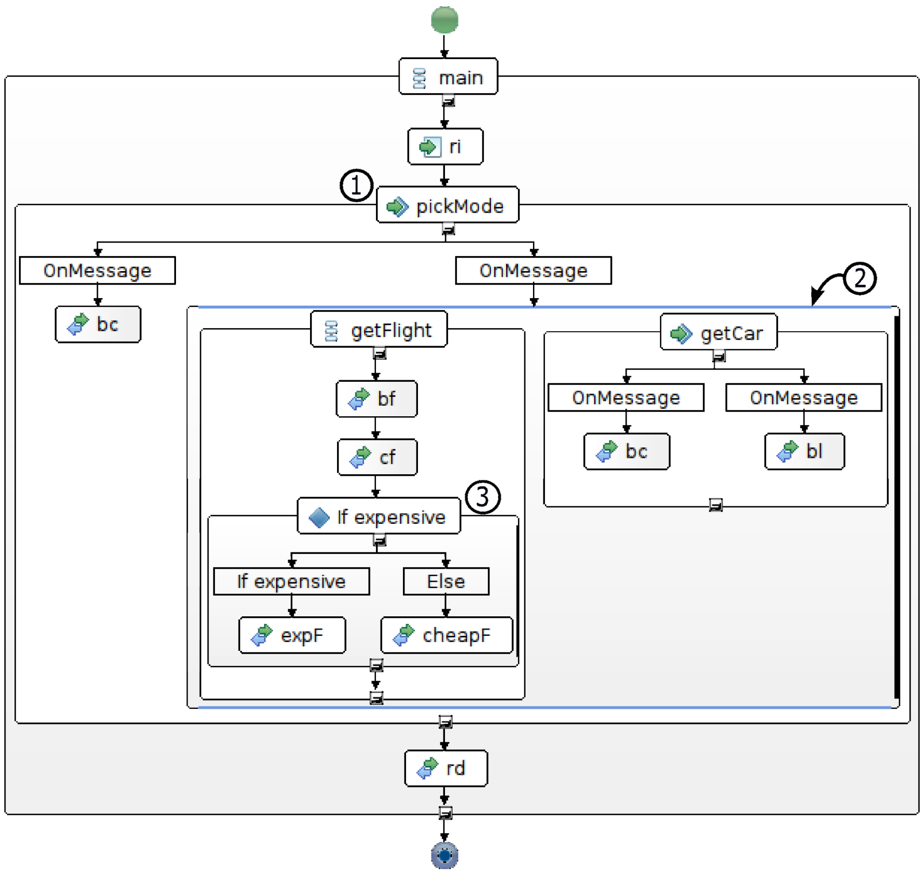


Figure 2.4: `TAS` workflow.

(expF) or cheap (cheapF). Service interactions are preceded by a symbol.

The main process is a <sequence> of activities (shown in Fig 2.5a). The workflow begins with <receive>'ing input (ri), followed by <pick>'ing (indicated by labelled ①) either the car rental (onMessage onlyCar) or the air/ground transportation combination (onMessage carAndFlight). The latter choice is modelled using a <flow> (scope enclosed in bold, blue lines —, labelled ②) since air (getFlight) and ground transportation (getCar) can be arranged independently, so they are executed in isolation. The air transportation branch books a flight (bf), then checks <if> it is expensive (cf) and finally updates the state of the system accordingly (<if> labelled ③). The ground transportation 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).

```
<sequence name="main">
  <receive operation="ri" ... />
  <pick name="pickMode">
    ...
  </pick>
  <reply operation="rd" ... />
</sequence>
```

(a)

```
<pick name="pickMode">
  <onMessage operation="onlyCar">
    <invoke operation="bc" />
  </onMessage>
  <onMessage operation="carAndFlight">
    <flow name="getCarAndFlight">
      ...
    </flow>
  </onMessage>
</pick>
```

(b)

```
<flow name="getCarAndFlight">
  <sequence name="getFlight">
    ...
  </sequence>
  <sequence name="getCar">
    ...
  </sequence>
</flow>
```

(c)

```
<if name="if expensive">
  <condition>$expensive == 1</condition>
  <invoke operation="expF" />
  <else>
    <invoke operation="cheapF" />
  </else>
</if>
```

(d)

```
<scope name="bf">
   <invoke ... operation = "bf" ... outputVariable = "flightConf" />
   <compensationHandler cost = "9">
       <invoke ... operation = "cancelF" inputVariable = "flightConf"/>
   </compensationHandler>
</scope>
```

(e)

Figure 2.5: BPEL specification of selected TAS activities: (a) the main process; (b) <pick> labelled ①; (c) <flow> labelled ②; (d) <if> labelled ③; and (e) compensation for booking a flight (bf).

Figures 2.5b, 2.5c and 2.5d show the BPEL implementation of the activities labelled ①, ② and ③, respectively.

2.1.2.2 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 2.5e: 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 $\mathsf{compensate}(a; b)$ is $\mathsf{compensate}(b); \mathsf{compensate}(a)$.

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).

While not in the BPEL standard, in this thesis we use BPEL extended with compensation costs. This extension lets application developers associate different costs to different compensations, e.g., to indicate that cancelling a flight might be significantly more expensive than cancelling 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 2.5e has been assigned a cost of 9 (out of 10), indicating that this is an expensive compensation and should be avoided if possible.

2.2 Behavioural Modelling Formalisms

This section presents a brief overview of the low- and high-level specification languages used in this thesis.

2.2.1 Low-level Specification Languages

There are two approaches to formalizing models low-level: state-based (e.g., using Kripke structures [48]) and action-based (e.g., using Labelled Transition Systems [70]). In the state-based approach, an execution of a system is viewed as a sequence of states in which every state is an assignment of values to some set of propositions. The action-based approach views an execution as a sequence of actions. The approaches are equivalent: an

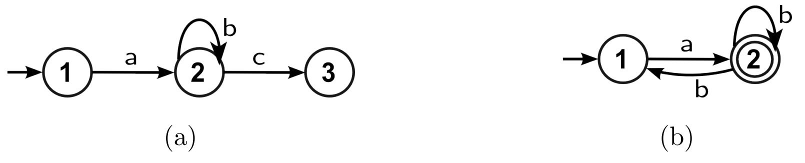


Figure 2.6: Examples of action-based formalisms: (a) an LTS and (b) an NFA.

action can be modelled as a state change, and a state can be modelled as an equivalence class of sequences of actions. Web service applications are event-based systems, so it is more natural to model these systems using action-based approaches. This section gives an overview of the different low-level action-based modelling formalisms used in this thesis.

2.2.1.1 Labelled Transitions Systems

Definition 2.1 (LTS [70]). *A* Labelled Transition System *LTS is a quadruple* (S, Σ, δ, I)*, where* S *is a set of states,* Σ *is a set of actions/labels,* $\delta \subseteq S \times \Sigma \times S$ *is a transition relation, and* $I \in S$ *is the initial state. We often use the notation* $s \xrightarrow{a} s'$ *to stand for* $(s, a, s') \in \delta$*.*

An example LTS is shown in Figure 2.6a, where:

- $S = \{1, 2, 3\}$,
- $\Sigma = \{a, b, c\}$,
- $\delta = \{(1, a, 2), (2, b, 2), (2, c, 3)\}$, and
- $I = 1$.

An *execution*, or a *trace*, of an LTS M is a sequence $\mathsf{T} = s_0 a_0 s_1 a_1 s_2 ... a_{n-1} s_n$ such that $\forall i, 0 \leq i < n$, $s_i \in S$, $a_i \in \Sigma$ and $s_i \xrightarrow{a_i} s_{i+1}$. For example, $1 \xrightarrow{a} 2 \xrightarrow{b} 2 \xrightarrow{b} 2 \xrightarrow{c} 3$ is a trace of the LTS shown in Figure 2.6a.

2.2.1.2 Non-deterministic Finite Automata

A Non-deterministic Automaton can be defined as an LTS that has a set of final states:

Definition 2.2 (NFA [47]). *A Non-deterministic Finite Automaton 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 denote by Σ^* the set of all finite traces over Σ. We say that A *accepts* a word $a_0a_1a_2...a_{n-1} \in \Sigma^*$ iff there exists an execution $s_0a_0s_1a_1s_2...a_{n-1}s_n$ of A such that $a_0 \in I$ and $s_n \in F$. The *language* of A, $\mathcal{L}(A)$, is the set of all traces accepted by A.

An example NFA is shown in Figure 2.6b, where:

- $S = \{1, 2\}$,
- $\Sigma = \{a, b\}$,
- $\delta = \{(1, a, 2), (2, b, 2), (2, b, 1)\}$,
- $I = 1$, and
- $F = \{2\}$.

Definition 2.3 (Projection "$\downarrow$"). *Let $\Sigma' \subseteq \Sigma$ be an alphabet, and $\sigma = a_0 \dots a_n$ be a word over Σ. The* projection *of σ to Σ', denoted $\sigma \downarrow_{\Sigma'}$, is obtained by replacing every a_i $(0 \leq i \leq n)$ by the silent symbol ϵ iff $a_i \notin \Sigma'$.*

Let (q, a, q') be a transition in an NFA A. We often refer to a as the label of the transition from q to q'. For an NFA A with ϵ transitions, let $\mathcal{L}(A)$ be the set of traces of A with the occurrences of ϵ removed.

States in NFAs may have several outgoing transitions on the same input symbol, or may have transitions labelled ϵ, indicating a *silent* move. *Deterministic* finite automata (DFAs) are NFAs where each state has at most one outgoing transition on each non-silent symbol. Every NFA can be converted into a DFA using the subset construction algorithm [47].

2.2.1.3 Quantified Regular Expressions

Specifying behaviour using NFAs requires the explicit definition of transition relations, which can be a tedious and error-prone process for large models. Kleene [57] defined Regular Expressions (RE) as a way to declaratively specify regular languages, and proved that RE and finite automata are equivalent. Regular expressions can be used to concisely describe the behaviour of a system, by specifying the words that the system accepts.

Definition 2.4 (Operators over languages [47]). *Let $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2$ be languages over Σ. The following expressions can be used to define new languages.*

- *$\emptyset$ denotes the empty language*
- *$\mathcal{L}_1 \cdot \mathcal{L}_2 = \{x \cdot y : x \in \mathcal{L}_1, y \in \mathcal{L}_2\}$ denotes the product of two languages*
- *$\mathcal{L}^0 =$ if $\mathcal{L} \neq \emptyset$ then $\{\epsilon\}$ else $\emptyset$*
- *$\mathcal{L}^{i+1} = \mathcal{L} \cdot \mathcal{L}^i$, where $i \geq 0$*
- *$\mathcal{L}^* = \bigcup_{i=0}^{\infty} \mathcal{L}^i$*

Definition 2.5 (Regular Expression [57]). *The regular expressions over Σ and the languages they denote are defined inductively as follows:*

- *$\emptyset$ is a regular expression that denotes the empty set;*
- *ϵ is a regular expression that denotes the set $\{\epsilon\}$;*
- *a is a regular expression that denotes $\{a\}$, where $a \in \Sigma$.*

If J and K are regular expressions that represent $\mathcal{L}(J)$ and $\mathcal{L}(K)$, then the following are also regular expressions:

- *$J|K$ (alternation) represents $\mathcal{L}(J) \cup \mathcal{L}(K)$;*
- *$J \cdot K$ (concatenation) represents $\mathcal{L}(J) \cdot \mathcal{L}(K)$;*

- *J^* (Kleene star) represents $\mathcal{L}(J)^*$.*

By convention, regular expressions can be written with fewer parentheses by establishing the operator precedence ($* > \cdot > |$). Parenthesis can be used to override precedence. For example, the language of the monitor shown in Figure 2.6b is the regular expression $(a \cdot b^*)^*$.

Quantified Regular Expressions (QRE) [78] are regular expressions with limited quantifiers ($Q = \{no, all\}$). These quantifiers make specification writing easier, since one QRE represents a set of REs.

Definition 2.6 (Quantified Regular Expression [78]). *A QRE is a 3-tuple $P = (\Sigma, q, R)$, where Σ is the alphabet, $q \in Q$ and R is the regular expression to be satisfied.*

The *no* quantifier is used to indicate that R specifies an undesirable behaviour – P is satisfied if no traces of the system belong to $\mathcal{L}(R)$. The *all* quantifier indicates that R specifies a desired behaviour – P is satisfied if all traces of the system belong to $\mathcal{L}(R)$. For example, the QRE $(\{a, b\}, all, (a \cdot b^*)^*)$ holds on the NFA shown in Figure 2.6b but the QRE $(\{a, b\}, all, (a^*))$ does not.

Since regular expressions are closed under complement, the expressive power of *all* and *no* QREs are equivalent. In this thesis, when we omit the alphabet and quantifier of a QRE, then Σ is the alphabet of the system being modelled, and the quantifier is *all*.

Definition 2.7 (Additional RE operators). *If J and K are regular expressions, then the following are also regular expressions:*

- *if $p, q, r \in \Sigma$, then $[-p, q, r]$ (exclusion) denotes the expression that matches any symbol except p, q and r;*
- *if $p \in \Sigma$, then $p?$ denotes zero or one instances of expression p;*
- *if $k \in \mathbb{N}$ and $p \in \Sigma$, then p^k denotes k instances of p.*

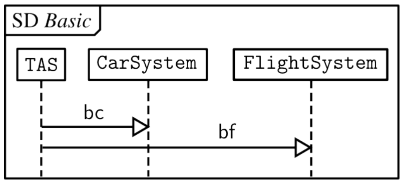


Figure 2.7: An SD describing a scenario of the `TAS` example.

Standard algorithms [40, 69, 2] can be used to turn REs and QREs into NFAs, as well as make their representation minimal and deterministic.

2.2.2 High-level Specification Languages

The low-level behavioural specification languages presented in the previous section are easy to analyze, but writing complete and consistent specifications in these languages can be a tedious and error-prone process, especially in the case of large systems. In this section, we give an overview of the high-level specification languages that are used in this thesis.

2.2.2.1 UML 2.0 Sequence Diagrams

UML 2.0 Sequence Diagrams [77] are a popular formalism for modelling behavioural scenarios by describing sequences of messages communicated between different objects over time. An example Sequence Diagram describing a scenario of the `TAS` system is shown in Figure 2.7. Sequence Diagrams have two dimensions: vertical, representing time, and horizontal, representing objects. Each object is illustrated by a rectangle with a vertical dashed line, called a *lifeline*. Lifelines are connected by horizontal arrows denoting messages that are sent from one object to another, synchronously (solid arrowhead) or asynchronously (open arrowhead). We refer to Sequence Diagrams with these features as *basic*.

In this thesis, we adopt the automata-theoretic approach of Alur and Yannakakis [4] for formalizing Basic SDs. In the rest of this section, we provide a formal description of semantics of Basic SDs.

Definition 2.8 (Basic SDs [4]). *A Basic SD S is a tuple ($\mathcal{I}$, E, f, $\mathcal{O}$), where*

- *$\mathcal{I}$ is a finite set of objects.*
- *E is a finite set of event occurrences that is partitioned into* send events*, denoted by $!E$, and* receive events*, denoted by $?E$. The set of events sent and received by an object $i \in \mathcal{I}$ is denoted by E_i.*
- *f: $!E \rightarrow ?E$ is a bijective mapping that associates each send event e with a unique receive event $f(e)$, and each receive event e' with a unique send event $f^{-1}(e')$.*
- *$\mathcal{O}$ is a set of total order relations $<_i$ defined over the events E_i for every object i. It corresponds to the order in which the events are physically displayed along the lifeline of an object i.*

Definition 2.9 (Partial Order [4]). *Let $S = (\mathcal{I}, E, f, \mathcal{O})$ be a Basic SD. We define a partial order relation $<$ over E as follows:*

$$(<) \ = \ [(\cup_{i \in \mathcal{I}}(<_i)) \ \cup \ (\{(s, f(s)) \mid s \in \, !E\})]^*$$

The SD shown in Figure 2.7 is a Basic SD, where:

- $\mathcal{I} = \{\texttt{TAS}, \texttt{CarSystem}, \texttt{FlightSystem}\}$,
- $E = \{!\mathsf{bc}, ?\mathsf{bc}, !\mathsf{bf}, ?\mathsf{bf}\}$,
- the total order $<_{\texttt{TAS}}$ for the object $\texttt{TAS}$ is $\{!\mathsf{bc} <_{\texttt{TAS}} \ !\mathsf{bf}\}$, and
- the partial order $<$ associated with the entire scenario is $\{!\mathsf{bc} < \ !\mathsf{bf}, !\mathsf{bc} < \ ?\mathsf{bc}, !\mathsf{bf} < \ ?\mathsf{bf}\}$.

This partial order assumes that messages are communicated asynchronously. Partial order for synchronous communication is a subset of the above because of synchronization.

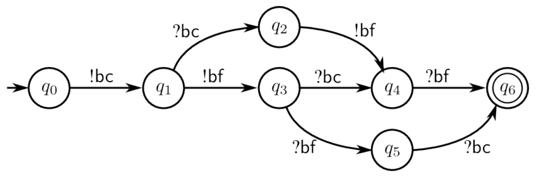


Figure 2.8: NFA corresponding to the SD in Figure 2.7.

In the rest of this thesis, we assume that messages are passed asynchronously. Also, without loss of generality, we assume that all event labels are unique.

We define the semantics of Basic SDs by translating them into their equivalent NFAs. Intuitively, an NFA A_S is equivalent to a Basic SD S iff A_S accepts exactly the set of traces that can be generated by S, i.e., those traces that respect the partial order of S. Therefore, translation of S to A_S reduces to the translation of the underlying partial order of S to A_S. The algorithm for translating partial orders to NFAs, proposed by [4], is as follows. Given a partial order $<$ over E, let *cut* c be a subset of E that is closed with respect to $<$, i.e., if $e \in c$ and $e' < e$, then $e' \in c$. Since all the events of a single process are linearly ordered, a cut can be specified by a tuple that gives the maximal event of each process. The set of all possible cuts associated with the partial order of a Basic SD generates the state space of its corresponding NFA. The empty cut is the initial state, and the cut with all the events is the final state. There is a transition labelled e from cut c to cut d, if the cut d equals the cut c plus the single event e.

Theorem 2.1. *A Basic SD S = ($\mathcal{I}$, E, f, $\mathcal{O}$) is semantically equivalent to an NFA A_S = (Q, Σ, δ, Q_0, F), where Σ is equal to E, Q is the set of all cuts, Q_0 is the empty cut, F is the maximal cut including all of the events, and δ allows a transition from a cut c to a cut d on an event $e \in E$ iff $d = c \cup \{e\}$.*

The above theorem follows from [4]. Listing 2.1 shows an implementation of this translation algorithm.

Since both the empty and the maximal cuts are unique, Q_0 and F consist of only one state each. The set of cuts obtained by unwinding the underlying partial order of the SD

Listing 2.1: Function that implements the Basic SD to NFA translation algorithm

```
sub SD_to_NFA (I, E, f, O){
        # par_order is a list of pairs, where a < b is [a, b]
        par_order = compute_partial_order(I, E, f, O)
        # events is a hash: events.keys() = E, and events[e] is the list of
        #    events that must be included in a cut that includes event e
        events = {}
        for e in E:
                events[e] = [ pair[1] for pair in par_order if pair[0] == e]
        cuts = []
        power = powerset(E)
        # check which subsets of E are valid cuts
        for p in power:
                if p == (): # empty cut
                        cuts.append(p)
                        Q0 = p
                        break
                if len(p) == len(E) # maximal cut
                        cuts.append(p)
                        F = p
                        break
                # create valid cut using the elements of p
                s = set()
                for e in p:
                        s.add(set(events[e]))
                # test equivalence and keep valid cuts
                if s >= p and p <= s:
                        cuts.append(p)
        delta = [(c, e, d) for c, d in cuts if d-c == (e)]
        Q = cuts
        Sigma = E
        return NFA.NFA(Q, Sigma, delta, Q0, F)
}
```

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 specified 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 2.1: SPS patterns.

in Figure 2.7 is

$$\{\langle\rangle, \langle \mathsf{!bc}\rangle, \langle \mathsf{!bc}, \mathsf{?bc}\rangle, \langle \mathsf{!bc}, \mathsf{!bf}\rangle, \langle \mathsf{!bc}, \mathsf{?bc}, \mathsf{!bf}\rangle,$$
$$\langle \mathsf{!bc}, \mathsf{!bf}, \mathsf{?bf}\rangle, \langle \mathsf{!bc}, \mathsf{!bf}, \mathsf{?bc}\rangle, \langle \mathsf{!bc}, \mathsf{?bc}, \mathsf{!bf}, \mathsf{?bf}\rangle\}.$$

Note that the number of states of the corresponding automaton in Figure 2.8 is less than the number of the above cuts, because we reduced the states with the identical outgoing transitions to a single state.

2.2.2.2 The Specification Pattern System

The *Specification Pattern System* (SPS), proposed by Dwyer et al. [26], 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) [82], computational tree logic (CTL) [20], quantified regular expressions (QRE) [78] and other state-based and event-based formalisms. This system has been advocated as a standard tool for measuring the practical usefulness and expressive power of specification languages, e.g., [6, 105].

The property patterns are organized into a hierarchy based on the kinds of system behaviours they describe (see Figure 2.9a): **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 2.1.

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** and **After-Until**. Definitions of these are given in Table 2.2 and pictorially described in Figure 2.9b – the shaded intervals indicate the portions of an execution where the specified property must hold. The action-based modelling formalisms used in this thesis do not allow the simultaneous occurrence of two events, so these scopes should be interpreted as open on both ends. Note that if a scope is specified but does not occur during execution, then the specification is vacuously true, since the execution does not contain any regions of interest. On the other hand, a pattern instance without a scope has an implicit **Global** scope.

For example, the specification "P occurs between Q and R" is true if:

1. each event Q is followed at some point by the occurrence of an event R (e.g., at times t and $t + k$, respectively; this defines a region of interest) and at least one P event occurs in each region of interest, i.e., P occurs at some time t', where $t < t' < t + k$, for each $t, t + k$ pair, or

2. Q occurs, but without a matching R. In this case, the pattern instance holds because there are no regions of interest.

For example, consider a property of a queue that says that there should be a `dequeue` event between every `enqueue` and `empty`. This is the **Existence** pattern, with the **Between** scope. Looking up the QRE formalization of this pattern/scope combination from

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 2.2: SPS scopes.

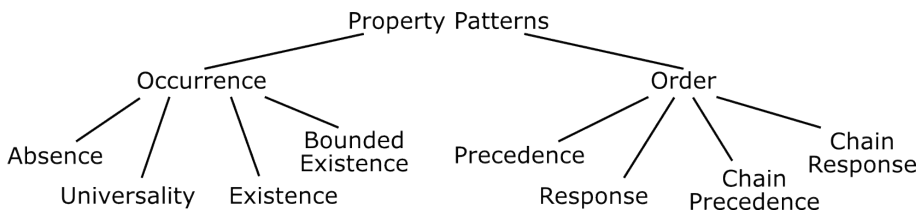
the catalogue and substituting our event names, we obtain the formula

$$\big([-\mathsf{enqueue}] * \cdot \mathsf{enqueue} \cdot [-\mathsf{dequeue}, \mathsf{empty}] * \cdot \mathsf{dequeue} \cdot [-\mathsf{empty}] * \cdot \mathsf{empty}\big) * \cdot$$
$$[-\mathsf{enqueue}] * \cdot (\mathsf{enqueue} \cdot [-\mathsf{empty}]*)?$$

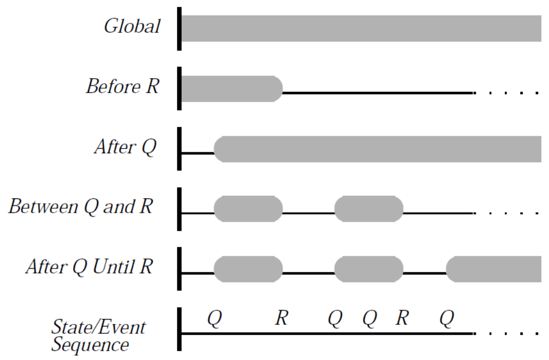
The first line of this formula checks that, once an enqueue has occurred, at least one dequeue event occurs before an empty event. In other words, if no dequeue events are sent between an enqueue and empty event, the QRE does not hold on the execution trace. The second part encodes the possibility that the scope never occurs: the property also holds if empty never occurs after enqueue. For reference, we have included a summary of the QRE pattern and scope encodings in Appendix A.

2.3 Formalizing BPEL

In [30, 31], Foster specified how to map all BPEL 1.1 activities into LTS. For example, Figure 2.10b shows the translation of the <invoke> activity bf defined in Figure 2.10a, which returns a confirmation number. The activity is a sequence of two transitions: the actual service invocation (invoke_bf) and its return (receive_bf). Foster's translation includes partner, activity and variable names in the labels, in order to include traceability information, but we omit these in this thesis for simplicity.



(a)

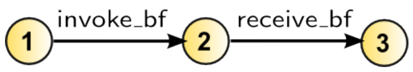


(b)

Figure 2.9: Specification property system: (a) the pattern hierarchy and (b) pattern scopes.

```
<scope name="bf">
   <invoke ... operation = "bf" ... outputVariable = "flightConf" />
   <compensationHandler cost = "9">
       <invoke ... operation = "cancelF" inputVariable = "flightConf"/>
   </compensationHandler>
</scope>
```

(a)



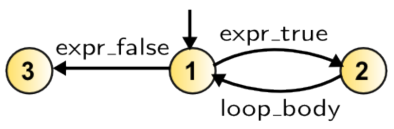
(b)

Figure 2.10: Book flight <invoke> activity (bf) (a) BPEL declaration (with compensation); and (b) the LTS translation of <invoke> activity (with output parameter).

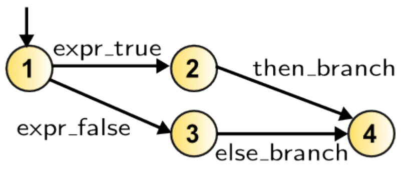
Note that BPEL processes may have multiple <receive> activities. For such processes, the BPEL engine non-deterministically chooses a creation point on execution. In his thesis, Foster assumes that processes have only one creation point, the first <receive> that appears in the process definition, and we make the same assumption in this thesis.

Conditional activities like <while> and <if> are represented as states with two outgoing transitions, one for each valuation of the activity condition. The LTSs for these

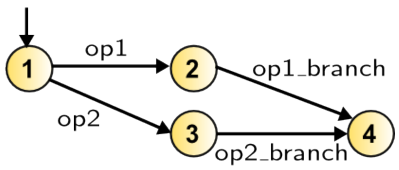
```
<while name="while">
  <condition>expr</condition>
  <scope name="loop_body">
    ...
  </scope>
</while>
```



```
<if name="if">
  <condition>expr</condition>
  <scope name="then_branch">
    ...
  </scope>
  <else>
    <scope name="else_branch">
      ...
    </scope>
  </else>
</if>
```

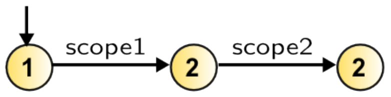


```
<pick name="pick">
  <onMessage operation="op1">
    <scope name="op1_branch">
      ...
    </scope>
  </onMessage>
  <onMessage operation="op2">
    <scope name="op2_branch">
      ...
    </scope>
  </onMessage>
</pick>
```

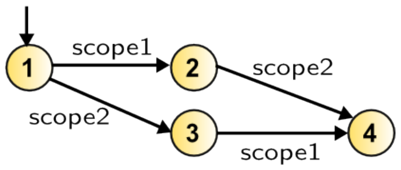


(a)

```
<sequence name="seq">
  <scope name="scope1">
    ...
  </scope>
  <scope name="scope2">
    ...
  </scope>
</sequence>
```



```
<flow name="flow">
  <scope name="scope1">
    ...
  </scope>
  <scope name="scope2">
    ...
  </scope>
</flow>
```



(b)

Figure 2.11: (a) BPEL conditional activities and their corresponding LTSs; (b) BPEL structural activities and their corresponding LTSs.

two activities are shown in Figure 2.11a. 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 2.11a. If multiple <onMessage> branches are simultaneously activated, the BPEL engine non-deterministically chooses which one should be

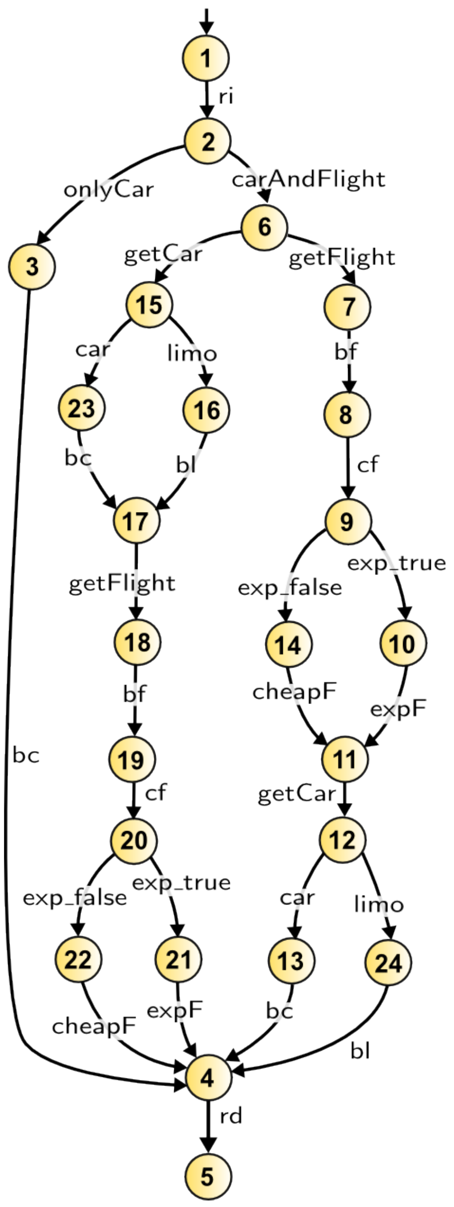


Figure 2.12: LTS $\mathsf{L}(\texttt{TAS})$.

executed. <sequence> and <flow> activities result in the sequential and the parallel composition of the enclosed activities, respectively (see Figure 2.11b). In the case of the <flow> activity, the BPEL engine non-deterministically chooses the order in which to execute the branches.

Thus, formally, we are going from a BPEL program B to its LTS translation $\mathsf{L}(\mathsf{B})$. The set of labels Σ of $\mathsf{L}(\mathsf{B})$ is derived from the possible events in B: service invocations and returns, <onMessage> events, <scope> entries, and condition valuations. For example, Figure 2.12 shows `L(TAS)`. This LTS has 24 states and 29 transitions.

Chapter 3

Specifying Correctness

The goal of this thesis is the creation of an industrial-strength monitoring framework. The first milestone for achieving this goal is the creation of an adequate property specification language. This language must be expressive enough to capture the distributed, interactive, and message-driven nature of web service applications, but must also be amenable to efficient runtime monitoring.

In this chapter, we propose Web Sequence Diagrams (W-SD) [89], a language that, we feel, meets these criteria. This language is a subset of UML 2.0 Sequence Diagrams (SD), a feature-rich language without a formal semantics. In Section 3.1, we describe the syntax of the subset of UML 2.0 sequence diagrams used for expressing properties of web service conversations. We describe the semantics of our chosen subset of SDs and show how to translate it into automata for runtime monitoring in Section 3.2. In Section 3.3, we show that this language is sufficiently expressive to capture a wide variety of frequently used properties, captured and catalogued in the Specification Pattern System (SPS) [28]. In this section, we also show how to use the W-SD templates to specify properties of a new application, the Loan Application System. Finally, in Section 3.5, we report on related work studying UML 2.0 Sequence Diagrams as a specification language.

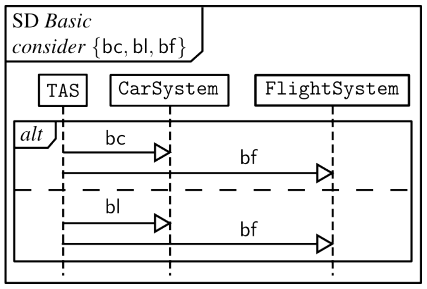


Figure 3.1: A W-SD describing a scenario of the `TAS` example.

3.1 Web Sequence Diagrams

As discussed in Chapter 2, Section 2.2.2.1, UML 2.0 Sequence Diagrams (SD) [77] can be used to describe complex scenarios between different objects over time. Properties of web service applications can be specified using SDs: service providers are modelled as objects and service invocations and other system events are modelled as messages between the corresponding objects. Basic SDs only contain messages between objects (see Definition 2.8), and can be augmented by a number of operators to capture more sophisticated scenarios. We describe some of these operators below:

- **Compositional operators:** Operators *parallel (par)* and *alternatives (alt)* are used to compute intersection and union of two SDs, respectively. The operator *loop* is used for repeating the scenario described by an SD multiple times, and *opt* – for denoting an optional scenario, equivalent to *alt* with only one argument.

- **Alphabet changing operators:** Operators *consider* and *ignore* are used for modifying the communicating alphabet of SDs.

- **Critical operator:** The *critical* operator is used to ensure atomicity of the enclosed sequence.

- **Assertion and negation operators:** Operators *assert* and *negate* allow users to express mandatory and forbidden system scenarios, respectively.

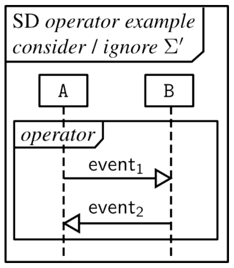


Figure 3.2: Visual representation of SD operators.

- **Interaction use operator:** SDs can be shared by reference, using the *ref* operator. This is a shorthand for copying the contents of the referred SD where the *ref* operator occurs, and is a new feature in UML 2.0.

Figure 3.2 shows how these operators are specified on an SD. Visually, all operators except the alphabet changing operators are represented as boxes that enclose the operator's arguments (alphabet changing operators appear below the SD name). The diagram in Figure 3.1 shows an SD that uses the *alt* operator to describe two alternative scenarios of the `TAS` system: in the top scenario, `TAS` first sends a car booking request (bc) to `CarSystem`, and then a flight booking request (bf) to `FlightSystem`. In the bottom scenario, `TAS` first tries to book a limo (bl), and then book a flight.

The grammar for our language, Web Sequence Diagrams (W-SD), is given in Figure 3.3 where *BasicSD*, *par*, *alt*, *loop*, *critical*, *opt*, *negate*, *assert*, *consider*, *ignore* and *ref* are terminal symbols, and E is a set of SD messages. Since operators *consider* and *ignore* change the communicating alphabet of SDs, they take a set E of messages as an input argument. Note that every event that appears in a W-SD must be either sent or received by the object that represents the application being analyzed, as messages between partner services cannot be seen by the orchestrating web service.

Basic SDs are the building blocks of our language. The *critical*, alphabet changing, interaction use, *assert*, and compositional operators, except for *par*, can be intermixed and applied any number of times to Basic SDs. The use of *negate* and *par* operators, however,

$$
\begin{array}{rcl}
\texttt{SD} & ::= & \mathit{BasicSD} \mid \texttt{unaryOp SD} \mid \texttt{SD}\ \mathit{alt}\ \texttt{SD} \mid \mathit{negate}\ \texttt{NotAssertedSD} \mid \\
 & & \texttt{NotAssertedSD}\ \mathit{par}\ \texttt{NotAssertedSD} \mid \mathit{assert}\ \texttt{SD} \\
\texttt{NotAssertedSD} & ::= & \mathit{BasicSD} \mid \texttt{unaryOp NotAssertedSD} \mid \mathit{negate}\ \texttt{NotAssertedSD} \mid \\
 & & \texttt{NotAssertedSD}\ \mathit{alt}\ \texttt{NotAssertedSD} \mid \\
 & & \texttt{NotAssertedSD}\ \mathit{par}\ \texttt{NotAssertedSD} \\
\texttt{unaryOp} & ::= & \mathit{consider}_E \mid \mathit{ignore}_E \mid \mathit{loop} \mid \mathit{critical} \mid \mathit{opt} \mid \mathit{ref}
\end{array}
$$

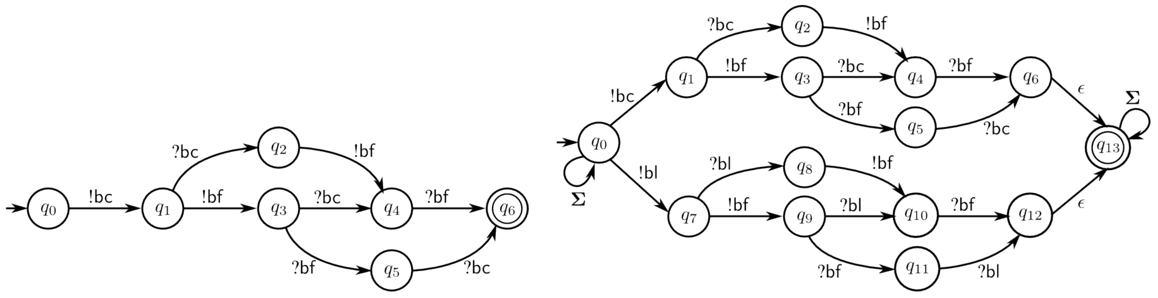
Figure 3.3: Grammar of the W-SD language.

is restricted to sequence diagrams which do not use an *assert* operator. We discuss this assumption and the rationale behind it in Section 3.2.4.2 and show in Section 3.3 that even with this restriction, the resulting language remains very expressive.

Note that we often need to express complementation of an individual message or a set of messages appearing on the same arrow. The *negate* operator is unsuitable for complementing sets because it captures negative sequences of messages rather than set complementation. Instead, we use the *message complementation* operator, originally introduced in the Property Sequence Charts (PSC) language [6]. We denote the complement of a message m by $\neg m$ and define it as the set of all messages that are potentially exchanged between objects of the system except for m.

3.2 Formalizing Sequence Diagrams

In this section, we provide a formal description of semantics of W-SD. UML 2.0 SDs is a very expressive language, without formal semantics. We adopt the automata-theoretic approach of Alur and Yannakakis [4], where a Basic SD is transformed into a semantically equivalent NFA (see Chapter 2, Theorem 2.1). Below we define the semantics of the W-



(a) (b)

Figure 3.4: NFA corresponding to: (a) the first argument of the *alt* operator of the W-SD in Figure 3.1; and (b) the W-SD in Figure 3.1.

SD operators, which are given in terms of how these operators combine or affect the NFAs corresponding to their SD arguments.

3.2.1 Compositional operators

The semantics of the compositional operators can be given in terms of the standard operations defined on NFAs (e.g., see [47]). In particular,

- *par* corresponds to the parallel composition operator or the intersection operator over NFA;
- *alt* corresponds to the union operator;
- *loop* corresponds to the Kleene star operator.

The theorem below, which follows from Theorem 2.1 and [47], shows that the set of NFAs associated with SDs is closed under the compositional operators.

Theorem 3.1. *Let S, S_1 and S_2 be SDs, and let $S = S_1$ op S_2, where op is a compositional operator. Then, $A_S = A_{S_1}$ op A_{S_2}.*

For example, the automaton in Figure 3.4b corresponds to the sequence diagram in Figure 3.1. This automaton is obtained by computing the union of the two Basic SDs corresponding to the two alternative scenarios of the W-SD in Figure 3.1: the NFA equivalent to the first argument of the *alt* operator is shown in Figure 3.4a, and its construction is discussed in Chapter 2, Section 2.2.2.1. The automaton corresponding to the second argument is constructed in a similar fashion and is not shown here.

Note that we have also added a self-loop to the initial state of the automaton in Figure 3.4b, labelled with the underlying alphabet (Σ) of the W-SD in Figure 3.1. This self-loop allows the automaton to guess when the scenario specified by the W-SD begins.

3.2.2 Alphabet changing operators

Operators *consider* and *ignore* are used to change the set of communicating alphabet of an SD. Both of them receive an SD S and a set of events E as input, but *consider* adds the elements in E to the set of events of S, whereas *ignore* removes the elements in E from the set of events of S. Formally, let S and S' be SDs, E be a set of events, and let $A_S = (\Sigma, Q, \delta, \{q_0\}, F)$ be the automaton associated with S. For $S' = \mathit{consider}_E S$, $A_{S'} = (\Sigma \cup E, Q, \delta, \{q_0\}, F)$, and for $S' = \mathit{ignore}_E S$, $A_{S'} = (\Sigma \setminus E, Q, \delta', \{q_0\}, F)$, where

$$\delta' = \big(\delta \cap (Q \times (\Sigma \setminus E) \times Q)\big) \cup \{(q, \epsilon, q') \mid \exists \sigma \in E \cdot (q, \sigma, q') \in \delta\}$$

It is easy to see that the set of NFAs associated with SDs is closed under the operators *consider* and *ignore* as well.

Recall that any missing transition at a state leads to an error state. Increasing the input alphabet Σ of A_S without changing the transition relation δ means that more execution traces end up in the error state, while shrinking the input alphabet without changing the transition relation means that more execution traces are accepted. For example, the *consider* operator in Figure 3.1 extends the underlying alphabet, Σ, of the automaton in Figure 3.4b from $\{\mathsf{!bc}, \mathsf{?bc}, \mathsf{!bf}, \mathsf{?bf}\}$ to $\{\mathsf{!bc}, \mathsf{?bc}, \mathsf{!bf}, \mathsf{?bf}, \mathsf{!bl}, \mathsf{?bl}\}$.

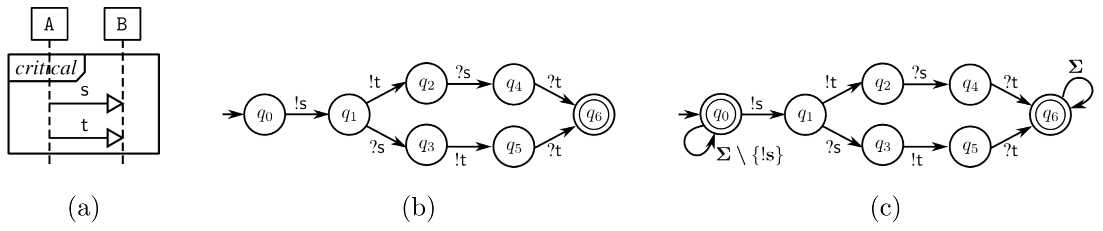


Figure 3.5: (a) A basic SD enclosed by a *critical* operator and its corresponding NFAs: (b) before applying *critical*; (c) after applying *critical*.

3.2.3 Critical operator

A critical region in a sequence diagram can be specified using the *critical* operator. A critical region means that the scenarios of the region cannot be interleaved by other messages and thus should be treated atomically. We formalize the semantics of this operator as follows: if the first message of the critical region is observed, then the rest of the behaviour must be observed as well, without seeing any intermediate messages.

Let S be an SD enclosed within a *critical* operator, and let A_S be the automaton for S. The automaton for *critical* S is obtained by adding a self-loop to every initial state of A_S labelled by $\Sigma \setminus \{e \mid \exists q_0 \in I \cdot q_0$ has an outgoing transition on $e\}$. This self-loop transition at the initial state allows the automaton to wait for a satisfying run to begin. The initial state also becomes final.

For a sequence enclosed by a critical operator, once the first symbol of the sequence has been seen, the entire sequence should be seen as well. For this reason, the self-loop at the initial state of an automaton corresponding to a critical region is labelled by Σ minus the initial symbols of the expected sequences. For example, Figure 3.5a shows a sequence diagram with a critical operator, and Figure 3.5c – its corresponding automaton (Figure 3.5b shows the automaton before the *critical* operator is applied). Similar to the automaton in Figure 3.4b, we have added a self-loop to the initial state of the automaton in Figure 3.5c to allow this automaton to guess when the scenario of interest begins.

3.2.4 Assertion and negation operators

The *negate* operator provides a mechanism for specifying undesirable (negative) scenarios, and the *assert* operator allows us to specify desirable (positive) scenarios. Recall the two requirements of the TAS system introduced in Chapter 1, Section 1.1.1: the system is to make sure the customer has the transportation needed to get to her destination (this is a desired behaviour 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 behaviour which we refer to as P_2). The *negate* operator can be used to express safety properties like P_2, and the *assert* operator – finitary liveness properties like P_1.

Various formal treatments of the semantics of the *assert* and *negate* operators are given in the literature, e.g., [44, 41, 91]. These operators have a rich expressive power, and yet their arbitrary combinations are not well understood. In particular, it is unclear whether negating an asserted scenario should mean that this scenario is not required to occur or that its negation has to occur. In this section, we define the semantics of *assert* and *negate* operators in terms of NFAs. Our formalization allows us to arbitrarily combine these operators as long as we never attempt to apply a *negate* operator to a sequence diagram containing an *assert*ed fragment.

3.2.4.1 The negate operator

As mentioned above, *negate* allows us to express safety properties. By applying *negate* to an SD S, we indicate that the scenario represented by S is forbidden, and therefore, a safe system should never produce it [41]. For example, consider Figure 3.6a which shows a W-SD corresponding to the safety property P_2. If the user picked air and ground transportation, TAS attempts to book a flight and a car/limousine. A limousine should never be booked (bl) after an expensive flight has been booked (expF), and vice versa. So, there are two forbidden sequences of events: $\mathsf{expF} \cdot \mathsf{bl}$ and $\mathsf{bl} \cdot \mathsf{expF}$; and this property is expressed in W-SD by applying a *negate* operator to an *alt* operator over these two

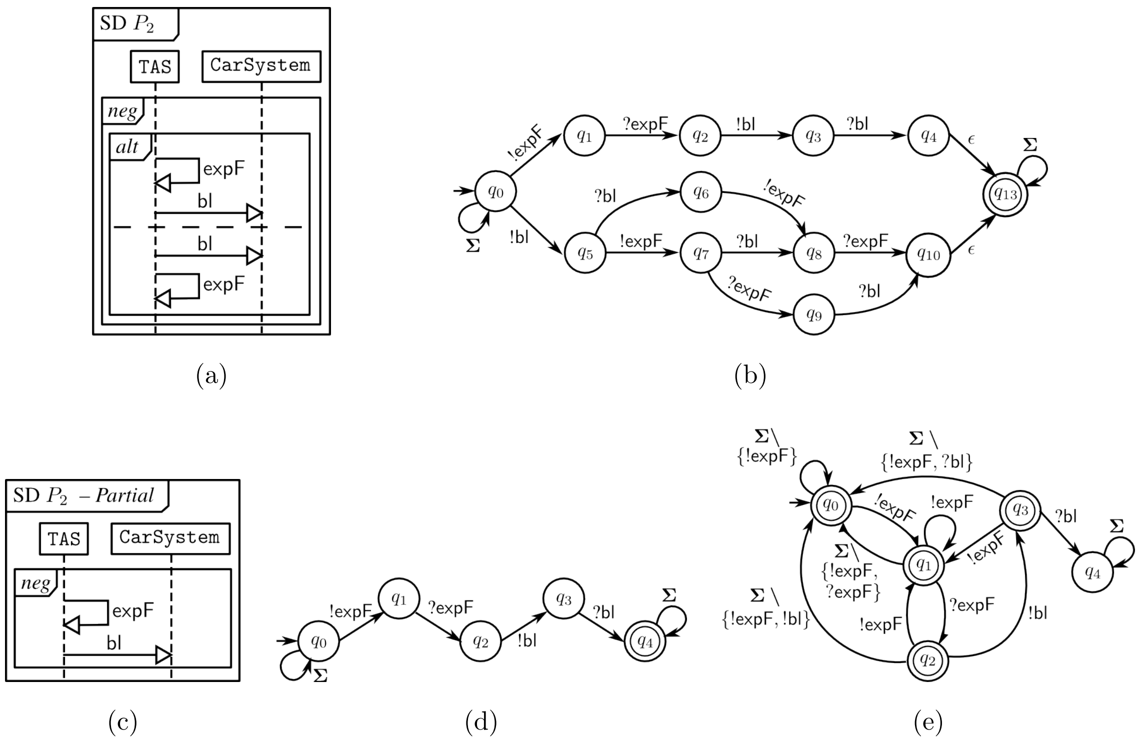

SD P_2
TAS
CarSystem
neg
alt
expF
bl
bl
expF
(a)
!expF
?expF
!bl
?bl
ϵ
Σ
q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10} q_{13}
(b)
SD P_2 – Partial
TAS
CarSystem
neg
expF
bl
(c)
!expF ?expF !bl ?bl Σ
(d)
$\Sigma \setminus$ {!expF}
$\Sigma \setminus$ {!expF, ?bl}
$\Sigma \setminus$ {!expF, ?expF}
$\Sigma \setminus$ {!expF, !bl}
(e)

Figure 3.6: (a) A W-SD describing P_2 and (b) its corresponding NFA before applying *negate*; (c) A W-SD describing a part of P_2 and its corresponding NFAs: (d) before applying *negate*; and (e) after determinization and complementation.

forbidden event sequences. The resulting W-SD is shown in Figure 3.6a.

The *negate* operator over SDs is equivalent to the complementation operator of NFA. Given an SD S and its corresponding automaton A_S, we first add a self-loop transition labelled Σ, i.e., the underlying alphabet of S, to the initial state of A_S in order to enable A_S to guess when a satisfying run begins. Note that after adding this self-loop, A_S becomes non-deterministic. To obtain the automaton for the negated SD, we need to first determinize A_S, and then complement the result.

For example, the automaton corresponding to the W-SD in Figure 3.6a, after adding the self-loop and before complementation, is shown in Figure 3.6b. We do not show the complemented automaton for P_2, which has 10 states and 33 transitions. Instead, we show the automata corresponding to a simplified version of P_2, that only includes one of the arguments of the *alt* operator. The simplified W-SD is shown in Figure 3.6c, and

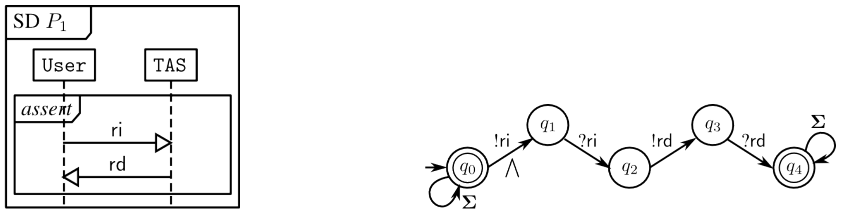
Figures 3.6d and 3.6e show the corresponding NFAs: before and after complementation, respectively.

Note that since the sequence S is nonempty, the initial state of the complement of A_S is always accepting, and hence, the empty string is always in the language of the complement of A_S. This is expected because the negate operator holds (1) when the negative scenario S does not completely occur, and (2) when no messages at all are exchanged.

3.2.4.2 The assert operator

The meaning of the *assert* operator is given by the UML standard as follows [77], "*the sequences of the operand are the only valid continuations. All other continuations result in invalid behaviour*". This interpretation has been formalized in different ways [44, 41]. The one that we have adopted is that of [44] which is described as follows: given an asserted behaviour $\sigma = \sigma_0 \ldots \sigma_n$ and a system behaviour σ', every occurrence of σ_0 in σ' should be followed by the rest of σ. Thus, an SD with an *assert* is interpreted universally: "for every run, once it satisfies the start of the sequence, it must complete the sequence before termination". Note that the difference between *assert* and *critical* is that the former checks all possible suffixes of the input run to probe the sequence, whereas the latter only checks the first occurrence of its sequence.

Harel and Maoz [44] use *alternating* automata with *universal* initial states to capture this meaning of *assert.* Such automata accept a trace if *all* of the runs emanating from their initial states are accepting. NFA, however, accept a trace when *there exists* an accepting run emanating from the initial state. Rather than moving outside NFA (and thus complicating the monitoring framework), we chose to reinterpret the acceptance for the *assert* operator instead: an NFA for an asserted trace σ checks all suffixes of the system traces, and if one is not accepted, a failure is reported. This "universal" treatment is given to the entire sequence diagram, not just the part containing *assert.* This



(a) (b)

Figure 3.7: (a) A W-SD describing P_1; and (b) its corresponding NFA after applying *assert*.

works correctly as long as such NFAs are not complemented or composed (in parallel) – the negation and parallel composition operators over automata with universally interpreted acceptance are different from those operators of NFA. While negation and parallel composition operators for NFA are computed via subset construction and cross-product, respectively, these operators for the alternating automata simply convert universal states into existential or add an additional universal state, respectively [100]. Thus, we restrict the application of *negate* and *par* to SDs that do not contain an *assert*, as described in Section 3.1.

Since alternating automata can be converted into NFA with a possibly exponential blow-up in size, we could have translated the *assert* operator directly into NFA. However, we chose not to do it to preserve the succinctness and relatively small size of our monitoring automata.

Given the above discussion, the translation of *assert* operator is straightforward: After deriving the NFA A_S for SD S and adding a self-loop labelled Σ at its initial state, the automaton for *assert* S is obtained by interpreting the initial state as universal (we follow the notation of [44], denoting this state with a "$\wedge$") and making it accepting. For example, the W-SD in Figure 3.7a describes the liveness property P_1 – the desirable scenario $\mathsf{ri} \cdot \mathsf{rd}$ is enclosed in the scope of an *assert* operator. Figure 3.7b shows the automaton corresponding to this W-SD.

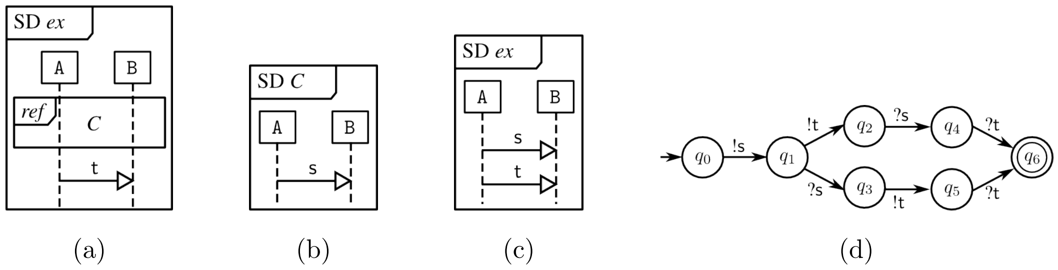


Figure 3.8: (a) An SD which references SD C; (b) SD C; (c) SD *ex* after copying the content of SD C; and (d) its corresponding NFA.

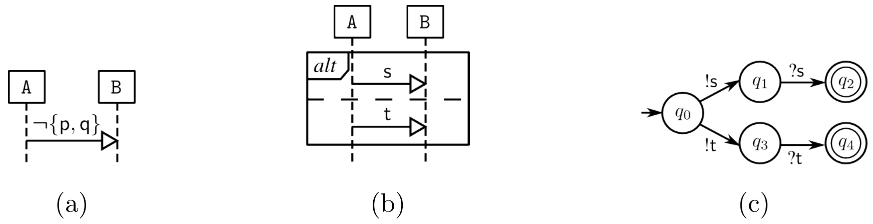


Figure 3.9: (a) An SD with message complementation; (b) the same SD after eliminating the *complement* operator if its underlying alphabet Σ is $\{\mathtt{p}, \mathtt{q}, \mathtt{s}, \mathtt{t}\}$; and (c) its corresponding NFA.

3.2.5 Interaction use operator

The *ref* operator is used for referring to an SD fragment from within another SD. Our treatment of *ref* is to inline the SD being referenced, as illustrated in Figure 3.8.

3.2.6 Message complementation

The message complement operator has been adopted from [6]. If Σ is the set of messages exchanged in an SD, and $m \in \Sigma$, then $\neg m$ is $\Sigma \setminus \{m\}$. For a set $\{m, n\}$ of messages, $\neg\{m, n\} = \Sigma \setminus \{m, n\}$. For example, let $\Sigma = \{\mathtt{p}, \mathtt{q}, \mathtt{s}, \mathtt{t}\}$. Then, $\neg\mathtt{p} = \{\mathtt{q}, \mathtt{s}, \mathtt{t}\}$ and $\neg\{\mathtt{p}, \mathtt{q}\} = \{\mathtt{s}, \mathtt{t}\}$.

This operator, although not being part of UML 2.0, can be expressed in terms of UML operators as follows: Let $S \subseteq \Sigma$ be a set of messages. We replace $\neg S$ by an SD fragment in which the operator *alt* is applied to individual messages in $\Sigma \setminus S$. For example, consider the SD in Figure 3.9a with a message $\neg\{\mathtt{p}, \mathtt{q}\}$, and let $\Sigma = \{\mathtt{s}, \mathtt{t}, \mathtt{p}, \mathtt{q}\}$. This SD is equivalent to the one in Figure 3.9b where $\neg\{\mathtt{p}, \mathtt{q}\}$ is replaced by an *alt* fragment in which `s` and `t` are two alternative messages. The NFA for the sequence diagram without message complement operators can be generated in a straightforward way following the translation for the *alt* operator (see Figure 3.9c).

3.2.7 Generating Monitors from NFA

To be able to use an automaton A_S obtained from an SD S for runtime monitoring, we need to extend the language of A_S to handle system behaviours over alphabets larger than S. We do so by adding stuttering self-loops to the automaton's states. Semantically, this means that A_S does not change its state when the input symbol is outside the alphabet of S.

Definition 3.1 (Stuttering). *Let Σ_{sys} be the set of system events, and let A = (Σ, Q, δ, Q_0, F) be an NFA s.t. $\Sigma \subseteq \Sigma_{sys}$. The automaton $A' = (\Sigma_{sys}, Q, \delta', Q_0, F)$ is* the stutter-closed form *of A w.r.t. Σ_{sys} if $\delta' = \delta \cup \{(q, \Sigma_{sys} \backslash \Sigma, q) \mid \forall\ q \in Q\}$.*

The transformation of Definition 3.1 is language-preserving:

Theorem 3.2. *Let A = (Σ, Q, δ, Q_0, F) be an NFA, and let Σ_{sys} s.t. $\Sigma \subseteq \Sigma_{sys}$ be given. Let A' be the stutter-closed form of A w.r.t. Σ_{sys} (see Definition 3.1). Then for every trace $\sigma \in \Sigma_{sys}$, $\sigma \in L(A')$ iff $\sigma \downarrow_{\Sigma} \in L(A)$ (see Definition 2.3).*

Proof. *The proof follows from the fact that the construction of Definition 3.1 does not change the state-space of A:*

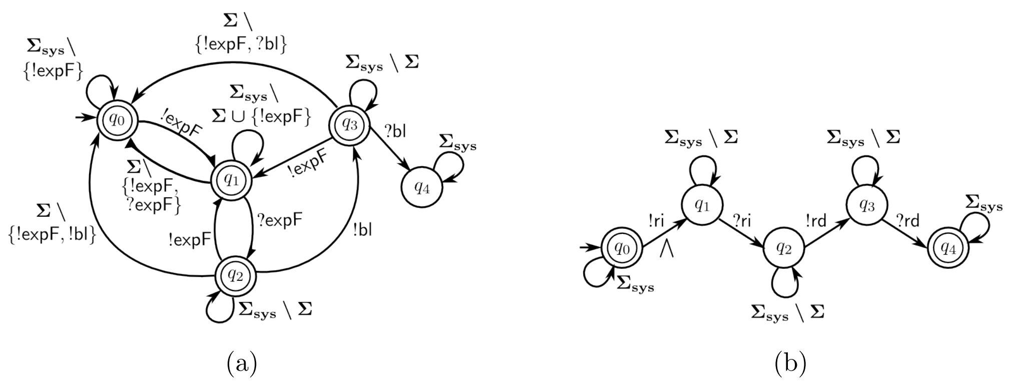

$\Sigma_{sys} \setminus \{!expF\}$
$\Sigma \setminus \{!expF, ?bl\}$
$\Sigma_{sys} \setminus \Sigma$
$\Sigma_{sys} \setminus \Sigma \cup \{!expF\}$
q_0
!expF
q_3
?bl
Σ_{sys}
$\Sigma \setminus \{!expF, ?expF\}$
q_1
!expF
q_4
$\Sigma \setminus \{!expF, !bl\}$
?expF
!expF
!bl
q_2
$\Sigma_{sys} \setminus \Sigma$
(a)
$\Sigma_{sys} \setminus \Sigma$
$\Sigma_{sys} \setminus \Sigma$
q_1
q_3
Σ_{sys}
!ri
?ri
!rd
?rd
q_0
$\land$
q_2
q_4
Σ_{sys}
$\Sigma_{sys} \setminus \Sigma$
(b)

Figure 3.10: Monitors corresponding to: (a) the simplified version of P_2 shown in Figure 3.6c; and (b) P_1.

$$
\begin{aligned}
& \sigma \in L(A') \\
\Leftrightarrow\ & \textit{(By definition of language acceptance in } A'\textit{)} \\
& \exists q_0, \ldots, q_{n+1} \in Q \cdot q_0 \in Q_0 \land q_{n+1} \in F \land \forall \sigma_i \in \sigma \cdot \delta'(q_i, \sigma_i, q_{i+1}) \\
\Leftrightarrow\ & \textit{(By definition of } \delta'\textit{)} \\
& \exists q_0, , q_{n+1} \in Q \cdot q_0 \in Q_0 \land q_{n+1} \in F \land \forall \sigma_i \in \sigma \downarrow_\Sigma \cdot \delta(q_i, \sigma_i, q_{i+1}) \\
\Leftrightarrow\ & \textit{(By definition of language acceptance in } A\textit{)} \\
& \sigma \downarrow_\Sigma \in L(A)
\end{aligned}
$$

□

For example, the monitor corresponding to the W-SD in Figure 3.6c is shown in Figure 3.10a. The language accepted by this monitor is

$$
\begin{aligned}
\Sigma^*_{sys} \quad \setminus \quad & \big(\Sigma^*_{sys} \cdot \mathsf{!expF} \cdot (\Sigma^*_{sys} \setminus \Sigma)^* \cdot \mathsf{?expF} \cdot \\
& (\Sigma^*_{sys} \setminus \Sigma)^* \cdot \mathsf{!bl} \cdot (\Sigma^*_{sys} \setminus \Sigma)^* \cdot \mathsf{?bl} \cdot \Sigma^*_{sys}\big)
\end{aligned}
$$

That is, this monitor rejects a trace that begins with a system notification that the current flight is expensive (expF) (perhaps with some events not in the vocabulary of this W-SD before or after this event), followed by a limousine booking (bl), finally followed by arbitrary events in the system. Thus, the behaviours during which an expensive flight and a limousine are booked (in that order) are rejected; these correspond to violations of the simplified version of property P_2.

The monitor for the W-SD in Figure 3.7a is shown in Figure 3.10b. Its language is

$$\big((\Sigma_{sys} \backslash !\mathsf{ri})^* \quad \cdot(!\mathsf{ri} \cdot (\Sigma_{sys} \setminus \Sigma)^* \cdot ?\mathsf{ri} \cdot (\Sigma_{sys} \setminus \Sigma)^* \\ \cdot !\mathsf{rd} \cdot (\Sigma_{sys} \setminus \Sigma)^* \cdot ?\mathsf{rd})^*\big)^*$$

This monitor accepts traces that either do not exhibit !ri at all, or, if !ri has been seen, exhibit the entire sequence ?ri·!rd·?rd. Traces not accepted by this monitor violate property P_1 of the `TAS` system.

Note that we do not add stuttering self-loops to the *critical* regions because behaviour specified in *critical* regions cannot be interleaved by other messages.

3.2.8 Complexity of the translation

Currently, our framework permits the definition of properties that depend only on the order and occurrence of system events. The size of an automaton A_S corresponding to a basic SD S, i.e., the number of states in A_S, is $O(n^k)$, where n is the number of events and k is the number of objects [4]. Applying the W-SD operators does not cause a significant increase in the size of the resulting automata except for the cases where we need to to determinize these automata which can exponentially increase their state-spaces. However, in our experience, the generated automata have been very small (see Section 3.3.3). Obviously, it remains to be seen whether the approach scales to larger web service systems and more complex properties.

By monitoring the actual *data* exchanged by conversation participants, we could check richer properties that depend on such data. We cannot use the existing automata translations for data-exchange properties directly, because the resulting automata would be too large to be useful for monitoring. Instead, we are currently investigating the use of Parameterized NFA [7] (PNFA) to create more succinct monitors, as single PNFA transitions represent sets of NFA transitions.

3.3 Sequence Diagram Templates for Temporal Logic Property Patterns

In this section, we study the expressive power of the W-SD language by using it to express temporal logic property patterns [26]. Property patterns (see Chapter 2, Section 2.2.2.2) have been shown to capture a wide variety of commonly used properties, and being able to express property patterns is a good indication of the expressive power of a new language, e.g., [6, 105]. In Sections 3.3.1 and 3.3.2, we introduce several W-SD templates and show how they can encode the SPS property patterns. We end this section by showing how to use the W-SD templates to specify properties of a new case study, the Loan Application System, in Section 3.3.3.

3.3.1 Mapping Property Patterns

In this section, we provide the W-SD templates for the SPS patterns (see Figure 3.11), and show how these templates are used to express patterns in the SPS hierarchy. Note that the actual direction of the arrows is determined when a template is instantiated.

- **Absence:** message p cannot occur in a given scope. This can be expressed as shown in Figure 3.11a.
- **Existence:** a message p must occur in a given scope. This can be expressed as shown in Figure 3.11b.
- $k-$**Bounded Existence:** message p can occur at most k times in a given scope. We can check the existence of at most k messages using the *loop* operator. After the loop, we need to check that p does not occur, which corresponds to the **Absence** pattern (see Figure 3.11c).
- **Universality:** only a sequence p^* of messages can occur in a given scope. This is equivalent to checking for the absence of complement messages (see Figure 3.11d).

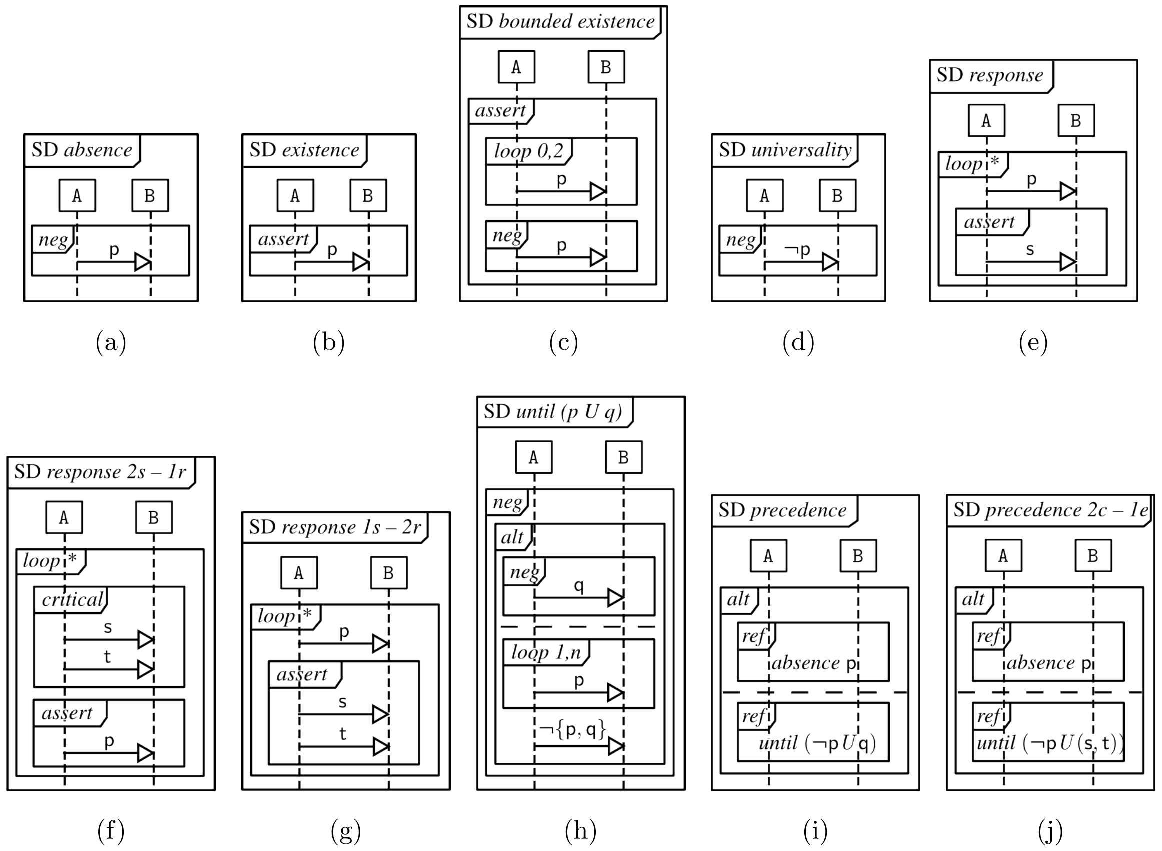


Figure 3.11: Property pattern mappings for SDs: (a) **Absence**; (b) **Existence**; (c) **Bounded Existence**; (d) **Universality**; (e) Response; (f) **Response Chain** (2 stimulus – 1 response); (g) **Response Chain** (1 stimulus – 2 response); (h) **Until**; and (j) **Precedence Chain** (2 cause – 1 effect). The directions of the arrows are determined when a template is instantiated, and $(\mathtt{s}, \mathtt{t})$ means message `s` followed by message `t`.

- **Response:** message `p` (stimulus) must be followed by message `s` (response), in a given scope. A response can occur without stimuli, so the stimulus is represented using a regular message, whereas the response is mandatory. The existence of stimulus/response pairs are checked in an infinite *loop*, as there can be many stimulus/response pairs in one execution trace (see Figure 3.11e).

- **Response Chain:** a sequence $\mathsf{p}_1, \ldots, \mathsf{p}_n$ of messages must be followed by the sequence $\mathsf{q}_1, \ldots, \mathsf{q}_m$ of messages, in a given scope. We show two examples of this pattern: p responds to s, t (see Figure 3.11f), and s, t responds to p (see Figure 3.11g). This pattern has the same basic form as **Response**.

- p responds to s, t: 2 stimulus – 1 response. The *critical* operator is used to enclose the message sequence s, t, to ensure atomicity of this sequence. An *assert* cannot be used since the stimulus sequence is optional.
- s, t responds to p: 1 stimulus – 2 response. The message sequence now occurs within the *assert* operator, so an additional *critical* operator would be superfluous.

- **Until:** This pattern is not part of the SPS; however, it is used to specify the **Precedence** patterns. A sequence p^* of messages occurs until the first occurrence of message q, in a given scope (see Figure 3.11h). This pattern, formalized using a single "until" temporal operator [20], can be refuted in one of two ways: either q never occurs, or after seeing a finite number of p messages (expressed using *loop 1, n*), neither a p nor a q message occurs (expressed as $\neg\{\mathsf{p}, \mathsf{q}\}$).

- **Precedence:** a message s (cause) precedes a message p (effect), as shown in Figure 3.11i. This pattern allows the cause part to occur without the effect. We describe this pattern in W-SD by expressing the two possible cases that this pattern specifies: a) p never occurs, or b) p never occurs before s. The first case corresponds to checking *absence* of p; the second – to checking $\neg\mathsf{p}\ U\ \mathsf{s}$ (the "until" template), since we want to be sure that *no* p messages are sent before the first s message.

- **Precedence Chain:** a sequence $\mathsf{p}_1, \ldots, \mathsf{p}_n$ of messages must precede the sequence $\mathsf{q}_1, \ldots, \mathsf{q}_m$ of messages, in a given scope. We show an example of this pattern, 2 cause – 1 effect, p is preceded by s, t (see Figure 3.11j). This pattern is implemented using the **Absence** and **Until** patterns, just like in the **Precedence** pattern. The

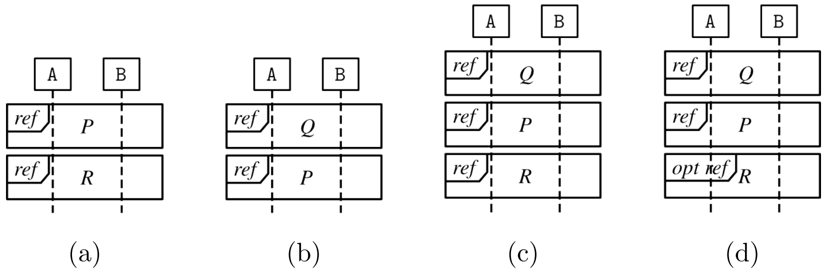


Figure 3.12: Scope mapping for sequence diagrams, where P represents the property being scoped: (a) **Before** R; (b) **After** Q; (c) **Between** Q **and** R; and (d) **After** Q **until** R.

implicit *negate* operators in the **Absence** and **Until** patterns handle the message sequences, so there is no need to add *critical* operators.

In the SDs in Figure 3.11, symbols `p`, `q`, `s`, and `t` can denote complex SDs rather than just the individual messages. In this case, we treat these symbols as placeholders and use a *ref* operator for the SDs that should be inserted in their place, and replace message complementation by negation.

3.3.2 Mapping Property Scopes

We now show how to express property patterns involving scopes which are used to define the traces over which a property will be monitored. Scopes can be simple messages or more complex scenarios in our specification language. The *ref* operator is used to introduce scope delimiters in the corresponding locations. For example, to apply the **Before** R scope to property P, the scope delimiter R is inserted after the property we wish to verify (see Figure 3.12a). In the case of the **After** Q scope, the delimiter is inserted before the property (see Figure 3.12b). Finally, both the **Between** (see Figure 3.12c) and the **After-until** (see Figure 3.12d) scopes add before/after delimiters. In the **After-until** scope, the property is valid even if the “until” part does not occur. Therefore, the second delimiter in this scope is optional. Thus, there is an implicit *opt* operator in each scope delimiter.

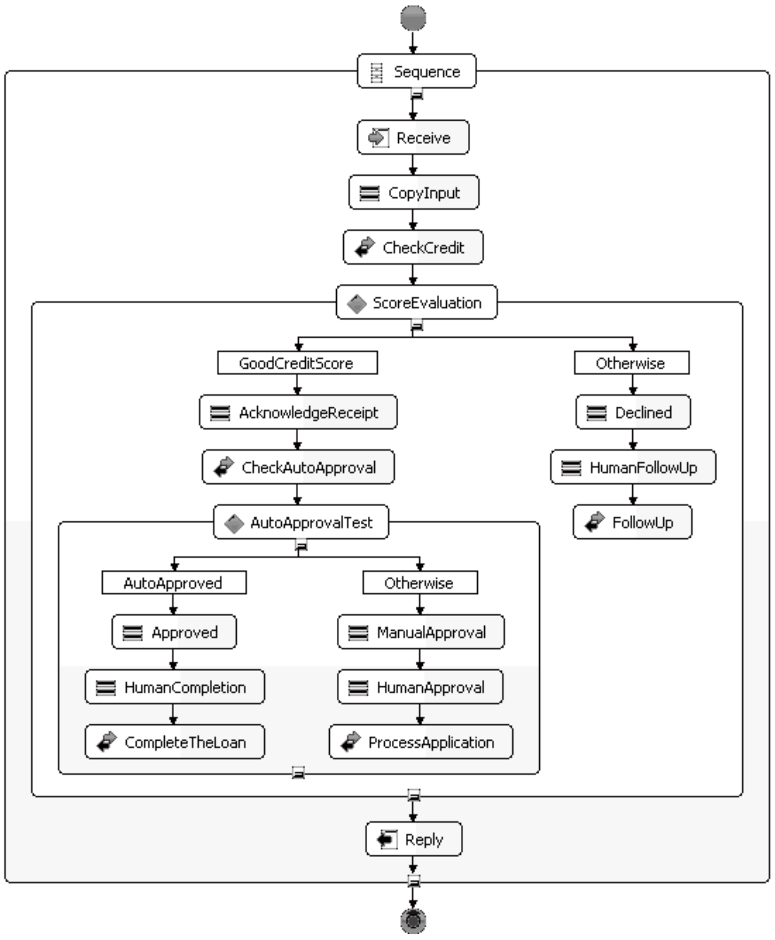


Figure 3.13: Workflow describing the high-level steps of the `LAS` system.

3.3.3 Specifying Properties of the Loan Application System

In this section, we present a complete example of how to specify properties.

The Loan Application System (`LAS`) is distributed as a sample application with the IBM® WebSphere® Integration Developer v6.0.2. Users enter loan application information (name, taxpayer id, loan amount) through a web page, and are eventually informed of the status of their applications. The `LAS` workflow first checks if the user's credit score is valid, and will decline their loan request if the user has a bad credit score, i.e., less than 750. A credit score is considered valid if it is between 300 and 850. If the credit score is good, the workflow then checks the loan amount: loans for $50,000 or less are automatically approved; loans for larger amounts are earmarked for manual approval.

$P_1^{\texttt{LAS}}$	The credit score should always be valid, i.e., between 300 and 850.
$P_2^{\texttt{LAS}}$	The credit score should eventually be checked if the loan amount is greater than zero.
$P_3^{\texttt{LAS}}$	A loan cannot be granted if the loan amount is less than or equal to zero.
$P_4^{\texttt{LAS}}$	After checking that the applicant has a good credit score, a loan cannot be granted if the loan amount is less than or equal to zero.
$P_5^{\texttt{LAS}}$	No-one can get a loan without first going through a credit check.

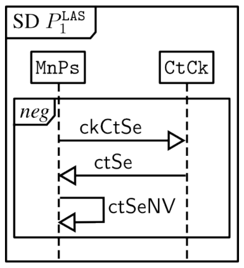
Table 3.1: Several properties of the LAS system.

3.3.3.1 BPEL Model

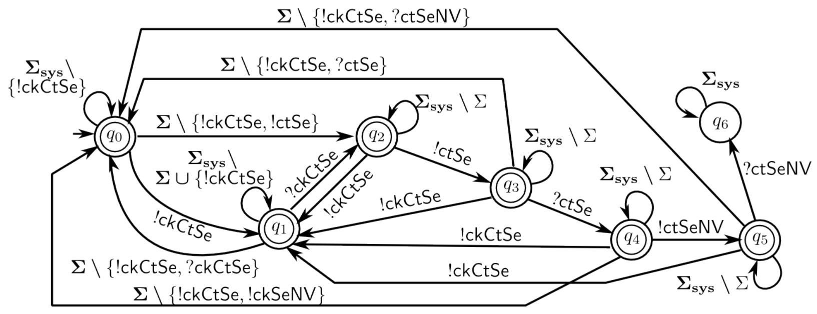
Figure 3.13 shows a BPEL diagram that implements the previously described LAS workflow. The LAS system interacts with four partners: the web service CreditCheck, implemented in Java, and three human tasks (FollowUpDeclinedApp, CompleteTheLoan and ProcessTheApplication). Specifically, the CheckCredit activity in Figure 3.13 invokes the CreditCheck partner, which uses the taxpayer id to retrieve the corresponding credit score. The human tasks CompleteTheLoan, ProcessApplication and FollowUp follow the application results Approved, ManualApproval and Declined, respectively. The two conditional activities ScoreEvaluation and AutoApprovalTest are assigned values by a local rule group that checks the credit score and the loan amount.

3.3.3.2 Properties

Since the LAS system is a composition of several distributed business processes, its correctness depends on the correctness of its partners and their interactions. For example, the system should guarantee that every request is eventually acknowledged and none are lost or blocked indefinitely, or that loans are only given to customers with a good credit score. However, in the provided LAS application, the CreditCheck module assigns a



(a)


$\Sigma \setminus \{$!ckCtSe, ?ctSeNV$\}$
$\Sigma \setminus \{$!ckCtSe, ?ctSe$\}$
$\Sigma_{sys} \setminus \{$!ckCtSe$\}$
$\Sigma \setminus \{$!ckCtSe, !ctSe$\}$
$\Sigma_{sys} \setminus \Sigma$
$\Sigma_{sys} \setminus \Sigma \cup \{$!ckCtSe$\}$
?ckCtSe
!ckCtSe
!ctSe
!ckCtSe
?ctSe
!ckCtSe
!ctSeNV
!ckCtSe
?ctSeNV
Σ_{sys}
$\Sigma \setminus \{$!ckCtSe, ?ckCtSe$\}$
$\Sigma \setminus \{$!ckCtSe, !ckSeNV$\}$
q_0
q_1
q_2
q_3
q_4
q_5
q_6

(b)

Figure 3.14: $P_1^{\mathtt{LAS}}$: **Absence** pattern. (a) A W-SD describing the property and (b) the resulting monitor.

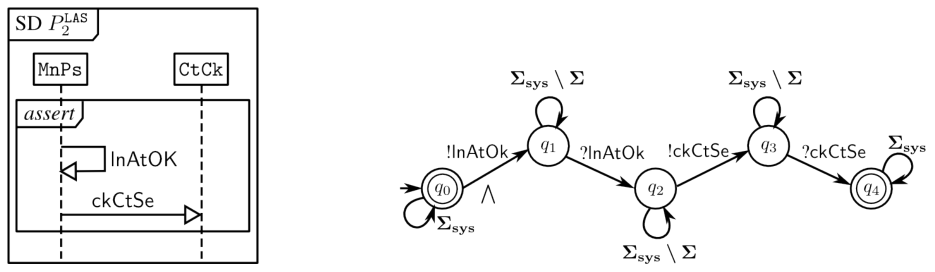
credit score at random, without using the customer identifier, thus preventing the overall system from satisfying this property. Table 3.1 shows some properties of the LAS system. We now show how to express these properties using W-SD:

Property $P_1^{\mathtt{LAS}}$**:** "The credit score should always be valid, i.e., between 300 and 850."

We express this property using the W-SD **Absence** pattern (see Fig. 3.11(a)): our property holds if there are no scenarios where an invalid credit score is returned in response to a check credit score request. In the LAS system, MnPs sends a check credit score request (ckCtSe), to CtCk. In response, CtCk sends the customer's credit score (ctSe) to MnPs. A creditScoreNotValid (ctSeNV) message is sent if the value is not in the correct range (300–850). This property is expressed in W-SD by applying a *negate* operator to the sequence !ckCtSe.?ckCtSe.!ctSe.?ctSe.!ctSeNV.?ctSeNV. The resulting W-SD is shown in Figure 3.14a, and the resulting monitor is shown in Figure 3.14b.

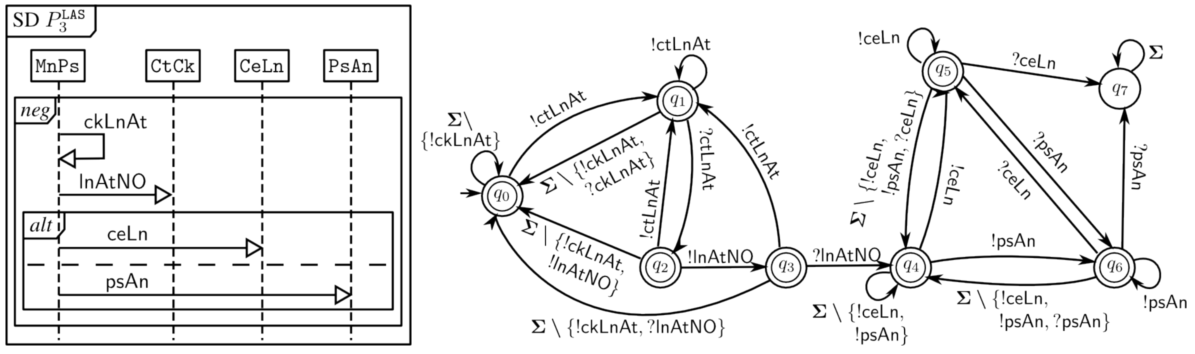
Property $P_2^{\mathtt{LAS}}$**:** "The credit score should eventually be checked if the loan amount is greater than zero."

We express this property using the W-SD **Existence** pattern (see Fig. 3.11(b)): the property holds if the credit score is checked after the system is notified that the loan



(a) (b)

Figure 3.15: $P_2^{\texttt{LAS}}$: **Existence** pattern. (a) A W-SD describing the property and (b) the resulting monitor.


SD $P_3^{\texttt{LAS}}$
MnPs
CtCk
CeLn
PsAn
neg
ckLnAt
lnAtNO
alt
ceLn
psAn
$\Sigma \setminus$ {!ckLnAt}
!ctLnAt
$\Sigma \setminus$ {!ckLnAt, ?ckLnAt}
?ctLnAt
!ctLnAt
$\Sigma \setminus$ {!ckLnAt, !lnAtNO}
!lnAtNO
$\Sigma \setminus$ {!ckLnAt, ?lnAtNO}
?lnAtNO
$\Sigma \setminus$ {!ceLn, !psAn}
!ceLn
$\Sigma \setminus$ {!ceLn, !psAn, ?ceLn}
!ceLn
?ceLn
?psAn
?ceLn
Σ
?psAn
!psAn
$\Sigma \setminus$ {!ceLn, !psAn, ?psAn}
!psAn
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7

(a) (b)

Figure 3.16: $P_3^{\texttt{LAS}}$: **Absence** pattern. (a) A W-SD describing the property and (b) the resulting monitor.

amount is greater than zero. In the `LAS` system, the main process `MnPs` first checks the predicate "loan amount is > 0", sending a loanAmountOkay (lnAtOK) message if the condition holds, and a loanAmountNotOkay (lnAtNO) message otherwise. In order to check the customer's credit, `MnPs` must then send a checkCreditScore (ckCtSe) message to the `CtCk` component. Thus, the desired scenario is !lnAtOK.?lnAtOK. !ckCtSe.?ckCtSe, which is enclosed in an *assert* operator in Figure 3.15a. Figure 3.15b shows the monitor corresponding to this W-SD.

Property $P_3^{\texttt{LAS}}$: "A loan cannot be granted if the loan amount is less than or equal to zero."

Like $P_1^{\texttt{LAS}}$, we express this property using the **Absence** pattern: $P_3^{\texttt{LAS}}$ holds if there

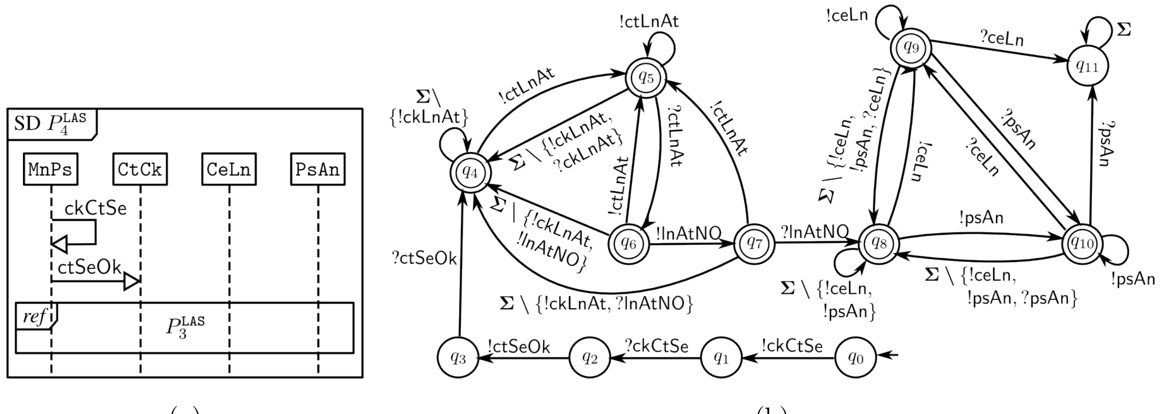
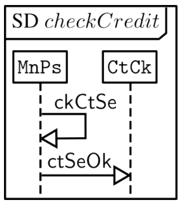


Figure 3.17: $P_4^{\texttt{LAS}}$: **Absence** pattern, Scope **After**. (a) A W-SD describing the property and (b) the resulting monitor, obtained by concatenating the NFAs for the scope and $P_3^{\texttt{LAS}}$.

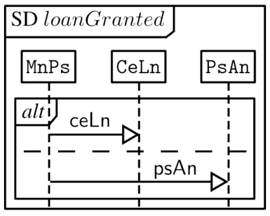
are no scenarios where a loan is granted after the system has been warned that the loan amount is less than or equal to zero. As mentioned before, the LAS system sends a loanAmountNotOkay (lnAtNO) if the predicate "loan amount is > 0" is false. A loan is considered granted if it is manually or automatically approved, which can be monitored by checking if the main process MnPs sends a completeTheLoan (ceLn) or processTheApplication (psAn) message. See Figure 3.16a for the corresponding W-SD; the resulting monitor is shown in Figure 3.16b.

Property $P_4^{\texttt{LAS}}$: "After checking that the applicant has a good credit score, a loan cannot be granted if the loan amount is less than or equal to zero."

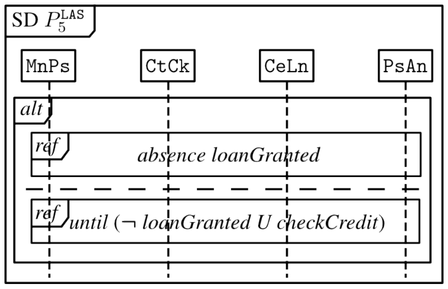
This property is equivalent to the property P_3 with the **After** Q scope, where Q is "checking for a good credit score". To express it, we introduce the scope delimiter Q before the property $P_3^{\texttt{LAS}}$, as seen in Fig. 3.12b. The W-SD corresponding to $P_4^{\texttt{LAS}}$ is shown in Figure 3.17a and consists of two parts: (1) scope Q and (2) property $P_3^{\texttt{LAS}}$, i.e., the fragment specified by a *ref* operator which should be replaced by the SD for $P_3^{\texttt{LAS}}$. The resulting monitor is shown in Figure 3.17b.



(a)



(b)



(c)

Figure 3.18: $P_5^{\texttt{LAS}}$: The **Precedence** pattern. (a) A W-SD for *checkCredit*; (b) A W-SD for *loanGranted*; (c) A W-SD showing application of the **Precedence** pattern.

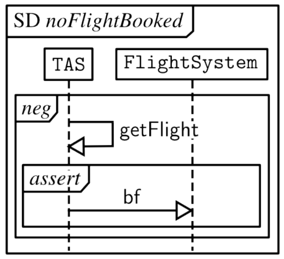
Id	Pattern/Scope	# Partners	# Events	# States	# Transitions
$P_1^{\texttt{LAS}}$	**Absence**	2	6	7	18
$P_2^{\texttt{LAS}}$	**Existence**	2	4	5	9
$P_3^{\texttt{LAS}}$	**Absence**	4	8	8	23
$P_4^{\texttt{LAS}}$	**Absence**; Scope **After**	4	10	12	27
$P_5^{\texttt{LAS}}$	**Precedence**	4	8	28	95

Table 3.2: Sizes of W-SD `LAS` monitors.

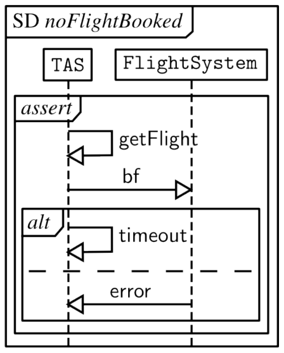
Property $P_5^{\texttt{LAS}}$: "No-one can get a loan without first going through a credit check."

> At this point, we have identified common scenarios that occur in the `LAS` system: SDs *creditCheck* (see Figure 3.18a) and *loanGranted* (see Figure 3.18b). We can now express property $P_5^{\texttt{LAS}}$ using the **Precedence** pattern: SD *creditCheck* must precede SD *loanGranted.* Note that the SD *creditCheck* is not optional and must occur for the property to hold. The SD for $P_5^{\texttt{LAS}}$ is shown in Figure 3.18c.

Table 3.2 shows the details of the W-SD specifications of the `LAS` properties. In this table, column "Id" contains a unique identifier for each property; "Property" is the actual property to be checked; "# Partner" corresponds to the number of partners involved in the corresponding SD; "# Events" is the number of events sent between partners in the SD; "# States" corresponds to the number of states in the corresponding automaton;



(a)



(b)

Figure 3.19: Expressing property P_3: (a) using the existing alphabet of the `TAS` system; (b) with additional events.

and "# Transitions" is the number of transitions in the monitor. Note that all of the constructed automata have fever than 100 transitions, so we expect that monitoring will not produce a significant performance overhead. We describe our experience monitoring this application in Chapter 6, Section 4.2.1.

3.4 Overcoming the Assertion Restriction

In Section 3.2, we provided a transformation from our language, W-SD, to NFA, showing that it can capture safety and finitary liveness properties. Our transformation further shows that W-SD is not more expressive than regular expressions, i.e., the language recognized by NFA.

The main restriction in W-SD is that we do not allow the nesting of *assert*s within the scope of *negate*s. For example, let us consider the following property (P_3) of the `TAS` system: "If the customer requests a flight, but the system cannot book one, inform the customer". This property can be expressed as a UML 2.0 Sequence Diagram but not in W-SD. The reason for this limitation is the chosen set of events of the `TAS` system: flight bookings are handled using only one event, `bf`. Thus, failure to book a flight means that we did not get a confirmation number from the `FlightSystem`. Since it is not clear how long TAS should wait before declaring a failure, we have to express the property using an *assert* inside a *negate*, as shown in Figure 3.19a, which is not allowed in our language.

However, this restriction is mainly syntactic, because we can always push the *negate* operator down to the atomic level, and reformulate the sequence diagram into a semantically equivalent one in which *negate* is not applied within the scope of *assert*. In our example, the problem can be fixed by adding two additional events to the TAS system that give a reason why the flight booking failed: timeout (produced if a confirmation number is not received by a certain time) and error (produced if the booking could not be made). With these events, property P_3 can be expressed as shown in Figure 3.19b, which is within the SD language.

Note that after removing the *negate* operator, the resulting sequence diagram may have brand new scenarios: to do the removal, we need to elicit the set of all possible scenarios complementary to the scenario enclosed by the *negate*. The process of enumeration and analysis of all possible alternative scenarios obviously requires domain knowledge and thus cannot be automated in general. However, the online nature of our monitoring framework allows us to register for and collect the alternative scenarios with ease.

3.5 Related Work

Like other partial-order scenario-based formalisms such as MSCs [53] and LSCs [23], UML Sequence Diagrams are enjoying an increasing usage as a specification language. In this section, we summarize some work studying UML 2.0 Sequence Diagrams as a specification language.

Lettrari and Klose [60] show how UML 1.3 Sequence Diagrams can be used to check properties of UML models. UML 1.3 SDs allow only simple event sequences, so the language formalized in [60] is a small subset of our specification language.

Ameedeen and Bordbar [5] show how a subset of UML 2.0 SDs can be transformed into Free Choice Petri nets, enabling the use of the corresponding analysis techniques. This SD subset is only used to specify possible system behaviours, and thus does not include the *negate* and *assert* operators. This work also assumes that sending and receiving an event happen simultaneously. While this assumption works well for synchronous systems, it does not hold for most web applications which rely on message queues for communication.

Autili et al. [6] propose a Property Sequence Chart (PSC) language, which is an extended notation of a subset of UML 2.0 SDs. PSC enables expressing safety and liveness properties by assigning attributes *fail* and *required* to messages. This is equivalent to applying operators *negate* and *assert* to individual SD message, respectively. The semantics of PSC is given using Büchi automata, designed to operate on infinite execution traces. Since we consider only finite executions of web services, automata over finite words are sufficient and significantly easier to implement.

STAIRS [46] is a trace-based requirement specification methodology that also uses extended UML 2.0 SDs. Trace scenarios are classified into positive (mandatory and potential), negative, and inconclusive. Negative traces are captured using the *negate* operator. STAIRS does not use *assert* and instead defines a new mandatory choice operator, *xalt*, to express the requirement that both alternatives be present in a choice. In our work, we enable expression of mandatory and forbidden behaviours without extending the language.

Grosu and Smolka [41] interpret positive and negative UML 2.0 Sequence Diagrams as safety and liveness properties and give formal semantics for such diagrams using safety and liveness automata, respectively. Their approach does not use the *assert* operator and defines automata over infinite traces.

Harel and Maoz [44] define Modal Sequence Diagrams (MSD), an extension of UML 2.0 Sequence Diagrams. The semantics of *negate* and *assert* operators in MSD is given via the universal/existential distinction made by the Live Sequence Charts (LSCs) [23]. In this formalism, diagrams, messages and constraints can be defined as either *hot*(universal) or *cold* (existential), and the semantics of MSDs is given via alternating weak word automata (AFA). This formalism includes not only non-deterministic choices of NFA (the language into which we translated SDs) but universal choices as well [44]. Given that any AFA can be translated to an (exponentially larger) NFA [100], we believe that SDs and LSCs have the same expressive power. These languages, however, differ in their syntactic and usability properties. Specifically, LSCs are more succinct because they can freely combine non-deterministic and universal choices. However, SDs are easier to implement and use in a monitoring framework because of the existence of several efficient packages for manipulating NFAs. Moreover, unlike LSCs, the syntax of SDs conforms to UML 2.0 and hence many existing UML tools can be used to capture and display these diagrams. Finally, an existential, *constant*, subset of LSCs has been expressed in terms of NFA [45]. It is a strict subset of SDs, not allowing universal traces.

While we concentrated on specifying behavioural properties of interactions between partners, Bultan [15] identified Collaboration Diagrams (CSs) and Conversation Protocols (CPs) as more appropriate formalisms for specifying such properties as realizability and synchronizability, which he then checks using model-checking. These formalisms are simpler than UML 2.0 Sequence Diagrams and are appropriate for expressing such special-purpose properties.

3.6 Summary

In this chapter, we presented W-SD, a subset of UML 2.0 SDs that can be used as a language for specifying properties of BPEL applications. Specifications expressed in W-SD permit the analysis of orchestrations involving multiple partners, from the point of view of the orchestrating service. We demonstrated the expressiveness of this subset by successfully mapping all the Specification Property System patterns into our SD subset.

In the rest of this thesis, we continue to use property patterns, but work with simpler safety and liveness properties, presented in Section 4.1.2.

Chapter 4

Monitoring and Recovery

In Chapter 1, we identified three major challenges facing the development of an online monitoring and error recovery framework for web service applications: Specification, Runtime Monitoring and Error Recovery. In the previous chapter, we focused on the Specification challenge, by proposing a Sequence Diagram-based language for specifying properties. In this chapter, we jointly address the monitoring and recovery challenges – given an application path which led to a failure and a monitor that detected it, our goal is to compute a set of suggestions, i.e., *plans*, for recovering from these failures. To this end, we have developed various techniques for computing recovery plans from runtime errors.

In the rest of this thesis, we consider behavioural correctness properties to be scenarios that the system should or should not exhibit as simple safety and liveness properties, instead of the more complex scenarios expressed in W-SD. The decomposition into safety and liveness is essential for recovery, as different techniques are used to compute recovery plans for each type of property. We cannot algorithmically decompose arbitrary W-SD properties into safety and liveness, so this simplification allows us to cleanly present the connection between the type of property that was violated, and the type of plans that can be generated for this violation. As discussed in Chapter 1, Section 1.5, we

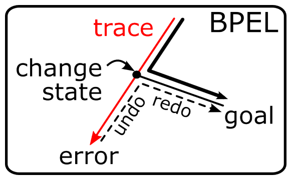


Figure 4.1: A schematic view on plan generation.

focus on two types of plans. For violations of properties capturing forbidden behaviour, a recovery plan should attempt to return the application to an earlier state, one at which an alternative path that potentially avoids the fault is available. For violations of properties capturing desired behaviours, merely going back is insufficient to ensure that the system can produce the desired behaviour. In this case, we compute plans that attempt to redirect the application towards executing new activities that may lead to the satisfaction of the property in question. Figure 4.1 (replicated from Chapter 1) schematically shows the second type of plans.

In this chapter, we express properties using the Specification Pattern System (SPS) (see Chapter 2, Section 2.2.2.2), converting the high-level patterns into Quantified Regular Expressions (QRE) (see Chapter 2, Section 2.2.1.3) 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. This monitoring part of our framework is based on the work presented by Yuan Gan in her M.Sc. thesis [37]. Gan's monitoring framework checks behavioural conformance by creating monitors and then eavesdropping on conversations between the application being monitored and the server it runs on. This approach is adequate for reporting property violations, but not for error recovery. The reason is that in some cases, the application terminates before a property violation is reported, at which point there is nothing to recover from.

Before we can formalize our strategies for monitoring, and computing recovery plans, we need to introduce various concepts, like change states and goal transitions, as well as various preprocessing steps, like creating a model of the application with compensation.

We do this in Section 4.1. In Section 4.2, we give an overview of Gan's framework, and explain how it must be adapted in order to permit error recovery. We then present our techniques for generating recovery plans for forbidden and desired behaviours in Sections 4.3 and 4.4, respectively. Finally, we present related work on runtime monitoring and error recovery in Section 4.6.

4.1 Preprocessing

The inputs to the Preprocessing stage of our framework are the BPEL program B and the set of properties expressed using SPS patterns. We begin with converting B into a formal representation, $\mathsf{L}(\mathsf{B})$, which is a Labelled Transition System (LTS) (see Section 2.3). We then enrich it with transitions on compensation actions to get $\mathsf{L}_\mathsf{C}(\mathsf{B})$ (see Section 4.1.1). In Section 4.1.2, we discuss the translation of user-specified properties into monitors. Finally, in Section 4.1.3 we formalize change states and potential goal transitions and provide an algorithm for computing these statically on $\mathsf{L}(\mathsf{B})$.

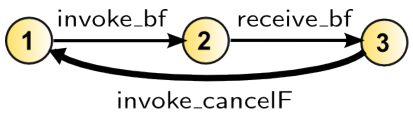
4.1.1 Formalizing BPEL Compensation

As described in Section 2.3, we have adopted Foster's [30] formalization of BPEL using Labelled Transition Systems. Thus, formally, a BPEL program B is represented by its LTS translation $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I)$, where the set of labels Σ is derived from the possible events in B: service invocations and returns, <onMessage> events, <scope> entries, and condition valuations. The set of states S and the transition relation δ are those produced by the translation.

In order to reason about termination, we have added a new state t to S, and a new system event TER to Σ. This state is reached from any state of S via a TER event: $\forall s \in S \setminus \{\mathsf{t}\}, (s, \mathsf{TER}, \mathsf{t}) \in \delta$. This TER event currently represents all possible termination causes. Since the BPEL standard treats service timeouts differently, we can, in principle,

```
<scope name="bf">
   <invoke ... operation = "bf" ... outputVariable = "flightConf" />
   <compensationHandler cost = "9">
       <invoke ... operation = "cancelF" inputVariable = "flightConf"/>
   </compensationHandler>
</scope>
```

(a)



(b)

Figure 4.2: (a) BPEL definition of flight booking service invocation (bf), including its compensation; and (b) LTS translation of the bf activity and its compensation (bold).)

distinguish between the termination of the orchestrating service (regular or due to an error) and partner service timeouts, and thus, can reason about service timeouts by introducing another special event, TIMEOUT. However, we cannot determine whether a service that timed out will eventually respond, or if it definitely cannot respond.

In order to capture BPEL's compensation mechanism, we introduce additional, backwards transitions. For example, the compensation for bf, specified in Figure 4.2a, is captured by adding the transition $3 \xrightarrow{\mathsf{invoke\_cancelF}} 1$ as shown in Figure 4.2b. Taking this transition effectively leaves the application in a state where bf has not been executed. However, the program state may or may not revert to its original state after the compensatory action completes its execution, as this depends on the definition the compensatory action. For example, cancelF cancels the booked flight, but does not reset the reservation confirmation number, which is increased by one each time a booking is made. We denote by τ an "empty" action, allowing undoing of an action without requiring an explicit compensation action. For example, transitions labelled with condition valuations are compensated by τ.

Note that we have made a major assumption that compensation returns the application to one of the states that has been previously seen. Thus, given a BPEL program B and its translation to LTS $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I)$, we translate B with compensation into an

LTS $\mathsf{L_C}(\mathsf{B}) = (S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I)$, where Σ_c is the set of compensation actions (including τ) and δ_c is the set of compensation transitions. Although service providers define compensation per service, compensation is defined per action in a BPEL specification. In other words, two different invocations of the same service may be compensated in different ways. For this reason, for each compensatable transition $(s, a, s') \in \delta$, there is an inverse transition $(s', a_c, s) \in \delta_c$, where a_c is the compensation action associated to the action a in the BPEL specification.

Figure 4.3 shows $\mathsf{L_C}(\mathtt{TAS})$. To increase legibility, we do not show the termination state t and transitions to it. Also, we only show one transition for each service invocation, abstracting the return transition and state. In this notation, the LTS in Figure 4.2b has two transitions: $1 \xrightarrow{\mathsf{bf}} 3$ and $3 \xrightarrow{\mathsf{cancelF}} 1$. This allows us to visually combine an action and its compensation into one transition, labelled in the form $a/\bar{a}$, where a is the application activity and $\bar{a}$ is its compensation. In other words, each transition $s \overset{a/\bar{a}}{\longleftrightarrow} t$ in Figure 4.3 represents two transitions: $(s, a, t) \in \delta$ and $(t, \bar{a}, s) \in \delta_c$.

For example, the <pick> activity (➡ labelled ① in Figure 2.4) corresponds to state 2 of Figure 4.3. The choice between $\mathsf{onlyCar}$ and $\mathsf{carAndFlight}$ is represented by two outgoing transitions from this state: $(2, \mathsf{onlyCar}, 3)$ and $(2, \mathsf{carAndFlight}, 6)$. Since these actions do not affect the state of the application, they are compensated by τ. The <flow> activity (scope enclosed in bold, blue lines — labelled ② in Figure 2.4) results in two branches, depending on the order in which the air and ground transportation are executed. The compensation for these events is also τ.

4.1.2 From Properties to Monitors

Apart from the system to be monitored, our framework also takes as input the 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. Our framework also includes an (optional) ranking of the properties in the order

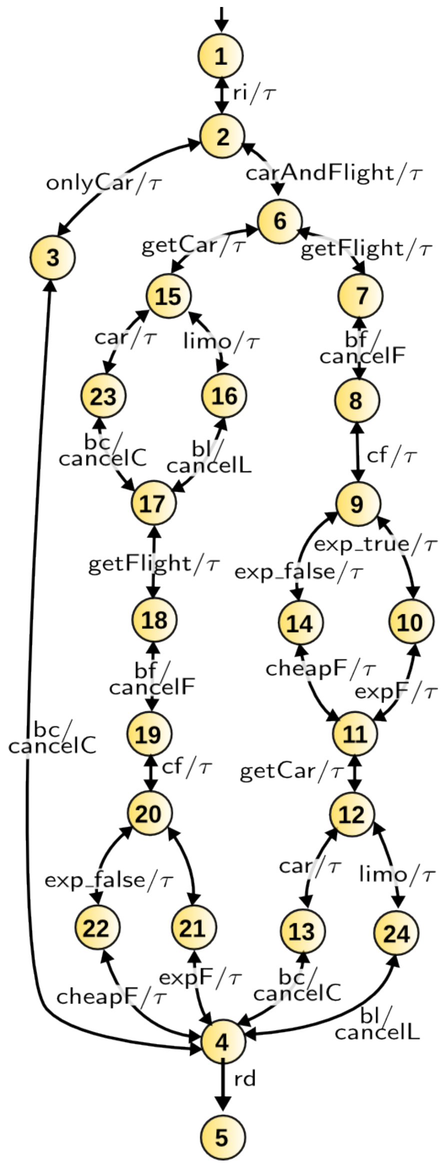


Figure 4.3: LTS $\mathsf{L_C}(\mathtt{TAS})$: downward and upward arrows show forward and compensation logic, respectively.

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).

As mentioned in Chapter 3, Section 3.6, we restrict ourselves to simpler safety and liveness properties in this chapter, instead of more complex properties specified in W-SD. These properties are expressed using the Specification Pattern System [26] (see Section 2.2.2.2 for an overview), and we use the Quantified Regular Expression (QRE) (see Section 2.2.1.3) encoding of these patterns to produce monitoring automata. For example, the properties of the `TAS` system can be formalized as follows:

Property 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" describes a positive behaviour (the destination must be reached). This property can be expressed using the **Response** pattern in a **Global** scope, and the resulting QRE property is:

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

Property P_2: "the user cannot book both a limousine (bl) and an expensive flight (expF)" describes a negative scenario that should be avoided (a limousine and an expensive flight are booked). 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:

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

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

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.

In order to be verified, properties are translated into deterministic finite automata (DFAs), which we call "monitors". Different algorithms to perform such a translation from a QRE formula exist in the literature [47]. 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 4.4a shows the monitor built for the property pattern: "s responds to p after q until r". State 4 of the monitor (coloured 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 (coloured 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 2.3.

Similar monitors are built for our example properties P_1 and P_2. Monitor $\mathtt{A}_1$ in Figure 4.4b represents P_1: if the application terminates before rd appears, the monitor moves to the (error) state 3. State 1 is a good state since the monitor enters it once the booked transportation reaches the destination (rd). Monitor $\mathtt{A}_2$ in Figure 4.4c represents both P_{2a} and P_{2b}. 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 (coloured) monitors below.

Definition 4.1 (Recovery Monitor). *A* recovery 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.*

A *accepts* a word $a_0a_1a_2...a_{n-1} \in \Sigma^*$ iff there exists an execution $s_0a_0s_1a_1s_2...a_{n-1}s_n$ of A such that $a_0 \in I$ 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 colours to states in S. Accepting states are coloured red, signalling violation of the property. State 3 of Figure 4.4b and state 4 in Figures 4.4c and 4.4a are red states (also shaded horizontally). Yellow states are those from which a red state can be reached through a single transition. Formally, for a state $s \in S$,

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

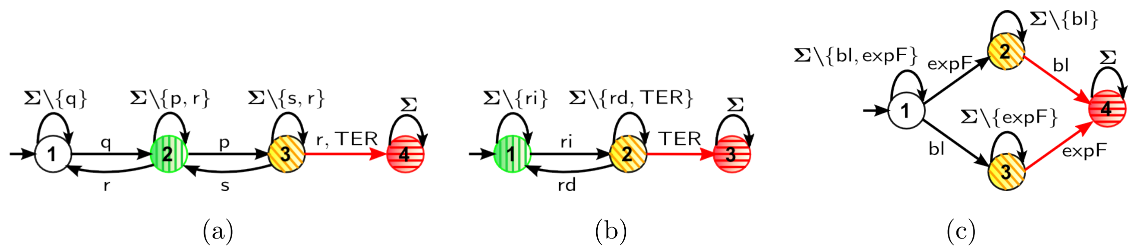


Figure 4.4: Monitors: (a) for a property pattern "`s` responds to `p` after `q` until `r`", (b) $\mathtt{A}_1$, and (c) $\mathtt{A}_2$. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

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

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

State 2 in Figure 4.4b, states 2 and 3 in Figure 4.4c and state 3 in Figure 4.4a are yellow states (also shaded diagonally).

The green colour is used for states that can serve as good places to which a recovery plan can be directed. We define green states to be those states that are not red or yellow, but that can be reached through a single transition from a yellow state. Formally,

$$\begin{aligned} colour(s) = \text{green} \quad \text{iff} \quad & (colour(s) \neq \text{red}) \ \wedge \ (colour(s) \neq \text{yellow}) \ \wedge \\ & (\exists a \in \Sigma, \ \exists s' \in S \cdot (colour(s') = \text{yellow}) \ \wedge \ ((s', a, s) \in \delta)). \end{aligned}$$

State 1 in Figure 4.4b, as well as state 2 in Figure 4.4a are coloured green (also shaded vertically). Note that not all monitors have green states. For example, in $\mathtt{A}_2$ of Figure 4.4c 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. Formally:

Definition 4.2 (Safety Monitor). *A recovery monitor* $A = (S, \Sigma, \delta, I, F)$ *is a* safety monitor *if* $\forall s \in S \setminus F, \text{colour}(s) \in \{\textit{white}, \textit{yellow}\}$ *and* $\forall s \in F, \text{colour}(s) = \textit{red}$

Definition 4.3 (Mixed Monitor). *A recovery monitor* $A = (S, \Sigma, \delta, I, F)$ *is a* mixed monitor *if* $\forall s \in S \setminus F, \text{colour}(s) \in \{\textit{white}, \textit{yellow}\}$ *and* $\forall s \in F, \text{colour}(s) \in \{\textit{red}, \textit{green}\}$

Given specification properties $\Phi_1 - \Phi_n$, we translate them to a set $\mathsf{A} = \{\mathsf{A_1}, ..., \mathsf{A_n}\}$ of recovery monitors, denoting by $\mathsf{A_S}$ the subset of A that includes all safety monitors. In the rest of this thesis, when we say "monitor", we mean "recovery monitor" (see Definition 4.1) instead of W-SD monitor (as defined in Chapter 3, Section 3.2.7).

4.1.3 Identifying Goal Transitions and Change States

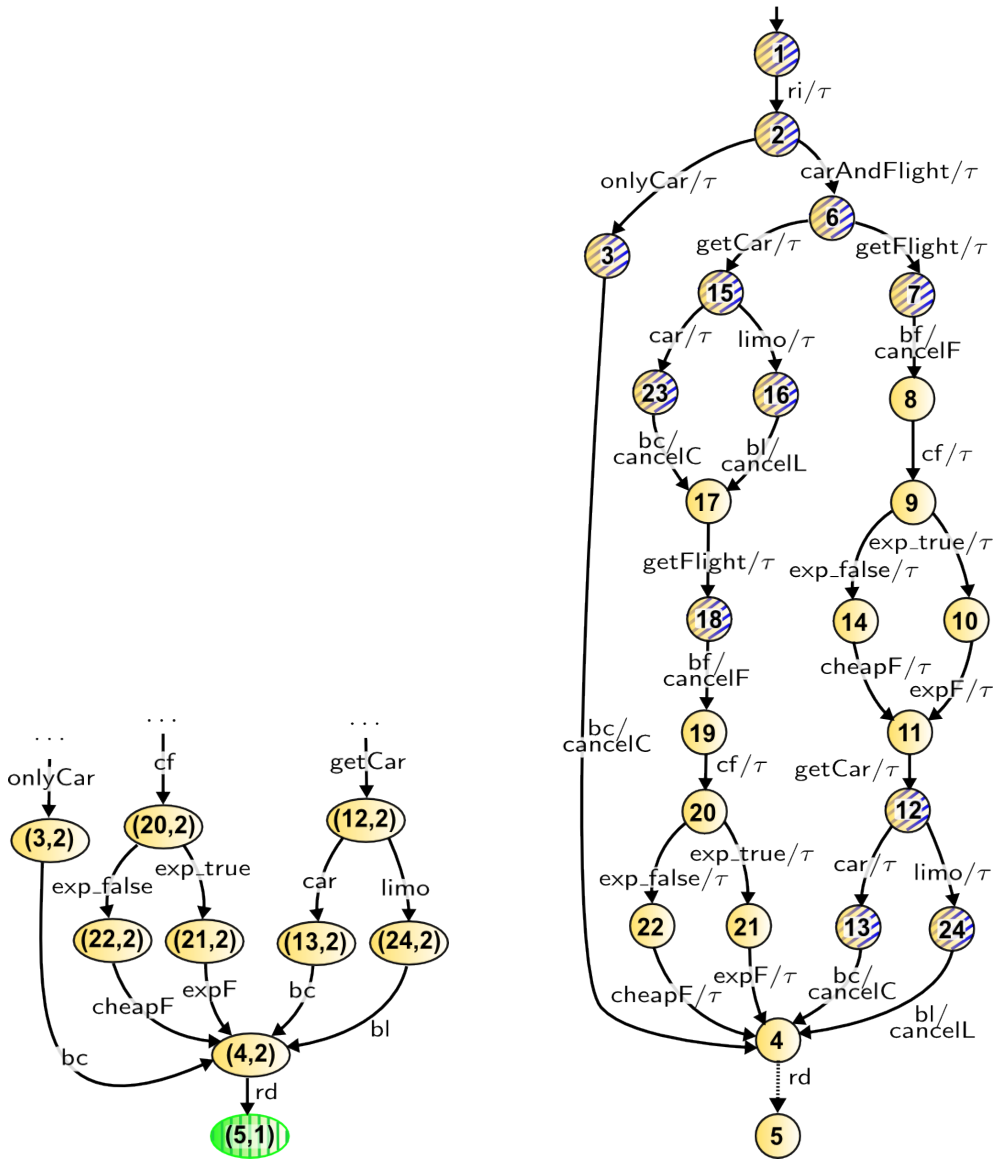
The last part of the preprocessing phase *statically* identifies strategic behaviours of the application $\mathsf{L(B)}$, aimed to help find an efficient recovery plan when a violation is encountered. Goal transitions, defined in Section 4.1.3.1, are applications transitions that may result in the (immediate) satisfaction of some properties. Change states, defined in Section 4.1.3.2, are application states from which alternative behaviours can be executed. See Sections 4.3 and 4.4 for details about how these application transitions and states are used to compute recovery plans.

4.1.3.1 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 satisfaction of some properties. We compute these on a per-property basis. Further, recall that only liveness properties can be satisfied, which is indicated by the monitor reaching a green state; safety properties can only be violated. Thus, for each mixed monitor $\mathsf{A}^\ell_\mathsf{i} \in \mathsf{LM} = (S_i, \Sigma, \delta_i, I_i, F_i)$, we are looking for transitions in $\mathsf{L(B)} = (S, \Sigma, \delta, I)$ corresponding to $\mathsf{A}^\ell_\mathsf{i}$ entering its green state(s). To find those, we compute the cross-product $\mathsf{L(B)} \times \mathsf{A}^\ell_\mathsf{i}$.

Definition 4.4 (Goal Transition)**.** *A transition* $(s, a, s') \in \delta$ *is a* goal transition *iff* $\exists q, q' \in S_i \cdot (s, q) \xrightarrow{a} (s', q') \in \delta_{\mathsf{L(B)} \times \mathsf{A}^\ell_\mathsf{i}} \wedge$ colour$(q) \neq$ *green* $\wedge$ colour$(q') =$ *green.*

That is, $s \xrightarrow{a} s'$ corresponds to taking a transition on a into a green state of $\mathsf{A}^\ell_\mathsf{i}$. The resulting set of goal transitions is denoted by $\mathsf{G}(\mathsf{B}, \mathsf{A}^\ell_\mathsf{i})$:



(a) (b)

Figure 4.5: (a) A fragment of $\mathsf{L}(\mathtt{TAS}) \times \mathtt{A}_1$; (b) LTS $\mathsf{L}(\mathtt{TAS})$, where goal transitions are depicted by tiny-dashed transitions and change states are shaded diagonally in purple.

For example, consider a fragment of $\mathsf{L}(\mathtt{TAS}) \times \mathtt{A}_1$ shown in Figure 4.5a. The green state of $\mathtt{A}_1$ is state 4, with transition on rd leading to it. The only transition in $\mathsf{L}(\mathtt{TAS}) \times \mathtt{A}_1$ satisfying the above definition is $(4,2) \xrightarrow{\mathsf{rd}} (5,4)$, and thus $\mathsf{G}(\mathtt{TAS}, \mathtt{A}_1) = \{(4, \mathsf{rd}, 5)\}$ (depicted by tiny-dashed transitions in Figure 4.5b).

When computing recovery plans, we need to direct the application towards taking its goal transitions.

4.1.3.2 Change States

Given an erroneous run, how far back do we need to compensate before resuming forward computation? If we want to avoid repeating the same error again, we need the application to take an alternative path. States of $\mathsf{L}(\mathsf{B})$ that have actions executing which can potentially produce a branch in control flow of the application are called *change states*.

Flow-changing actions are user choices, states modelling the <flow> activity (since each pass through this state may produce a different interleaving of actions), and those service calls whose outcomes are not completely determined by their input parameters but instead depend on the implicit state "of the world". 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.

Definition 4.5 (Change State)**.** *A state is a* change state *if it is identified by: 1) a <flow> activity, 2) a <pick> activity, or 3) a non-idempotent service call.*

We denote by $\mathsf{C}(\mathsf{B})$ the set of all change states in the LTS of the application B. For example, in the LTS in Figure 4.3, state 6 corresponds to the <flow> activity and represents the different serialization order of the branches. States 2, 12 and 15 model user choices. Non-idempotent partner calls are bf, bc, bl, and thus

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

identified in Figure 4.5b by purple diagonal shading.

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

Of course, it is possible that the computed recovery plan passes through a change state which does not affect its outcome, i.e., is irrelevant to the encountered error and its fix. We address computation of "relevant" change states in Chapter 6, Section 6.4.1.

4.2 Runtime Monitoring

In this section, we discuss two types of monitoring for BPEL applications: eavesdropping and monitoring for recovery, in Sections 4.2.1 and 4.2.2, respectively. Eavesdropping is a passive monitoring technique, i.e., the application continues execution even though a property violation has been detected. On the other hand, monitoring for recovery is an active technique, as it delays events that may cause property violations (and activates recovery when a violation is detected).

4.2.1 Eavesdropping

The runtime monitoring component of our framework uses the set of monitors to analyze the BPEL program B as it runs on a BPEL-specific Application Server. This component is based on the runtime monitoring framework developed by Yuan Gan as part of her M.Sc. thesis [37], which has been implemented within the IBM WebSphere business integration products [49]. In Gan's framework, the *interception mechanism* used is *eavesdropping* – watching events as they pass between partners and updating monitors accordingly. In other words, events in Σ are captured as they pass between the application server and the program, and then immediately used to update the state of the monitors (events are not stored). 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.

Experience: Monitoring the Loan Application System. The Loan Application System (`LAS`), introduced in Chapter 3, Section 3.3.3, automates loan application processing: the system uses a set of business rules to make decisions about loan applications. The system takes as input a customer taxpayer id and the desired loan amount, producing as output one of the following statuses: automatically accepted, automatically rejected, or flagged for manual authorization. This system has been implemented using BPEL (see Chapter 3, Figure 3.13). Table 3.1 shows some properties of the `LAS` system that can be expressed using W-SD, and Table 3.2 shows the sizes of the corresponding W-SD monitors. In this section, we report our experience on monitoring the application (without recovery).

Since `LAS` is a *sample* application, its original developers simplified some of the business logic, e.g., the `CreditCheck` component generates random credit scores rather than access the credit bureau. We began by testing the system to see if the application was correctly deployed. To do this, we ran it on two different taxpayer ids and three different loan amounts, with the following specific input configurations:

$$\begin{aligned}
c_1 &= \text{<taxpayer id = 1234, loan amount = \$10,000>},\\
c_2 &= \text{<taxpayer id = 1234, loan amount = \$60,000>},\\
c_3 &= \text{<taxpayer id = 1888, loan amount = -\$1,000>}.
\end{aligned}$$

As the system is supposed to generate random valid credit scores, we ran the system 10 times with each configuration. For configuration c_1, we expected to see some automatic approvals of the loan, and some rejections, based on whether the good or the bad score is generated. For c_2, we expected some manual approvals of the loan (the loan amount is above the automatic approval limit), and some rejections. Finally, since the loan amount in c_3 is invalid, we expected to see only loan rejections.

For configurations c_1 and c_2, the behaviour we observed was as expected: $P_1^{\mathtt{LAS}}$, $P_2^{\mathtt{LAS}}$, $P_5^{\mathtt{LAS}}$

always held and $P_3^{\texttt{LAS}}, P_4^{\texttt{LAS}}$ held when the loan was granted. However, for all executions of c_3, the system automatically approved the loan, meaning that properties $P_3^{\texttt{LAS}}$ and $P_4^{\texttt{LAS}}$ were violated. For all executions of c_3, the system produced the following faulty execution trace:

$$\begin{aligned} FT \;=\; & (\texttt{MnPs},\ \mathsf{ckCtSe},\ \texttt{MnPs}), (\texttt{MnPs},\ \mathsf{ctSeOK},\ \texttt{CtCk}), (\texttt{MnPs},\ \mathsf{ckLnAt},\ \texttt{MnPs}),\\ & (\texttt{MnPs},\ \mathsf{lnAtNO},\ \texttt{CtCk}), (\texttt{MnPs},\ \mathsf{ceLn},\ \texttt{CeLn}) \end{aligned}$$

where each triple $(Sender, m, Receiver)$ denotes a partner *Sender* sending a message m to a partner *Receiver*. The $(\texttt{MnPs}, \mathsf{lnAtNO}, \texttt{CtCk})$ triple in this trace indicates that the loan amount is less than or equal to zero. In other words, the main process MnPs checked the predicate "loan amount is > 0", and sent a loanAmountNotOkay (lnAtNO) message because the predicate did not hold. Therefore, this trace depicts a failure of $P_3^{\texttt{LAS}}$ because it includes an invalid behaviour, the acceptance of the invalid loan, indicated by the subtrace $(\texttt{MnPs}, \mathsf{ckLnAt}, \texttt{MnPs}), (\texttt{MnPs}, \mathsf{lnAtNO}, \texttt{CtCk}), (\texttt{MnPs}, \mathsf{ceLn}, \texttt{CeLn})$. As $P_4^{\texttt{LAS}}$ is a scoped version of $P_3^{\texttt{LAS}}$, it also fails on this trace.

To identify the cause of the violations, we examined the BPEL diagram in Figure 3.13 to see that the trace FT is produced if the LAS system obtains the taxpayer's credit score, checks if the credit score is greater than 750 (ScoreEvaluation), checks if the loan amount is greater than zero (input validation), and checks if the loan amount is less than \$50,001 (AutoApprovalTest). The ScoreEvaluation should only occasionally be true, as the CreditCheck component generates random credit scores. However, we obtained trace FT every time the system was run with the taxpayer id 1888, i.e., the system always approved a negative loan.

We traced this behaviour to two problems. The first, identified after looking at the BPEL code of the LAS system, was that the application did not use the results of the input validation, allowing requests for negative loans to go through. The second problem was only identified after examining the source code for the CreditCheck partner. Instead of ignoring the taxpayer id and generating a random credit score, this component always returns a good credit score when the taxpayer id ends with “888”. Combined, these two problems yielded the approval of the loan for configuration c_3 every single time.

Overall, our experience showed that the system can handle simultaneous failure of several monitors and allowed us to specify interesting properties which led to the discovery of two real faults in the LAS system.

4.2.2 Monitoring for Recovery

While adequate for identifying and reporting property violations, eavesdropping 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 Chapter 5, Section 5.2.2 for details).

Another difference with [37] is that we store intercepted events. For the LTS of the application $\mathsf{L}(\mathsf{B}) = (S, \Sigma, \delta, I)$, we store the trace of the execution:

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

We say that T is a *successful* trace iff $\forall \mathsf{A_i} \in \mathsf{A}$, $a_0 a_1 \dots a_{n-1}$ is rejected by $\mathsf{A_i}$. T is a *failure* (or an *error*) trace iff $\exists \mathsf{A_i} \in \mathsf{A}$ s.t. $a_0 a_1 \dots a_{n-1}$ is accepted by $\mathsf{A_i}$. In such a case, state s_n is an *error state* of the application. In addition to T, we also store traces $\mathsf{T}^{\mathsf{A_1}} \dots \mathsf{T}^{\mathsf{A_n}}$ that correspond to the executions of the *monitors* $\mathsf{A_1} \dots \mathsf{A_n}$, respectively. These are used in the recovery phase (see Sections 4.3 and 4.4). 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

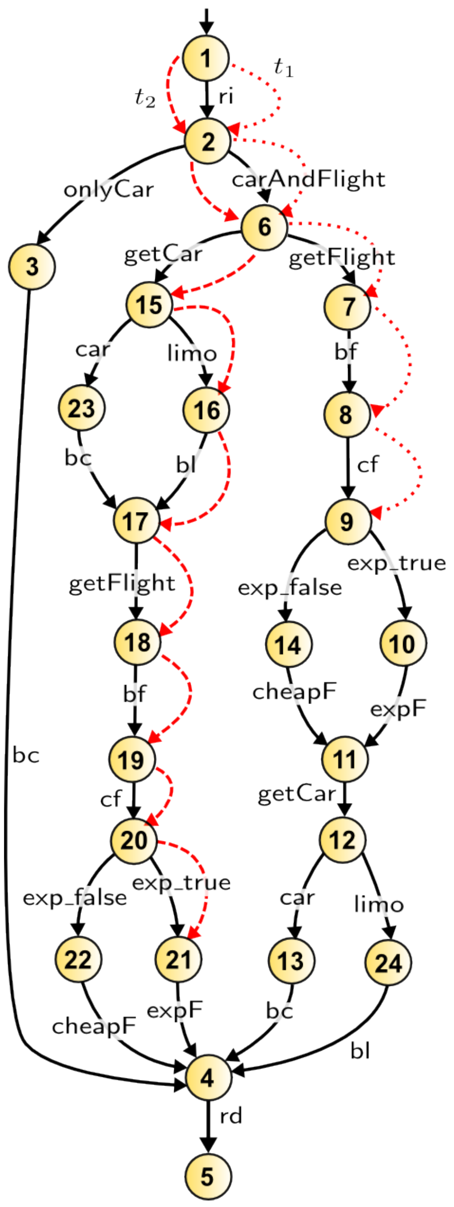


Figure 4.6: LTS L(TAS), showing traces t_1 (dotted) and t_2 (dashed).

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 4.6. In addition to t_1, our tool stores $t_1^{\mathtt{A}_1}$ and $t_1^{\mathtt{A}_2}$ – the corresponding traces of the enabled monitors:

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

The application server detects that the cf invocation timed out, and sends a TER event (not shown in Figure 4.6) to the application. Our framework intercepts this TER event and determines that executing it turns t_1 into a failing trace, because the monitor $\mathtt{A}_1$ would enter its error (red) state 3. In response, our framework does not deliver the TER event to the application, and instead initiates recovery.

In another scenario, the customer attempts to arrive at her destination via a limo (bl) and an expensive flight (expF). This corresponds to the trace t_2, depicted by dashed transitions in Figure 4.6 (the traces of the monitors are omitted):

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

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.

4.3 Recovery Plans for Safety Property Violations

Once an error has been detected during runtime monitoring, the goal of the recovery phase is to suggest a number of *recovery plans* that would lead the application away from the error.

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

(a)

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

(b)

Figure 4.7: (a) Plans for TAS for recovery from the safety violation of trace t_2; (b) XML version of recovery plan r_{18}.

Definition 4.6 (Plan)**.** *A* plan *is a sequence of actions. A* BPEL recovery plan *is a sequence of actions consisting of user interactions, compensations (empty or not) and calls to service partners.*

Recovery plans differ depending on the type of property that failed. We treat safety properties below, and recovery from liveness properties is described in Section 4.4.

4.3.1 Computing Plans

The recovery procedure for a safety property violation receives $\mathsf{L_C(B)}$ – the LTS of the running application B with compensation (see Section 4.1.1), T – the executed trace ending in an error state e (see Section 4.2) and $\mathsf{C(B)}$ – the set of change states (see Section 4.1.3.2).

In order to recover, we need to "undo" a part of the execution trace, executing available compensation actions, as specified by δ_c. We do this until we either reach a state in $\mathsf{C(B)}$ or the initial state of $\mathsf{L_C(B)}$. Multiple change states can be encountered along the way, thus leading to the computation of multiple plans.

For example, consider the error trace t_2 described in Section 4.2 and shown in Figure 4.6. $\{1, 2, 6, 15, 16, 18\}$ are the change states seen along t_2. This leads to the recovery plans shown in Figure 4.7a. We add state names between transitions for clarity and refer to plans as 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 the definition of the latter. For example, recovery to state 16 starts with recovery to state 18 and then includes two more backward transitions, the last one with a non-empty compensation. Plan r_{18} can avoid the error if, after its application, the user chooses a cheap flight instead of an expensive one. Executing plan r_{15} gives the user the option of changing the limousine to a rental car, and plan r_2 – the option of changing from an air/ground combination to just renting a car. Both of these behaviours do not cause the violation of A_2.

Computed plans are then converted to BPEL for presentation to the user. For example, plan r_{18} is shown in Figure 4.7b. The chosen plan can then be applied (see Section 5.2.3), allowing the program to continue its execution from the resulting change state.

4.3.2 Analysis

In what follows, let B be a BPEL application, $\mathsf{L}(\mathsf{B}) = \{S, \Sigma, \delta, I\}$ be the LTS that represents B, and $\mathsf{C}(\mathsf{B})$ be the set of change states of application B. Let $\mathsf{L}_\mathsf{C}(\mathsf{B}) = \{S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I\}$ be the LTS of application B with compensation, where Σ_c is the set of compensation actions and δ_c the set of compensation transitions, computed as described in Section 4.1.1.

Let $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ be an execution trace, where $s_n = e$ (the error state), A_s be the safety monitor that identified the violation, and $r_i = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_i} s_i$ be a safety recovery plan for T that leaves the application in state s_i, where c_k compensates a_k and $s_i \in \mathsf{C}(\mathsf{B})$. Finally, let $\mathsf{C}_\mathsf{T}(\mathsf{B})$ be the set of change states that appear in T: $\mathsf{C}_\mathsf{T}(\mathsf{B}) = \{s | (s, a, s') \in \mathsf{T} \wedge s \in \mathsf{C}(\mathsf{B})\}$.

Definition 4.7 (Adequate Compensation)**.** *Let* B *be a BPEL application and* $\mathsf{L_C}(\mathsf{B}) = \{S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I\}$ *be the LTS of application* B *with compensation (as defined above). A transition* $(s, a, s') \in \delta$ *has* adequate compensation *when used in application* B *if in the representative LTS* $\mathsf{L_C}(\mathsf{B})$ *there is a transition* $(s', a_C, s) \in \delta_C$*, where* a_C *is the compensation action for* a*. Actions compensated by* τ *(the empty action) are always adequately compensated.*

Definition 4.8 (Compensated Execution Trace)**.** *Let* $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ *be an execution trace, where* $s_n = e$ *(the error state), and* $r_i = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_i} s_i$ *be a safety recovery plan for* T*. The* compensated execution trace T_{r_i} *is the result of applying* r_i *to* T*, assuming that none of the participating services timeout. In other words,* $\mathsf{T}_{r_i} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{i-1}} s_i$*.*

Proposition 4.1. *Let* $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ *be an execution trace, where* $s_n = e$ *(the error state),* A_s *be the safety monitor that identified the violation, and* $r_i = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_i} s_i$ *be a safety recovery plan for* T*.*

If compensation for actions $a_i, ..., a_{n-1}$ *is adequate, and the default BPEL compensation order is observed at runtime, then the compensated execution trace* T_{r_i} *will be the i-length prefix of* T*, i.e.,* $\mathsf{T}_{r_i} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{i-1}} s_i$*, where* $s_i \in \mathsf{C}(\mathsf{B})$*. State* s_i *is a change state because of the definition of recovery plans for safety violations.*

This proposition follows from Definitions 4.7 and 4.8. Since the default BPEL compensation respects the forward execution order during compensation, compensating the tail of the execution trace leaves the application in the state from which the tail sequence of actions was executed.

The exact number of plans is determined by the number of change states encountered along the trace. 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").

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

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

Finally, 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 4.1.3.2 treats all non-idempotent service calls as the same, whereas not all might be *relevant* to the satisfaction of properties of interest. In Chapter 6, Section 6.4.1 we discuss the computation of relevant change states, as well as evaluate the effectiveness of this technique in reducing the number of plans computed.

4.4 Recovery Plans from Mixed Property Violations

Failure of a mixed monitor during execution means that some required actions have not been seen before the application tried to terminate, and the recovery plan should attempt to perform these actions.

The recovery procedure receives:

- A^ℓ, the monitor that identified the violation,
- $\mathsf{L}_\mathsf{C}(\mathsf{B})$, the LTS of the application,

- $\mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)$, the set of goal transitions corresponding to A^ℓ,
- T, the executed trace ending in an error state e, and
- $\mathsf{C}(\mathsf{B})$, the set of change states.

A recovery plan effectively "undoes" actions along T, starting with e and ending in a change state (otherwise, the plan would not be executable!) and then "re-plans" the behaviour to reach the goal (see Figure 4.1 for a schematic view of the overall process). Our solution adapts techniques from the field of *planning* [29], described in the rest of this section.

4.4.1 Recovery as a planning problem

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

In addition to P, a planner often gets as input k – the length of the longest plan to search for, and applies various search algorithms to find a plan of actions of length $\leq k$, starting from 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 recovery problem into a planning problem, we use $\mathsf{L}_\mathsf{C}(\mathsf{B})$ as the domain and e as the initial state. The third component needed is a set of goal states. Recall that $\mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)$ is a set of goal *transitions*. We define $\mathsf{G}_\mathsf{s}(\mathsf{B}, \mathsf{A}^\ell) = \{s \mid \exists a, s' \cdot (s, a, s') \in \mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)\}$. That is, $\mathsf{G}_\mathsf{s}(\mathsf{B}, \mathsf{A}^\ell)$ is a set of *sources* of transitions in $\mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)$. We can now define the planning problem

$$\mathsf{P}(\mathsf{B}, \mathsf{A}^\ell, \mathsf{T}) = (\mathsf{L}_\mathsf{C}(\mathsf{B}), \mathsf{e}, \mathsf{G}_\mathsf{s}(\mathsf{B}, \mathsf{A}^\ell))$$

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 $(s, a, s') \in \mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)$. For example, consider the trace t_1 of Figure 4.6, described in Section 4.2, in which monitor $\mathtt{A_1}$ fails. We define

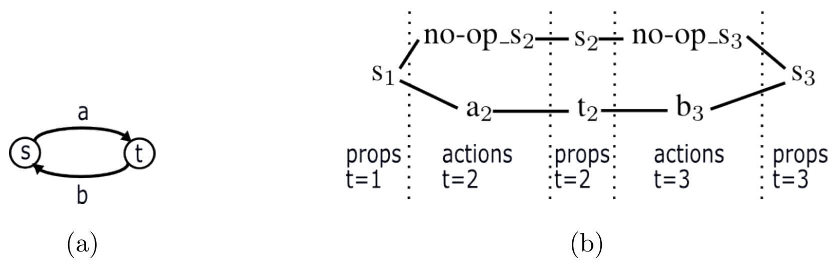


Figure 4.8: (a) a simple LTS and (b) its encoding as the planning graph of size 3.

the planning problem $\mathsf{P}(\texttt{TAS}, \texttt{A}_1, t_1) = (\mathsf{L_C}(\texttt{TAS}), 9, \{4\})$, where 9 is the initial state (see Figure 4.3) and $\mathsf{G_s}(\texttt{TAS}, \texttt{A}_1) = \{4\}$ (see Section 4.1.3.1). The resulting plan p should be expanded to $p \xrightarrow{\mathsf{rd}} 5$.

Unfortunately, not every trace returned by solving $\mathsf{P}(\mathsf{B},\ \mathsf{A}^{\ell}, \mathsf{T})$ is acceptable: the recovery plans for liveness violations should also go through change states. Thus, we cannot simply use a planner as a “black box”.

Instead, we look at how planners encode the planning graph and then manipulate the produced encoding directly, to add additional constraints. Consider the LTS in Figure 4.8a, which is the planning domain, with s as both the initial and the goal state. The planning graph expanded up to length 3 is shown in Figure 4.8b and is read as follows: at time 1 we begin in state s_1. If action a occurs (modelled as a_2), then at time 2 we move to state t (modelled as proposition t_2 becoming true); otherwise, we remain in state s (i.e., proposition s_2 is true). If action b occurs while we are in state t (modelled as b_3), then at time 3 we move to state s. Two plans of length 2 are extracted from this graph: a_2, b_3, corresponding to executing a first, followed by b, and “do nothing” – a planner-specific treatment of a sequence of no-ops.

Several existing planners, such as BlackBox [54], translate the planning graph into a CNF formula and then use a SAT solver, such as SAT4J [12], to find a satisfying assignment for it. Such an assignment, if found, represents a plan. For example, the CNF encoding of the planning graph in Figure 4.8b is as follows:

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

Note that it explicitly models pre- and post-conditions of the execution of actions. Such a formula is passed to a SAT solver which produces a satisfying assignment s, if one exists. The desired plan is extracted from s by taking propositions that correspond to actions and that are assigned positive values in s. For the above example, these are a_2, b_3 and "do nothing".

Our approach has been inspired by existing work a related problem – that of automatically creating new web service compositions that accomplish non-trivial tasks [67, 73, 68, 94]. In this case, the planning domain is the set of available web services, the goal is a specification of the desired behaviour, and plans are service compositions that accomplish the desired behaviour. Research in this area has focused on using different planning techniques to solve this problem in an efficient manner, dealing with the non-deterministic behaviour of web services, the partial observability of their internal status, and the specification of complex goals expressing temporal conditions and preference requirements. In this thesis, we do not use planning to generate new service compositions, but use it instead to explore existing applications. However, the approach presented here can be augmented with the work presented in [73, 68, 94], especially in the case when there is not enough compensation or redundancy in the application to permit the computation of recovery plans according to our approach.

In what follows, we first discuss how to produce a single recovery plan using a SAT-based approach (Section 4.4.2) and then show how to extend it to produce multiple plans (Section 4.4.3).

4.4.2 Producing a single recovery plan

Let $\mathsf{f_P}$ be the encoding of the planning problem $\mathsf{P}(\mathsf{B}, \mathsf{A}^\ell, \mathsf{T})$ produced by an existing planner. We augment $\mathsf{f_P}$ to follow our "undo until a change state and then redo" approach by adding conjuncts to $\mathsf{f_P}$ with the purpose of restricting its solutions. For efficiency, some additional filtering is done after all plans have been computed (see Section 4.4.5).

1. We want to make sure a recovery plan visits at least one of the change states encountered on the execution trace T. Let $\mathsf{S}(\mathsf{T})$ be the set of states on T. We define $\mathsf{C}(\mathsf{T}) = \mathsf{S}(\mathsf{T}) \cap \mathsf{C}(\mathsf{B})$ to be the change states that appear on T and denote by $c^1, ..., c^n$ the propositions that correspond to states in $\mathsf{C}(\mathsf{T})$. If k is the maximum length of the plan which is being searched for, propositions c_1^j, c_2^j, ..., c_k^j correspond to expansions of c^j to times 1 ... k. For example, consider Figure 4.8 again. If t is a change state and $k = 3$, then propositions t_1, t_2, t_3 in $\mathsf{f_{lts}}$ correspond to expansions of t to times 1, 2, 3. We define $\mathsf{c} = (c_1^1 \vee ... \vee c_k^1 \vee ... \vee c_1^n ... \vee c_k^n)$, or, in the case of our example, $\mathsf{c} = (t_1 \vee t_2 \vee t_3)$. This formula is true when at least one of the change states in $\mathsf{C}(\mathsf{T})$ is part of the plan.

2. In order to further lead the planner towards the "undo and then redo" plans, we want to make sure that the only compensations used in the plan correspond to actions in the original trace T. More formally, let $\mathsf{T_C}$ be the set of compensation actions corresponding to the actions in T, and let $\Sigma^c \setminus \mathsf{T_C}$ be all other compensation actions. Let a be a formula which excludes (timed versions) of actions in $\Sigma^c \setminus \mathsf{T_C}$: i.e., neither of these compensation actions is true at any step in the plan. For example, for trace t_1 over the LTS $\mathsf{L_C}(\mathsf{TAS})$ (see Figure 4.3), formula a would exclude all compensations except $\mathsf{cancelF}$ and τ.

We now build a new propositional formula, based on $\mathsf{f_P}$:

$$\mathsf{R_0}(\mathsf{f_P}) = \mathsf{f_P} \wedge \mathsf{c} \wedge \mathsf{a}$$

$\mathsf{R_0}(\mathsf{f_P})$ describes the original planning problem for $\mathsf{P}(\mathsf{B}, \mathsf{A}^\ell, \mathsf{T})$, and in addition requires that at least one of the change states is visited and no compensation actions for events

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

(a)

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

(b)

Figure 4.9: (a) Plans for TAS of length $\leq$ 10 for recovery from the mixed property violation of trace t_1; (b) XML version of recovery plan p_0.

that did not occur in T appear in the plan.

4.4.3 Producing multiple recovery plans

Let π_0 be the plan produced for $\mathsf{R}_0(\mathsf{f}_\mathsf{P})$ (see Section 4.4.2), leading to a goal state $g \in \mathsf{G}_\mathsf{s}(\mathsf{B}, \mathsf{A}^\ell)$. To give the user options for recovery, we want to produce other plans, different from π_0. The simplest way to do this is to remove g from $\mathsf{G}(\mathsf{B}, \mathsf{A}^\ell)$ and repeat the process described in Section 4.4.2. The new plan will necessarily lead to a different goal transition and thus will be different from π_0. However, this method cannot produce multiple plans to the *same* destination.

Instead, we constrain $\mathsf{R}_0(\mathsf{f}_\mathsf{P})$ to explicitly rule out π_0. For example, to rule out the plan a, b for the LTS in Figure 4.8a, we use $\mathsf{R}_0(\mathsf{f}_\mathsf{lts})$ computed in Section 4.4.2 and modify it as

$$\mathsf{R}_1(\mathsf{f}_\mathsf{lts}) = \mathsf{R}_0(\mathsf{f}_\mathsf{lts}) \wedge (\neg a^2 \vee \neg b^3)$$

This guarantees that the plan, if found, is different from the previously found one in at least one action. This encoding does not prevent the generation of time-stuttering equivelent plans. For example, if the planning graph corresponding to the LTS in Figure 4.8a is expanded to 4 or more time steps, then the plan ($\text{no-op\_}s_2 \wedge a^3 \wedge b^4$) is also generated. This is the same plan as ($a^2 \wedge b^3$), except delayed by one time step. In order to avoid generating time-stuttering equivalent versions of the same plan, we must add stronger constraints, e.g., plans cannot contain no-op values and must start at time 2. We leave this as future work.

We continue this way, restricting $\mathsf{R_i}(\mathsf{f_P})$ with the set of previously computed plans to get $\mathsf{R_{i+1}}(\mathsf{f_P})$, until the number of desired plans is reached or until no new plan can be found, that is, $\mathsf{R_j}(\mathsf{f_P})$ is not satisfiable for some j.

We now apply this method to the TAS problem and the error trace t_1 shown in Figure 4.3 and ending in state 9. Looking for plans up to length 10, we get plans p_0, p_1 and p_2 shown in Figure 4.9a. And, as mentioned earlier, each plan is extended with the last goal transition $4 \xrightarrow{\mathsf{rd}} 5$.

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 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.

The produced plans are than ranked based on the length of the plan and the cost of compensation actions in it. For example, plan p_0 is the shortest and the additional compensation, for action carAndFlight, is of zero cost. Thus, it is ranked the highest. Of course, this plan does not take into account the time the user will spend driving rather than flying, 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 4.7b, and the re-planning part consists of a sequence of BPEL <invoke> operations. The XML translation of plan p_0 is shown in

Figure 4.9b.

4.4.4 Analysis

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

In what follows, let B be the BPEL application, $\mathsf{L}(\mathsf{B}) = \{S, \Sigma, \delta, I\}$ be the LTS that represents B, and $\mathsf{C}(\mathsf{B})$ be the set of change states of application B. Let $\mathsf{L}_{\mathsf{C}}(\mathsf{B}) = \{S, \Sigma \cup \Sigma_c, \delta \cup \delta_c, I\}$ be the LTS of application B with compensation, where Σ_c is the set of compensation actions and δ_c the set of compensation transitions, computed as described in Section 4.1.1.

Let $\mathsf{T} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$, where $s_n = e$, be a trace of the program leading to an error, A_ℓ be the mixed monitor that detected the violation and $\mathsf{G_s}(\mathsf{B}, \mathsf{A}^\ell)$ be the set of goal transitions corresponding to A_ℓ. Let $p = (p_i, p_m)$ be a mixed recovery plan for T that tries to lead the application to the goal state s_m through the change state s_i. The first part of the plan, p_i, compensates trace T, leaving the application in state s_i, and the second part, p_m, is a trace that leads to the state $s_m \in \mathsf{G_s}(\mathsf{B}, \mathsf{A}^\ell)$ when executed from s_i, i.e., $p_i = s_n \xrightarrow{c_{n-1}} s_{n-1} \xrightarrow{c_{n-2}} ... \xrightarrow{c_i} s_i$, where c_k compensates a_k and $s_i \in \mathsf{C}(\mathsf{B})$, and $p_m = s_i \xrightarrow{b_0} s_j \xrightarrow{b_1} s_{j+1} \xrightarrow{b_2} ... \xrightarrow{b_{m-1}} s_m$.

Let T_{p_i} be the compensated execution trace resulting from the application of p_i to T according to Definition 4.8. If we assume that compensation for actions $a_i, ..., a_{n-1}$ is adequate, and that the default BPEL compensation order is observed at runtime, then, according to Proposition 4.1, the execution of p_i leaves the application in state s_i.

Definition 4.9 (Updated Execution Trace). *Let* T *be an execution trace,* $p = (p_i, p_m)$ *be a mixed recovery plan and* T_{p_i} *be the compensated execution trace (as defined above). The* updated execution trace T_{p_m} *is the result of applying the sequence of actions associated* *to p_m to the compensated execution trace T_{p_i}, assuming that none of the participating services timeout.*

In other words, the updated execution trace T_{p_m} is the result of executing the sequence $b_0 b_1 ... b_m$ from state s_i. Note that BPEL applications considered in this thesis have several sources of non-determinism (from <pick> and <flow> activities). In the proposition below, we define suficient conditions under which the trace produced as a result of executing p_m reaches the goal state.

Proposition 4.2. *Let T be an execution trace, $p = (p_i, p_m)$ be a mixed recovery plan and T_{p_m} be the updated execution trace (as defined above).*

If compensation for actions $a_i, ..., a_{n-1}$ is adequate, the default BPEL compensation order is observed at runtime, the user acts as suggested by the plan in the case of external choices, and the suggested <flow> activity interleavings are executed, then the updated execution trace T_{p_m} is the result of compensating actions $a_i a_{i+1} ... a_{n-1}$, leaving the applicatin in state s_i, and then executing p_m, leaving the application in state s_m, i.e., $\mathsf{T}_{p_m} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{i-1}} s_i \xrightarrow{b_0} s_j \xrightarrow{b_1} s_{j+1} \xrightarrow{b_2} ... \xrightarrow{b_{m-1}} s_m$, where $s_i \in \mathsf{C}(\mathsf{B})$ and $s_m \in \mathsf{G_s}(\mathsf{B}, \mathsf{A}^\ell)$.

This proposition follows from Definitions 4.7 and 4.8, as well as the fact that we expect both the user and the BPEL engine to execute the suggested actions. Of course, ensuring that a particular execution of the <flow> is chosen is difficult – and in fact, often unnecessary. A better approach might be to make property automata more permissive to allow them to reach green states under different orderings of relevant events. However, we leave such cases as future work, and formally define successful plan execution below.

Definition 4.10 (Successful Mixed Recovery Plan)**.** *Let T be an execution trace, $p = (p_i, p_m)$ be a mixed recovery plan and T_{p_m} be the updated execution trace (as defined above). A mixed recovery plan p is* successful *on an execution trace T iff the execution* *of the updated execution trace T_{p_m} on the mixed monitor A_ℓ (the monitor that detected the violation) leaves A_ℓ in a state s, where* $\text{colour}(s) = $ *green.*

Worst Case Analysis. SAT-based planning is an NP-hard problem [54]. However, due to advances in the SAT community, checking satisfiability has a good average case performance, allowing the solution of problem instances involving tens of thousands of variables and millions of constraints.

As with violations of safety properties, the maximum number of plans we compute for mixed violations is controlled by the user either directly or indirectly, by controlling the maximum plan length. In the second case, the maximum number of plans is also indirectly determined by the number of change states encountered along the trace and the number of goal transitions reachable from these change states. If n_c change states and n_g goal transitions are reachable from the error state within k steps, then the maximum number of plans of at most length k is $n_c \times n_g$. We check the satisfiability of an increasingly larger SAT instance in order to compute each new plan, since we add a set of constraints for each plan found. This process continues until all plans of length k are found, or the maximum number of plans is reached. In the average case, we expect that the maximum number of plans will be much smaller than the number of application change states and goal transitions. As in the case of safety property violations, we expect that the application developer will limit the maximum number of plans to a small number (≤ 5) to avoid overwhelming users with a large number of plans. The maximum plan length should depend on how far apart the application goal transitions are, since we want to ensure that at least one goal transitions is reachable from each possible error state.

4.4.5 Discussion

Precision. Our treatment of goal transitions effectively means that we model satisfaction of the required sequence of actions of a liveness property by executing the last event in the sequence. Thus, our approach may include some plans that do not result in the satisfaction of the desired property (we did not encounter this problem in the examples reported in Section 6). One way to approach this problem that we intend to investigate in the future is to define goal *traces*, based on the computation tree of $\mathsf{L}(\mathsf{B})$. While this will lead to the extra precision in plan generation, we expect to pay a potentially steep price in performance.

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. We describe and evaluate these optimizations in Chapter 6, Section 6.4.

Controlling unnecessary compensations. Plans p_1, p_2 and p_3 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 determined 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 paper is reasonable but intend to revisit this issue as we gain more experience with the approach.

Furthermore, as plan lengths get large, the planner can generate plans with *compensation loops* which involve doing an action and then immediately undoing it. For example, in recovering from a violation in trace t_1 in LTS $\mathsf{L}_\mathsf{C}(\mathsf{TAS})$, shown in Figure 4.6, the plan may include booking a flight and then cancelling it several times (i.e., going between states 7 and 8 of $\mathsf{L}_\mathsf{C}(\mathsf{TAS})$. Clearly, such situations should be avoided. We could have encoded a corresponding formula as the SAT problem, conjoining it to $\mathsf{R}_0(\mathsf{f}_\mathsf{P})$: "at any point in the plan, when a non-compensatory action appears, all follow-on actions should not include compensation". However, we feel that this modification should make SAT computation significantly less efficient. Instead, we filter computed plans so that the ones with compensation loops are not presented to the user.

Can generated plans still fail? There are a number of reasons our plans can fail. The first one, addressed earlier in this subsection, are 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 probably be addressed using additional static analysis.

4.5 Monitoring and Recovery for Other Languages

Our interest in BPEL was motivated by our interaction with IBM. However, BPEL is a very verbose language, making large BPEL programs hard to read/debug, and BPEL tool support leaves a lot to be desired (e.g., memory-intensive IDEs, non-standard visual representation of the language). There is also the issue of its formal semantics: the official BPEL specification is in natural language. Subsets of the language have been formalized using, for example, labelled transition systems [30], Petri nets [96] and guarded automata [16]. However, certain parts of the language (like compensation) have not been formally specified. For example, in this thesis, we only formalized simple event-based compensation without data. For these reasons, as well as our belief that our monitoring and recovery framework is of general interest, we are interested in extending our framework to other languages.

For example, consider the specific case of Orc [56, 71], a general purpose programming language that allows the concise encoding of concurrent and distributed applications. Orc was originally presented as a process calculus, but has now evolved into a full programming language. *Sites* provide uniform access to local and external services – a site is a proxy for either a Java class (allowing access to the methods of the instanced class) or a web service (allowing access to the service through its published interface). Using simple concurrency primitives, programmers can orchestrate the invocation of sites to achieve a goal, while managing timeouts, priorities, and failures. van der Aalst et al. [98] proposed a set of workflow patterns that characterize the kinds of control flow that appear frequently in workflow processes. Most of these patterns are available in both BPEL [104] and Orc [21].

Like BPEL, Orc allows both synchronous and asynchronous communication with web services (through sites). Kitchin et al. [56] defined Orc's semantics using labelled transition systems, similar to Foster's BPEL formalization (see Chapter 2, Section 2.3). Wehrman et al. [103] extended Orc's semantics to allow reasoning about delays, which are introduced explicitly by time-based constructs or implicitly by network delays. This means that we can tell the difference between *halted* and *pending* site calls: a call is in a halted state if the corresponding site indicates that it will never respond to the call; and a call is pending if the site has neither returned a value, nor indicated that the call has been halted. These two states can be distinguished using *otherwise*, one of the available concurrency primitives.

These characteristics make Orc the ideal process specification language for our framework. Unfortunately, Orc is missing two fundamental concepts required by our framework. The first is the ability to compensate executed tasks. As discussed in Section 7.2.2, BPEL allows arbitrary, user-defined compensation, which we limit to individual activity compensation. This limitation allows us to easily formalize and reason about compensation, and the recovery plans we generate can attempt to reverse parts of the error trace. In an unpublished proposal, Coons [22] proposes transactional Orc, which would allow site-specific rollbacks. However, this proposal has not yet been officially adopted and the current version of Orc does not natively provide any form of compensation specification.

The second is the ability intercept events, as well as execute plans. Orc is implemented in Java, so we may be able to accomplish this using an existing Java Virtual Machine monitoring framework, like the one proposed by Orlando and Russo [79]. This remains as future work.

4.6 Related Work

In this section, we first survey the research on runtime monitoring in the context of web services, classified according to the axes defined in Chapter 1, Section 1.2.1. We then present a summary of the error recovery work presented in Chapter 1, Section 1.2.2, focusing on the work by Carzaniga et al. [17], which is the most similar to ours in terms of automatically generated recovery plans.

4.6.1 Runtime Monitoring

Initial work in the area focused on offline runtime monitoring frameworks for web service applications (e.g., [65, 66, 97]); however, recent work has overwhelmingly focused on the development of online monitoring frameworks that, like in our method, monitor the system as it runs (e.g., [9, 10, 62, 81, 63, 42, 43]).

In the offline monitoring framework proposed by Mahbub and Spanoudakis [65, 66], expected choreographies, as well as assertions about individual service invocations, are expressed using Event Calculus (EC) [83]. This specification is then formalized as a set of integrity constraints over a temporal deductive database. System events are collected at runtime and stored in this database – a violation of the integrity constraints indicates a violation of the EC specification. Since event timestamps are stored, this framework can reason about timing and lost messages. van der Aalst et al. [97] built an offline monitoring framework using DecSerFlow, a graphical language for expressing properties, similar to our patterns, but not allowing nesting. Properties specified in this language are turned into LTL properties, and a specialized model checker is used to statically check whether stored event logs satisfy these properties.

The online frameworks described in [9, 10, 62, 63] are restricted to local properties. Li et al. [62] specify properties using Interaction Constraints (IC) [61] – another pattern-based language like ours, but without pattern nesting. In this language, new events must be introduced in order to reason about sequences of events. Just as in our framework, these properties are turned into monitoring automata that are updated at runtime. The frameworks proposed by Baresi et al. [9, 10] and Lohmann et al. [63] check service pre- and postconditions associated with external service invocations. These two frameworks are more invasive than ours, as properties (specified using simple predicate logic) are checked by inserting assertions at the relevant program locations.

The approach introduced by Pistore et al. [81] is the closest to our monitoring framework with respect to the type of properties that can be checked. Global properties are specified in LTL, which is somewhat more expressive than our input properties (although may prove more difficult to use [27]). Monitoring is accomplished through ActiveBPEL [1], an open source, aspect-oriented BPEL engine – monitors are expressed as aspects and dynamically woven into the monitored process.

Hallé and Villemaire in [42, 43], suggest a monitoring framework where data-aware properties are written in LTL enriched with first-order quantifications (LTL-FO+). Generating runtime monitoring automata for such an expressive language is significantly more complex than in our framework, and monitors must be generated on-the-fly because of exponential blowup due to data representation.

4.6.2 Recovery and Self-Healing

The majority of the error recovery frameworks surveyed in Chapter 1, Section 1.2.2 focus on statically defined recovery plans, i.e., the developer specifies possible recovery plans in response to predetermined error conditions, before the application starts running.

The allowed recovery actions are: retrying a service invocation [36, 25, 11], finding an alternative but equivalent service [36] that can replace the faulty service, generating an interface adaptor [93] when an interface mismatch is detected, stopping execution and notifying users that an error has occurred [11], and executing alternative/predefined execution sequences [72, 43].

The framework proposed by Carzaniga et al. [17] is the closest to ours in terms of error recovery, as the authors also attempt to automatically compute recovery plans for runtime errors. This framework exploits redundancy in web applications to find workarounds when errors occur, assuming that the application is given as a finite-state machine (without compensation), with an identified error state as well as the "fallback" state to which the application should return (one per error).

A workaround is a path in the finite-state machine starting at the error state and ending at a fallback state. Before computing all the possible workarounds for an error, the current error transitions are removed from the application model. In some cases, this makes the fallback state unreachable, and additional system events must be inserted to make the model connected again. The approach then exhaustively generates all possible workarounds, prioritizing them solely by length.

Fallback states are manually identified by the developer, whereas our method attempts to compute an approximation of these states using user-specified properties of the system (goal transitions). Our framework also attempts to filter out unusable recovery plans (those that do not include change states) and ranks the remaining ones. In Chapter 6, Section 6.2, we provide a detailed comparison of our approach with the work of Carzaniga et al. [17]. Specifically, we reverse-engineered the two Flickr examples from [17], ran them through our framework and then compared the number and quality of the plans obtained by the two frameworks.

4.7 Summary

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. In the next chapter, we report on our implementation of this framework.

Chapter 5

Tool Support

In this chapter, we describe the implementation of the monitoring and recovery framework presented in Chapter 4. Our tool is called RuMoR, which stands for **RU**ntime **MO**nitoring and **R**ecovery. We have implemented RuMoR on top of the IBM WebSphere product suite, using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts. It takes as input the target BPEL application, enriched with the compensation mechanism allowing us to undo some of the actions of the program, and a set of properties (specified as desired/forbidden behaviours) that need to be maintained by the application as it runs. We discuss the architecture of our tool in Section 5.1, and discuss the implementation of the tool in Section 5.2. When runtime violations are discovered, RuMoR automatically proposes and ranks recovery plans which users can then select for execution.

5.1 Architecture

We show the architecture of our framework in Figure 5.1. In this diagram, rectangles are components of our framework, and ovals are artifacts. We have also grouped the components and artifacts by phase: preprocessing – green, with a ★ symbol; runtime monitoring – yellow, with a ◆ symbol; and recovery – blue, with a ✚ symbol. Artifacts

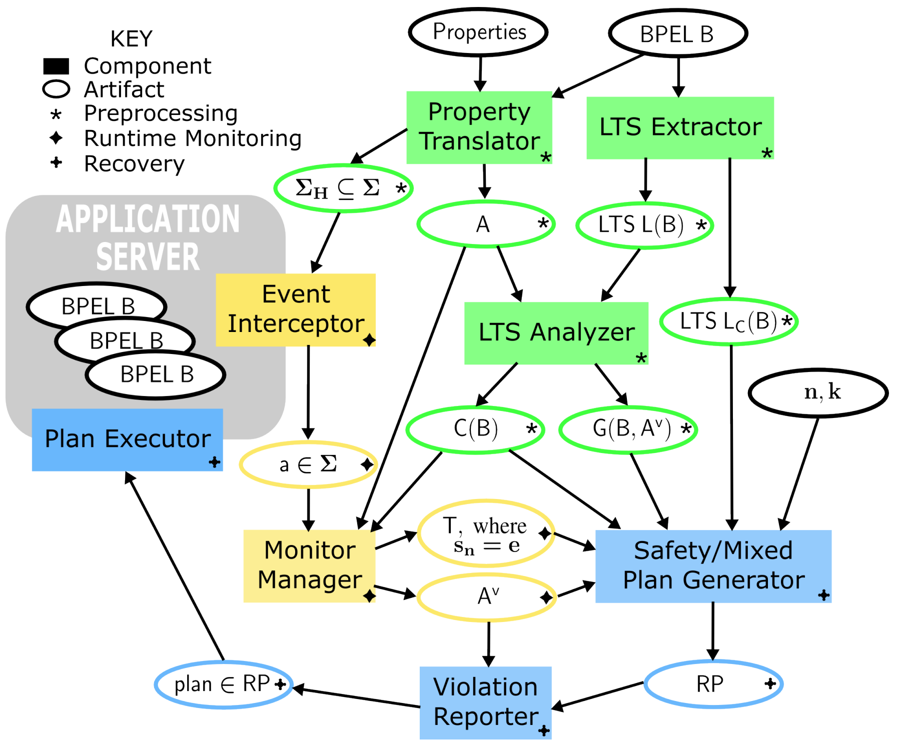


Figure 5.1: Architecture of the framework.

with a thick border are the initial inputs to our framework. The preprocessing and runtime monitoring phases of our framework are the same for both safety and mixed properties, but different components are required for generating plans from the two types of properties.

Developers create properties for their web services using property patterns and system events. During the preprocessing phase, the *Property Translator* (PT) component receives the specified properties and turns them into monitors (as described in Chapter 4, Section 4.1.2). The *LTS Extractor* (LE) component extracts an LTS model from the BPEL program (see Chapter 2, Section 2.3) and creates a second LTS model with compensation (described in Chapter 4, Section 4.1.1). The *LTS Analyzer* (LA) computes goal links and change states using the techniques described in Chapter 4, Section 4.1.3.

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

Chapter 4, Section 4.2 for details). For events that appear in the monitored properties, the EI extracts key information related to the operation invocation: message sender and receiver, whether the invocation is synchronous or asynchronous, whether the message can cause property violation, etc. EI then packs all this information together with the timestamp of when the events were intercepted, and sends them to the message queue associated with MM. The MM pops events off the queue, updating the state of each active monitor until an error has been found (which activates the recovery phase and resets the event queue) 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 Chapter 4, Section 4.3). 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 Chapter 4, Section 4.4).

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 in the next section.

5.2 Implementation

In this section, we discuss the implementation of each RuMoR module, as well as the expected input and output for each module. Selected scripts are listed in Appendix B, while sample input and output files can be found in Appendix C. Both RuMoR and our case studies are available online at `http://www.cs.toronto.edu/fm/RuMoR/`.

5.2.1 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 Chapter 4, Section 4.1.2. These monitors are stored in the Aldebaran [13] 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 [33] to LTSA [64] to translate B into an LTS $\mathsf{L(B)}$ and then export it in the Aldebaran format [13], 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 (see Appendix B, Listing B.1) which uses B and $\mathsf{L(B)}$ to produce $\mathsf{L_C(B)}$ as described in Chapter 4, Section 4.1.1. 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 $\mathsf{L(B)}$ (both in the Aldebaran format). We wrote another Python script (see Appendix B, Listing B.3) that computes the cross-product between the application and each $\mathsf{A_i}$ in $\mathsf{A} \setminus \mathsf{A_S}$ and uses these cross-products to identify goal links, as described in Chapter 4, Section 4.1.3.1. 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 Chapter 4, Section 4.1.3.2).

5.2.2 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. This may cause a noticeable delay during execution, especially if the event does not cause a violation and the application must now execute the remaining events on the queue. This delay has not been a problem in our experiments thus far, but we have not yet studied the impact of this delay in larger, more complex applications, which we leave as future work.

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.

5.2.3 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, $\mathsf{L_C(B)}$, $\mathsf{C(B)}$, and T. We use a new Python script (see Appendix B, Listing B.4) to determine which visited change states are reachable from the error state e on $\mathsf{L_C(B)}$, 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, $\mathsf{L_C(B)}$, $\mathsf{C(B)}$, $\mathsf{G(B, A_v)}$, $\mathsf{A_v}$, and T. We use another Python script (see Appendix B, Listing B.5) to translate $\mathsf{L_C(B)}$ into a planning problem $(\mathsf{L_C(B), e, G_s(B, A^v)})$ (see Chapter 4, Section 4.4). The planning problem is expressed in STRIPS [29] – an input language to the planner Blackbox [54] 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 a new Java script (see Appendix B, Listing B.6) 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 $\mathsf{A_v}$ 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 5.2. Developers must include this snippet in the default error page, so that the computed recovery plans snippet.jsp

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

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

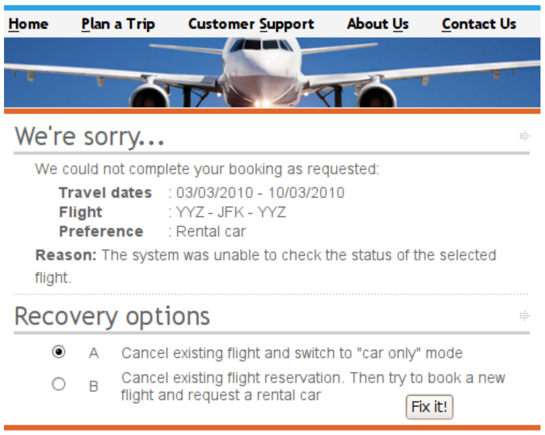
Figure 5.2: `snippet.jsp`, an automatically generated code snippet that contains the computed recovery plans.

can be shown when an error is detected. Figure 5.3a shows the (simplified) source code of such an error reporting page, where the bold 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 5.3b shows a screen shot of `error.jsp` after recovery plans for P_1 have been computed.

Plan Executor receives as input a BPEL plan. In IBM WebSphere Integration Developer v7 [50], developers can also add <collaboration> scopes to their processes. These special scopes, inspired by the work on dynamic workflows [99], can be used to alter the application logic at runtime. Statically, we add a <collaboration> scope to each process before execution, and, at runtime, the BPEL plan chosen by the user is set 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.

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

(a)



We're sorry...

We could not complete your booking as requested:

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

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

Recovery options

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

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

(b)

Figure 5.3: `error.jsp`: (a) relevant code snippet and (b) page displayed on a browser.

5.3 Summary

In this chapter, we described the implementation of RuMoR, an instantiation of the runtime monitoring and recovery framework presented in Chapter 4. RuMoR is built on top of the IBM WebSphere product suite, using a series of publicly available tools and several short (200-300 lines) new Python or Java scripts (available in Appendix B). We applied our tool to several web service applications. We describe these applications in Chapter 6, as well as report on our experience on recovery from property violations.

Chapter 6

Case Studies

In this chapter, we report on three case studies that we have conducted in order to validate our runtime monitoring and recovery framework: the Travel Booking System (`TBS`), Flickr Visibility (`FV`) and Flickr Comments (`FC`). For each of these, we include a description of their BPEL implementation, as well as a set of simple properties and scenarios that violate these properties. The Travel Booking System is an adaptation of the system presented in [38]. The two Flickr case studies originally appeared in [17], where several aspects of the Flickr system are modelled as finite-state models. The Flickr properties and violating scenarios were extracted from the application descriptions in [17].

We monitored each case study using the RuMoR framework. In the `TBS` case study (see Section 6.1), we show how to recover from multiple seeded problems. Experience with this larger, more complex system shows that our approach is both effective and scalable. In Section 6.2, we compare the plans generated using RuMoR to those reported in [17]. In Section 6.3, we discuss the scalability of our framework, as well as the quality of the recovery plans produced. Although our approach generates mostly desirable plans, users can still be overwhelmed by the number of plans produced. Our experience with recovery suggested a couple of techniques for reducing the number of plans generated, which we formalize in Section 6.4, where we also show the results of applying these techniques to the TBS system.

6.1 Travel Booking System

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 6.1 shows the BPEL implementation of this system.

6.1.1 BPEL Model

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 cancelling 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 labelled 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 labelled ③): the user <pick>'s a rental location (pickAirport or pickHotel), and the system tries to place holds

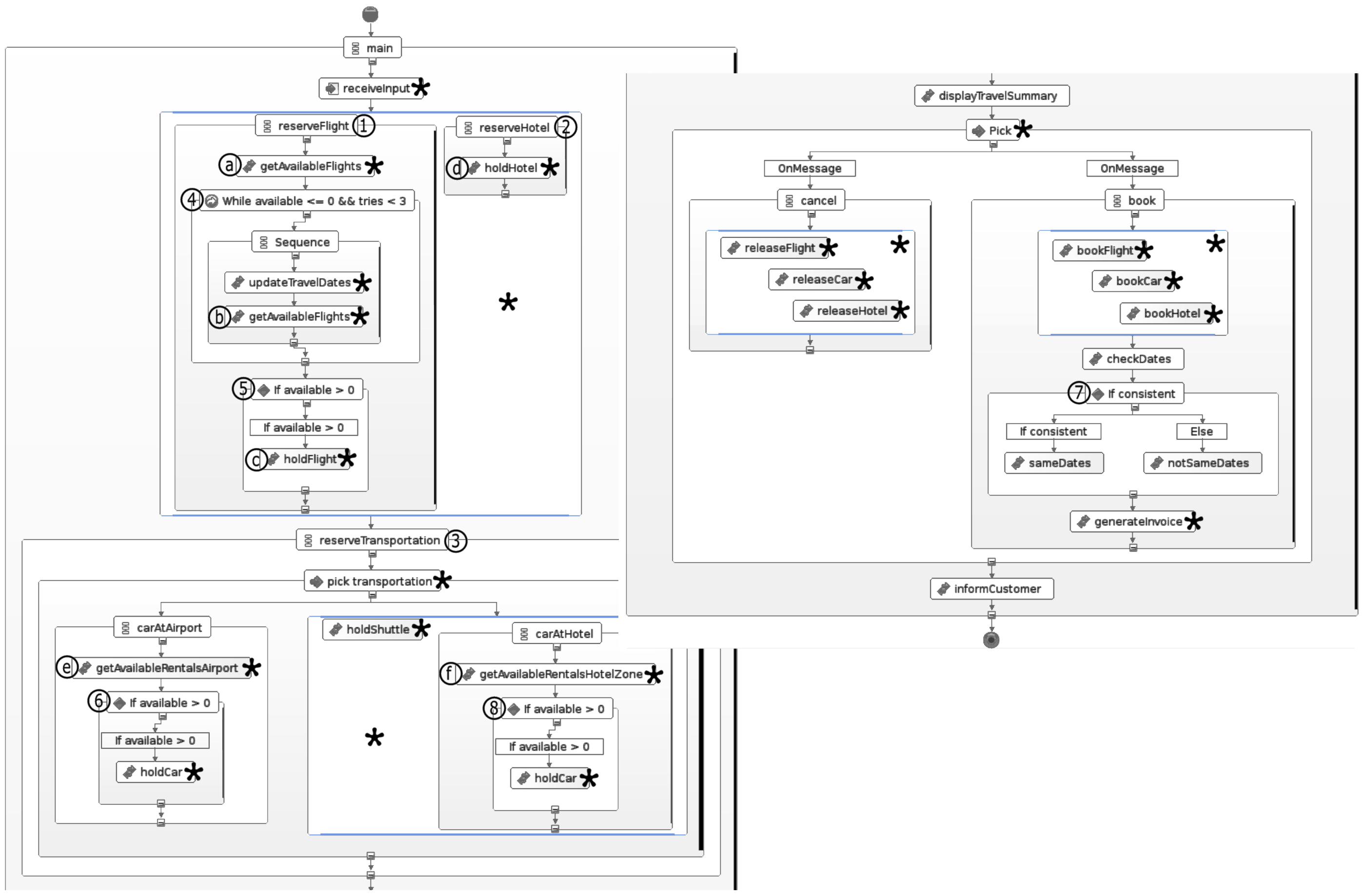


Figure 6.1: BPEL implementation of the Travel Booking System.

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 ($\mathsf{informCustomer}$).

6.1.2 Properties

Some correctness properties of TBS are $P_1^{\mathtt{TBS}}$: "there should not be a mismatch between flight and hotel dates" (expressing a safety property, or a forbidden behaviour), $P_2^{\mathtt{TBS}}$: "a car reservation request will be fulfilled regardless of the location (i.e., airport or hotel) chosen" (expressing a bounded liveness property or a desired behaviour), and $P_3^{\mathtt{TBS}}$: "ground transportation must not be picked before a flight is reserved" (forbidden behaviour). These are expressed using property patterns [26], converted into quantified regular expressions (QRE) [78] and then become monitoring automata.

Property $P_1^{\mathtt{TBS}}$: This property can be expressed using two nested instances of the **Absence** pattern in an **After** scope: **Absence** of a date mismatch event ($\mathsf{notSameDates}$) **After** both a flight and hotel have been booked ($\mathsf{bookFlight}$ and $\mathsf{bookHotel}$, in any order). The resulting QRE properties are:

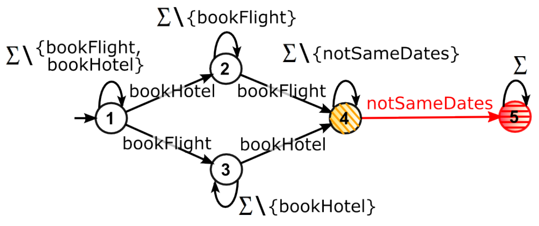
$$P_{1a}^{\mathtt{TBS}} = [-\mathsf{bookFlight}]{*}\cdot\big(\mathsf{bookFlight}\cdot[-\mathsf{bookHotel}]{*}\cdot(\mathsf{bookHotel}\cdot[-\mathsf{notSameDates}]{*})?\big)?$$

$$P_{1b}^{\mathtt{TBS}} = [-\mathsf{bookHotel}]{*}\cdot\big(\mathsf{bookHotel}\cdot[-\mathsf{bookFlight}]{*}\cdot(\mathsf{bookFlight}\cdot[-\mathsf{notSameDates}]{*})?\big)?$$

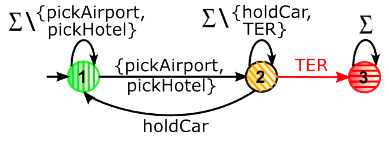
When monitoring the application, we need to make sure that both $P_{1a}^{\mathtt{TBS}}$ and $P_{1b}^{\mathtt{TBS}}$ hold in order to comply with the requirement $P_1^{\mathtt{TBS}}$.

Property $P_2^{\mathtt{TBS}}$: This property can be expressed using an instance of the **Response** pattern in a **Global** scope: **Globally** hold a car ($\mathsf{holdCar}$) in **Response** to a rental location selection ($\mathsf{pickAirport}$ or $\mathsf{pickHotel}$). The resulting QRE property is:

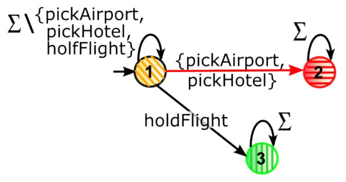
$$P_2^{\mathtt{TBS}} = [-\mathsf{pickAirport}, \mathsf{pickHotel}] * \cdot((\mathsf{pickAirport}|\mathsf{pickHotel}) \cdot [-\mathsf{holdCar}] * \cdot\mathsf{holdCar}\cdot$$
$$[-\mathsf{pickAirport}, \mathsf{pickHotel}]*)*$$



(a)



(b)



(c)

Figure 6.2: Monitors: (a) $\mathtt{A}_1^{\mathtt{TBS}}$, (b) $\mathtt{A}_2^{\mathtt{TBS}}$ and (c) $\mathtt{A}_3^{\mathtt{TBS}}$. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

Property P_3^{TBS}: This property can be expressed using an instance of the **Existence** pattern in a **Before** scope: **Existence** of flight reservation (holdFlight) **Before** the rental location selection (pickAiport or pickHotel). The resulting QRE property is:

$$P_3^{\mathsf{TBS}} = [-\mathsf{pickAirport}, \mathsf{pickHotel}]*|\big([-\mathsf{pickAirport}, \mathsf{pickHotel}, \mathsf{holdFlight}]*\cdot\mathsf{holdFlight}\cdot\Sigma*\big)$$

6.1.3 Preprocessing

The monitoring automata for the three properties are shown in Figure 6.2. A_1^{TBS}, A_2^{TBS} and A_3^{TBS} have 5, 3 and 3 states, and 10, 6 and 5 transitions, respectively. A_1^{TBS}, A_2^{TBS} and A_3^{TBS} have 0, 1 and 1 green states, 1, 1 and 1 yellow states, and finally, 1, 1 and 1 red states.

The monitor A_1^{TBS} in Figure 6.2a enters its error state (5) when the application determines that the hotel and flight booking dates do not match (the hotel and flight can be booked in any order). The monitor A_2^{TBS} in Figure 6.2b represents property P_2^{TBS}: if the application terminates (i.e., sends the TER event) before holdCar appears, the monitor

Scenario	**k**	**Change states**	**Variables**	**Clauses**	**Plans**	**Time (s)**
t_1	6	8	135	254	1	0.01
	8	8	798	10,355	5	0.13
	13	8	1,398	25,023	13	0.27
$t_1^{\texttt{TBS}}$	5	2	–	–	2	0.01
	10	5	–	–	5	0.02
	15	8	–	–	8	0.02
	20	12	–	–	12	0.02
	25	13	–	–	13	0.02
	30	13	–	–	13	0.02
$t_2^{\texttt{TBS}}$	5	4	108	464	0	0.01
	10	7	883	30,524	2	0.14
	15	10	1,456	74,932	8	1.37
	20	10	2,141	135,047	18	4.72
	25	10	3,298	246,210	60	29.16
	30	10	5,288	464,654	68	61.34

Table 6.1: Plan generation data for the TAS and TBS systems. "–" mark cases which are not applicable, such as references to SAT for recovery from forbidden behaviour violations.

moves to the (error) state 3. State 1 is a good state since the monitor enters it once a car has been placed on hold (holdCar). The monitor $A_3^{\texttt{TBS}}$ in Figure 6.2c represents property $P_3^{\texttt{TBS}}$: it enters the good state 3 once a hold is placed on a flight (holdFlight), and enters its error state 2 if the rental location (pickAirport or pickHotel) is picked before a flight is reserved (holdFlights).

The LTS L(TBS) has 52 states and 67 transitions, and $|\Sigma| = 33$, which makes TBS double the size of the TAS example (the LTS is available in ALDEBARAN format in Appendix C). 20 of the BPEL activities (highlighted with a $\star$ symbol in Figure 6.1) yield a total of 35 change states in the LTS. $P_2^{\texttt{TBS}}$ ($P_3^{\texttt{TBS}}$) is a mixed property, with three (two) goal links corresponding to it.

6.1.4 Experience: Recovery from a safety property violation

We generated a recovery plan for the following scenario (called trace $t_1^{\texttt{TBS}}$, of length $k = 21$) which violates property $P_1^{\texttt{TBS}}$: The application first makes a hotel reservation (holdHotel) and then prompts the user for new travel dates (updateTravelDates), since there were no flights available on the current travel dates. The car rental location is the airport (pickAirport). The system displays the itinerary (displayTravelSummary) but the user does not notice the date inconsistency and decides to book it. The TBS makes the bookings (bookFlight, bookHotel and bookCar) and then checks for date consistency (checkDates). Since the dates are not the same (notSameDates), we detect a violation of $P_1^{\texttt{TBS}}$ 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 6.1 summarizes the results. A total of 13 plans were generated, and the longest plan, which reaches the initial state, is of length 21 (and thus the rows corresponding to $k = 25$ and $k = 30$ contain identical information). Since $t_1^{\texttt{TBS}}$ violates a safety property, no SAT instances were generated, and the running time of the plan generation is trivial.

The following plans turn $t_1^{\texttt{TBS}}$ into a successful trace: p_A^1 – cancel the flight reservation and pick a new flight using the original travel dates, and p_B^1 – cancel the hotel reservation and pick a new hotel room for the new travel dates. Our tool generated both of these plans, but ranked them 11th and 12th (out of 13), respectively. They were assigned a low rank due to the interplay between the following two characteristics of our case study: (i) the actual error occurs at the beginning of the scenario (in the flight and hotel reservation <flow>), but the property violation was only detected near the end of the workflow (in the book flow), and (ii) $t_1^{\texttt{TBS}}$ 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 allow us to catch an error as soon as it occurs. We recognize, of course, that this can be difficult to do. The second stems from the fact that not all service calls marked as non-idempotent are relevant to $P_1^{\texttt{TBS}}$ or its violation. In Section 6.4.1, we present a method for identifying those non-idempotent service calls that are relevant to the violation, i.e., their execution may affect the control flow of the current execution. By reducing the number of change states considered, fewer recovery plans will be generated.

6.1.5 Experience: Recovery from a bounded liveness property violation

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

We are again using $n =$ MAX_INT and varying k between 5 and 30, in increments of 5, summarizing the results in Table 6.1. The first thing to note is that our approach generated a relatively large number of plans (over 60) as k approached 30. While in general the further we move away from a goal link, the more alternative paths lead back to it, this was especially true for TBS which had a number of <flow> activities. The second thing to note is that our analysis remained tractable even as the length of the plan and the number of plans generated grew (around 1 min for the most expensive

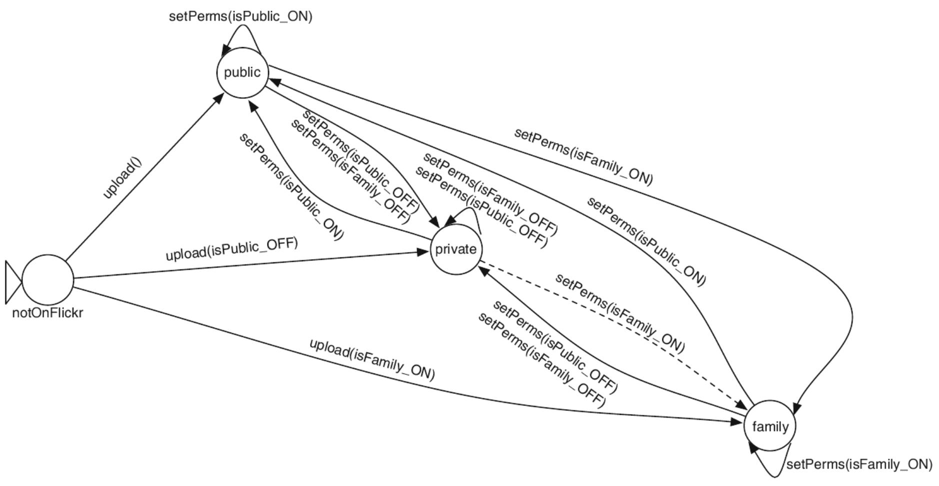


Figure 6.3: Flickr Visibility: behavioural model from [17].

configuration).

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

The rest of the plans we generated compensate various parts of $t_2^{\mathtt{TBS}}$, and then try to reach one of the three goal links. While these longer plans include more compensations and are ranked lower than p_A^2 and p_B^2, we still feel that it may be difficult for the user to have to sift through all of them. As in the case of safety property violations, we can reduce the number of plans generated by picking relevant change states. Furthermore, some of the computed recovery plans, when executed, lead to violations of safety properties, and thus should not be offered to the user. In Section 6.4.2, we present a method for identifying such recovery plans that always lead to violations of safety properties.

6.2 Flickr examples

In the case of the Travel Booking System, we were comparing the effectiveness of the generated plans to our expectations. The goal of this section is to compare the effectiveness of our framework to that of the most similar related work – the approach proposed by Carzaniga et al. [17]. To do so, we took existing examples and ran them on our framework. In Sections 6.2.2 and 6.2.1 we explain how we adapted the two Flickr examples from [17], and in Section 6.2.3, we report our results and compare the two methodologies. Although these case studies are originally from Web 2.0 applications, the work presented in [17] does not not take these Web 2.0 characteristics into account, and we believe that this is a valid comparison.

6.2.1 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".

6.2.1.1 BPEL Model

We created the Flickr visibility system (`FV`) by reverse-engineering the behavioural model in Figure 6.3 (given in [17]) and expressing it in BPEL (see Figure 6.4). The behavioural 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 (public, private or family – we call these three states "visibility states" in the rest of this section). The BPEL model `FV`, consists of 20 activities, of which 6 with explicit compensations.

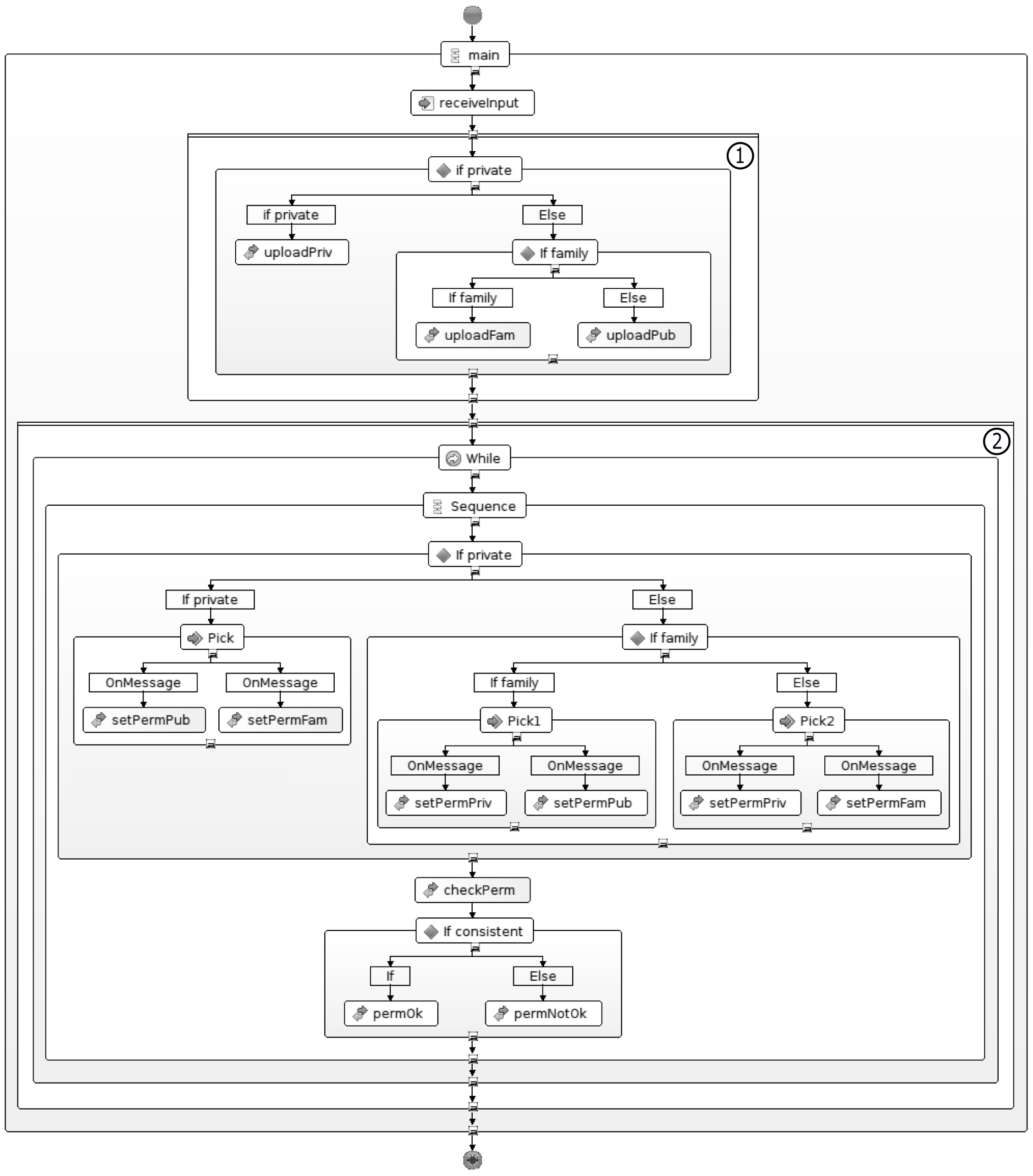


Figure 6.4: BPEL FV.

In Figure 6.4, the transitions from the initial state are modelled in the <scope> called upload (labelled 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 modelled using case statements in the <scope> called changePerm (labelled 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.

In order to check for the described vulnerability, we added a new activity (checkPerm) to this model: after invoking setPerm to change the photo's permission, we call checkPerm to check whether the expected and actual permission settings of the photo are the same (sending a permOk event if they are the same, and a permNotOk event if they are not).

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 – reversing the permission setting, e.g., setPerm(isPublic_ON) is compensated by setPerm(isPublic_OFF). The upload and setPerm service calls are non-idempotent.

6.2.1.2 Properties

We 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 in a **Global** scope:

$P_1^{\mathtt{FV}}$**:** $[-\mathsf{setPermPub}] * \cdot(\mathsf{setPermPub} \cdot [-\mathsf{permOk}] * \cdot\mathsf{permOk} \cdot [-\mathsf{setPermPub}]*)*$

$P_2^{\mathtt{FV}}$**:** $[-\mathsf{setPermFam}] * \cdot(\mathsf{setPermFam} \cdot [-\mathsf{permOk}] * \cdot\mathsf{permOk} \cdot [-\mathsf{setPermFam}]*)*$

$P_3^{\mathtt{FV}}$**:** $[-\mathsf{setPermPriv}] * \cdot(\mathsf{setPermPriv} \cdot [-\mathsf{permOk}] * \cdot\mathsf{permOk} \cdot [-\mathsf{setPermPriv}]*)*$

Property $P_2^{\mathtt{FV}}$ will "catch" the identified vulnerability in the case where a photo is initially loaded as *private*.

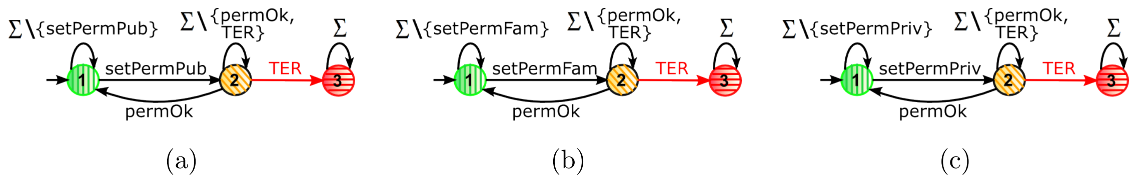


Figure 6.5: Monitors: (a) $\mathtt{A}_1^{\mathtt{FV}}$, (b) $\mathtt{A}_2^{\mathtt{FV}}$ and (c) $\mathtt{A}_3^{\mathtt{FV}}$. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

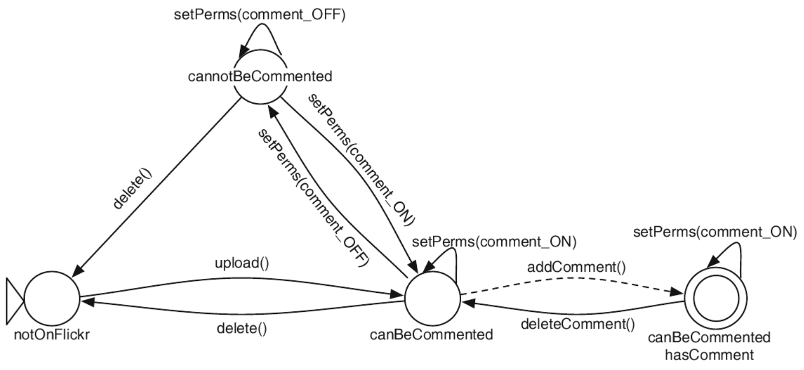


Figure 6.6: Flickr Comments: behavioural model from [17].

6.2.1.3 Preprocessing

The monitoring automata for the three properties are shown in Figure 6.5. The three monitors are similar instances of same pattern, so they all have three states (one green, one yellow and one red) and three transitions each.

Converted to LTS, the resulting model has 28 states and 37 transitions, and is available in ALDEBARAN format in Appendix C. $\mathtt{L}(\mathtt{FV})$ is larger than the original behavioural model since the LTS includes BPEL-induced actions such as entering scopes, and we used if statements to model operation parameters.

6.2.2 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 *family* photos. The vulnerability identified in [17] is "after uploading a photo as *public*, no comments could be added".

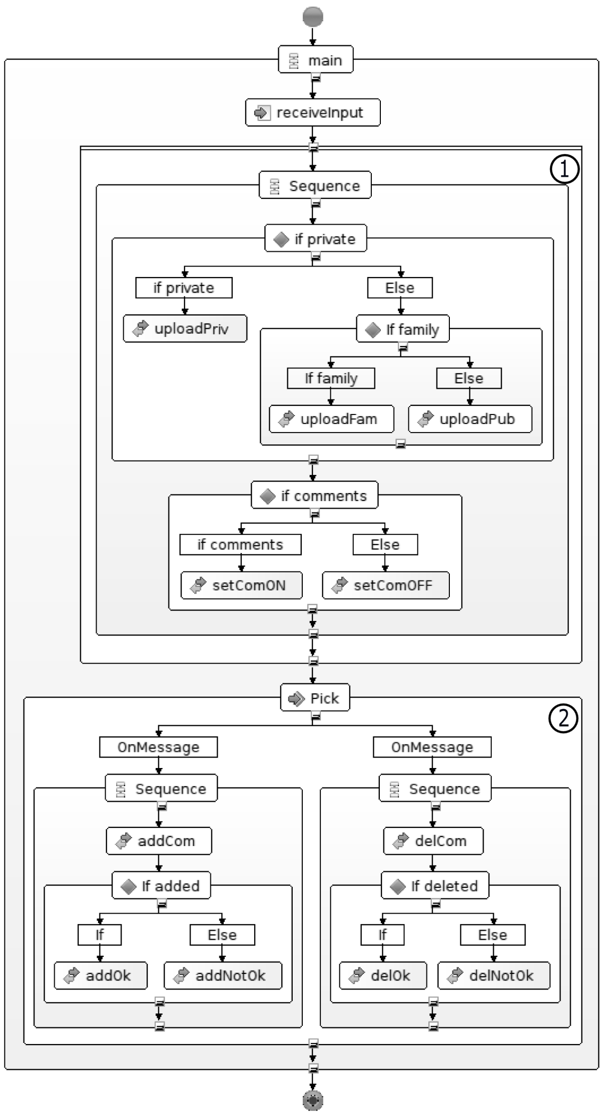


Figure 6.7: BPEL `FC`.

6.2.2.1 BPEL Model

Using the same process as for `FV`, we created the BPEL model `FC` (shown in Figure 6.7), consisting of 18 activities (5 with compensations). This Flickr functionality subset is modelled in BPEL as a <sequence> of two <scopes>: 1) upload the photo and set its initial visibility (using setPermPub, setPermFam or setPermPriv) and set the comment permissions (using setComON or setComOFF), and 2) allow the user to pick between add-ing or delete-ing a comment (using addCom, delCom, respectively), and return a success

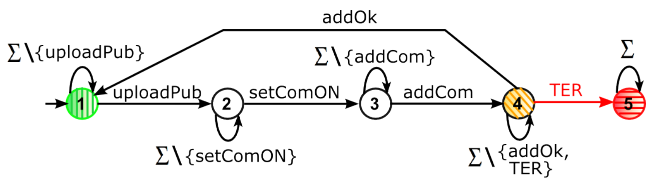


Figure 6.8: Monitor $\mathtt{A}_1^{\mathtt{FC}}$. Red states are shaded horizontally, green states are shaded vertically, and yellow states are shaded diagonally.

message (addOK, delOK, respectively).

We also defined compensation for FC. Since all the transitions to the notOnFlickr state in Figure 6.6 are on the delete event, we defined this event as the compensation for upload. Similarly, delCom compensates addCom, and setComON compensates setComOFF (and vice versa). The upload and addCom service calls are non-idempotent.

6.2.2.2 Properties

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 an instance of the **Response Chain** pattern.

$P_1^{\mathtt{FC}}$**:** $([-\mathsf{uploadPub}]*\cdot\mathsf{uploadPub}[-\mathsf{setComON}]*\cdot\mathsf{setComON}\cdot[-\mathsf{addCom}]*\cdot\mathsf{addCom};[-\mathsf{addOk}]*\cdot\mathsf{addOk})*\cdot[-\mathsf{uploadPub}]*\cdot(\mathsf{uploadPub}\cdot[-\mathsf{setComON}]*\cdot(\mathsf{setComON}\cdot[-\mathsf{addCom}])?)?$

6.2.2.3 Preprocessing

The monitoring automata for $P_1^{\mathtt{FC}}$ is shown in Figure 6.8. This monitor has five states (two white, one green, one yellow and one red) and seven transitions. The resulting LTS model has 18 states and 22 transitions, and is available in ALDEBARAN format in Appendix C.

	Our approach					**[17]**	
App.	**k**	**vars**	**clauses**	**plans**	**time (s)**	**length**	**plans**
`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 6.2: Plan generation data for the Flickr examples.

6.2.3 Comparison with a Related Approach

The number of recovery plans generated for failed traces of `FV` and `FC` is shown in Table 6.2. For example, for the plan length up to 26, we have generated 8 plans for `FV`. The longest plan was of length 42. We looked 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 [17]. 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 looked at the list of workaround sequences computed by [17] (see Table 6.2). The work in [17] modelled the Flickr behaviour 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 6.2 shows that we generate significantly fewer plans of the corresponding length. We also generate every plan marked as desired in [17].

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

6.3 Summary of Evaluation

In this section, we discuss the scalability of our framework, as well as the quality of the recovery plans produced.

6.3.1 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 `TAS`, `TBS`, `FV`, and `FC`, listing them in Tables 6.1 and 6.2. 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.

A thorough experimental analysis would involve systematically modifying a control application, and registering how changes to it affect the SAT instance generated for liveness plans. For example, adding more <flow> activities adds more transitions to the application LTS, which in turn adds more clauses to the SAT encoding of the application. We expect similar results when adding more <pick> activities, with varying amounts of <onMessage> children. We are also interested in studying how the length and location of the error trace, as well as the number of change states and goal transitions affect the size of the SAT instance, as these artifacts determine how much of the planning graph is encoded. This more thorough experiment would give us a better idea of how the structure of the application and the error trace affect the scalability of our approach, and we plan to conduct such an experiment in the near future.

6.3.2 Plan Quality

We conducted a preliminary study involving potential users of our framework to find out whether RuMoR can compute plans that correspond to the recovery actions that a typical user of the application might try once an error is detected. We also want to determine how well our framework ranks plans.

We only included the `TAS` and `TBS` systems in this study since both of these systems have a clear design and implementation. In contrast, the BPEL models of the Flickr examples attempt to mimic the behavioural models presented in [17], which makes their implementation unintuitive and hard to follow. Our methodology (per application) is described below. Each application has two scenarios: a safety and a mixed property violation.

1. We started by presenting the application, its BPEL model and properties to the study participant.

2. We explained the concept of compensation, as well as recovery plan. We indicated which activities in the BPEL model have compensation, as well as their cost.

3. We presented each application's safety property violation scenario. We asked the user to produce, by hand, at least three recovery plans for the scenario, being as

specific as possible, and to rank these plans as well, in the order in which they would try to execute them.

4. We asked the users to do the same for the mixed property scenario.

Currently, our study just had two participants, but we plan to conduct more interviews in the short-term. Since there is not enough data to make any conclusive claims, in the rest of this section we discuss the most interesting observations we have made thus far. The raw data we collected is available in Appendix C.2.

Our participants produced at most three plans per scenario. In order to understand why our participants reported so few plans, we compared the user-created and RuMoR plans, trying to find semantically equivalent plans. What we found is that a single user-created plan could be matched to multiple RuMoR plans. For example, one user reported the following plan for t_1: "cancel the flight, book a car, and then book another flight" (see Appendix C.2, Section C.2.1, Subject #2, plan 2). Since the user does not care whether the flight is cheap or expensive (since they booked a car), this user-reported plan is equivalent to plans p_1 and p_2 in Figure 4.9a. This experience gives us an idea for improving our framework – we do not want to present plans that differ in the occurrence of side-effect-free actions, nor those that differ in the order in which activities are executed, since both types of plans will likely lead the system to the same goal state and are thus indistinguishable in their final result.

Another observation is that users were reticent to include compensation in their plans. This anecdotal evidence backs up our intuition that plans with more compensation should be ranked lower. Finally, we do not have enough data yet to do a precision/recall analysis to determine how accurately our plans were ranked, but we can report that all user-created plans were generated by RuMoR.

Overall, these are promising, but preliminary results. We cannot yet conduct a precision/recall analysis, but in the future, we plan to conduct more extensive user studies,

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

Table 6.3: Predicates that appear on trace $t_1^{\mathtt{TBS}}$, and the non-idempotent service invocations that affect their values.

increasing the complexity of the studied applications, as well as increasing the amount of study participants.

6.4 Optimization: Reducing the Number of Generated Plans

As discussed in Chapter 4, our tool attempts to produce a set of recovery plans for each detected violation. However, in some cases this set includes unusable plans. In this section, we look at techniques for filtering out two types of unusable plans: those that require going through unnecessary change states, where re-executing the partner call cannot affect the (negative) outcome of the trace (see Section 6.4.1), and those that fix a liveness property at the expense of violating some safety properties (see Section 6.4.2).

6.4.1 Relevant Change States

Change states (see Chapter 4, Definition 4.5) are application states from which *flow-changing* actions can be executed: states that model <flow> or <pick> activities, and states from which non-idempotent service calls can be made.

Let us reexamine the trace $t_1^{\mathtt{TBS}}$ from Section 6.1. This trace visited 13 change states, of which 11 correspond to non-idempotent service calls. The two flow activities executed on the trace identify two change states that coincide with two states already identified using non-idempotent service calls (holdHotel and bookCar). The remaining two change states correspond to the two <pick> activities on the trace (a choice between rental locations, and a choice between booking/cancelling the itinerary).

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

Definition 6.1 (Relevant Change State)**.** *A change state is relevant if and only if:*

- *the state is identified as a change state by a <flow> or <pick> activity, or*
- *the state is identified as a change state by a non-idempotent service call, and a variable that appears in a control activity is data dependent on the outcome of this service call.*

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

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

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

and

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

The set of variable definitions that occur on a trace is the union of the definitions that occur on the individual transitions of the trace: for a trace $\mathsf{T} = s_0a_0s_1a_1s_2\ldots a_{n-1}s_n$, $\text{Def}(\mathsf{T}) = \bigcup_i \text{Def}(s_i \xrightarrow{a_i} s_{i+1})$. Now we can define *direct data dependency*: a transition v is directly data dependent on another transition u if and only if v uses a variable defined by u, and there is a path from u to v where this variable is not redefined.

Definition 6.2 (Directly Data Dependent). *A transition* $(q \xrightarrow{b} q')$ *is* directly data dependent *on a transition* $(p \xrightarrow{a} p')$ *if and only if there is a trace* $\mathsf{T} = s_0a_0s_1a_1s_2\ldots a_{n-1}s_n$ *such that* $p' = s_0$, $q = s_n$ *and* $\big(\mathit{Def}(p \xrightarrow{a} p') \bigcap \mathit{Use}(q \xrightarrow{b} q')\big) - \mathit{Def}(\mathsf{T}) \neq \emptyset$.

For example, in Figure 6.1, the <if> activity labelled ⑤ and the holdFlight service call labelled ⓒ are both directly data dependent on the getAvailableFlights service calls at ⓐ and ⓑ.

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

Now we can carry out the data dependency analysis on trace $t_1^{\mathtt{TBS}}$. This trace executed four control activities: 1) the <while> labelled ④, 2) the <if> labelled ⑤, 3) the <if> labelled ⑥, and 4) the <if> labelled ⑦. Table 6.3 lists the corresponding predicates, as well as the non-idempotent service calls that can affect the values of these predicates. For example, the <while> condition is $\mathsf{availableFlights} <= 0 \&\& \mathsf{tries} < 3$. This use of the availableFlights variable is directly data dependent on both appearances of the non-idempotent getAvailableFlights service. On the other hand, the tries variable is not data dependant on any non-idempotent service calls, since it is a simple counter updated by an <assign> statement inside the <while> activity.

The data dependency analysis for predicates 2 and 3 is similar to that of predicate 1. Variable availableFlights also appears in predicate 2, so the non-idempotent service calls marked as relevant are the same as predicate 1. Predicate 3 tests the value of availableCars, which is directly data dependent on the non-idempotent service call getAvailableRentalsAirport. In the case of predicate 4, the variable consistent is directly data dependent on the idempotent service checkDates. The checkDates service call is directly data dependent on the non-idempotent service calls holdHotel and holdFlight, as these services modify reservationData, the input variable of the checkDates service. Thus, predicate 4 is data dependent on holdHotel and holdFlight. These are summarized in Table 6.3.

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

We also carried out the same analysis on trace t_2^{TBS}: six of the original 10 change states are marked as relevant. Since trace $t_2^{\mathtt{TBS}}$ visits the same <pick> and <flow> activities as $t_1^{\mathtt{TBS}}$, four of the relevant change states are those identified by these activities (two by the two <pick> activities, and two by the two <flow> activities). Trace $t_2^{\mathtt{TBS}}$ only visited two control locations: the <if> labelled ⑥, and the <if> labelled ⑧. The predicates for these activities are $\mathsf{availableFlights} > 0$ and $\mathsf{availableCars} > 0$. The availableFlights variable is directly data dependent on the only invocation of the non-idempotent service getAvailableFlights on this trace (labelled ⓐ), and the availableCars variable is directly data dependent on the non-idempotent invocation of the getAvailableRentalsHotel service (labelled ⓕ). Thus, the remaining two relevant change states correspond to non-idempotent service calls.

6.4.2 Avoiding Forbidden Behaviours

Our second method aims to remove those plans that result in the system performing behaviour which is explicitly forbidden. That is, we use safety properties to help filter recovery plans for liveness properties. As described in Chapter 4, Section 4.4, given a trace T that violates a mixed property, we compute a plan P which first "undoes" the trace until a change state and then computes an alternative path to a certain goal. P is *unsuitable* if the path from the initial state going through this change state and continuing via the computed alternative path towards the goal (shown using a thick line in Figure 6.9 and denoted $\mathtt{T_P}$) is forbidden. That is, there exists a safety monitor $\mathtt{A}_i$ which enters an error state when executed on $\mathtt{T_P}$.

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

The path from the initial state to the change state used in P can be very long, and thus we feel that simulating each monitor on the entire trace $\mathtt{T_P}$ is very inefficient. We

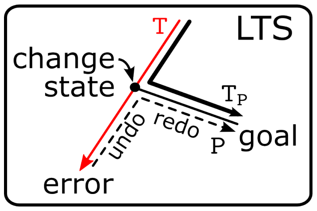


Figure 6.9: Schematic view of recovery plan $\mathtt{P}$ (replicated from Chapter 1), showing $\mathtt{T_P}$ (thick line), the path from the initial state that corresponds to $\mathtt{P}$.

also cannot execute monitors backwards from the error state of $\mathtt{T}$ along the "undo" part of $\mathtt{P}$: while our monitors are deterministic, their inverse transition relations do not have to be deterministic, making the execution in reverse problematic.

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

As an example, consider the $\mathtt{TBS}$ system and trace $t_2^{\mathtt{TBS}}$, described in Section 6.1, violating the property $P_2^{\mathtt{TBS}}$. Our approach produces over 60 plans to recover from this violation, for plan lengths $k \geq 25$ (see Table 6.1). Consider the plan that goes back all the way until encountering the change state associated with the call to getAvailableFlight, cancelling the booked flights on the way. Afterwards, this plan attempts to rebook a flight, but fails to do so. It continues executing, and tries to pick a car at the airport instead. However, this plan violates property $P_3^{\mathtt{TBS}}$ (i.e., monitor $\mathtt{A}_3^{\mathtt{TBS}}$ would enter its error state upon seeing an action pickAirport). Thus, we automatically filter this plan out and do not present it to the user.

Scenario	**k**	**Baseline (from Table 6.1)**		**Relevant Change States**		**Avoiding Forbidden Behaviours**	**Both improvements**
		change states	**plans**	**change states**	**plans**	**plans**	**plans**
$t_1^{\texttt{TBS}}$	5	2	2	0	0	–	–
	10	5	5	2	2	–	–
	15	8	8	5	5	–	–
	20	12	12	8	8	–	–
	25	13	13	8	8	–	–
	30	13	13	8	8	–	–
$t_2^{\texttt{TBS}}$	5	4	0	2	0	0	0
	10	7	2	4	2	2	2
	15	10	8	6	5	8	5
	20	10	18	6	15	11	8
	25	10	60	6	41	32	23
	30	10	68	6	41	38	23

Table 6.4: Results of applying both improvements (separately, and then combined) to the TBS case study. "–" marks cases which are not applicable, since the second improvement only applies to bounded liveness properties.

Overall, applying this approach to recovery for trace $t_2^{\texttt{TBS}}$ reduces the number of plans from over 60 to 41. Furthermore, combining it with the computation of the relevant change states, the number of plans is further reduced to 23 (see Table 6.4). While this number is still relatively large, it presents a considerable improvement and enables the user to pick a desired plan more easily. Of course, the usability of the approach is further significantly improved by limiting the maximum length of the plans produced, but such improvements are orthogonal to the ones studied in this thesis.

6.5 Summary

In this chapter, we reported on our experience of applying RuMoR to four case studies. Our framework successfully detected property violations and generated desired recovery plans. Our experience with the Flickr examples suggests that combining simple properties with the compensation mechanism is effective for producing recovery plans. 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. Finally, the two proposed optimizations reduced the number of computed plans, without discarding relevant plans.

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the contributions made in this thesis and outline directions for future research.

7.1 Summary

In this thesis, we have presented techniques to address several challenges facing the dynamic analysis of web service applications. These challenges range from making the process of property specification more amenable to developers, to automating the process of computing and applying recovery plans for runtime errors. We have explored these challenges in the specific context of BPEL.

In Chapter 3, we have presented W-SD, a subset of UML 2.0 SDs that can be used as a language for specifying properties of web service applications. Specifications expressed in W-SD permit the analysis of orchestrations involving multiple partners, from the point of view of the orchestrating service. We demonstrated the expressiveness of this subset by successfully mapping all the Specification Property System patterns into our SD subset.

In Chapter 4, we have presented our framework for runtime monitoring and recovery of web service applications. 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.

In Chapter 5, we described the implementation of RuMoR, an instantiation of our runtime monitoring and recovery framework. RuMoR is built on top of the IBM WebSphere product suite, using a series of publicly available tools and several short new scripts.

In Chapter 6, we reported on our experience of applying RuMoR to three case studies. RuMoR successfully detected property violations and efficiently generated desired recovery plans. We compared the plans generated by RuMoR to those generated by the recovery framework proposed by Carzaniga et al. [17] – in our experience, our property-guided error recovery framework was able to produce better (and fewer) plans than a framework that does not take properties into account. While the web applications we have analyzed have been relatively small, our experience also suggests that SAT instances used in plan generation remain small and simple and scale well as length of the plan grows. Finally, we have presented two optimizations that reduce the number of computed plans, without discarding relevant plans.

7.2 Future Work

The work presented in this thesis suggests various future research directions in the area of dynamic web service analysis. In this section, we outline some of these directions and their connection to this thesis.

7.2.1 Improving Tool Support

The following is a list of short-term improvements we plan to carry out:

1. 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. The latter issue is extremely important, since users will not execute plans they do not understand. With this in mind, we have designed the Violation Reporter component (see Chapter 5) so that violation reporting and plan suggestion are seamlessly integrated into the monitored application. However, we still need to address plan presentation issues. For example, we do not want to present plans that only differ in the order in which certain activities are executed, since executing any of these plans will likely lead the system to the same goal state. We will attempt to use partial-order reduction [95, 80] to compute representative plans in this situation. We also plan to study how to reorder the events in the executed trace in order to minimize the amount of compensation required by the computed recovery plans.

2. Throughout Chapter 4, 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. For example, since we over-approximate goal transitions by only identifying the last event that must occur in order to observe a desired behaviour, our framework may suggest plans that do not ensure the completion of the desired behaviour. We can remedy this situation by extending goal transitions to include a larger suffix of the desired behaviour, thus improving the chances that the suggested plans actually execute the desired behaviour.

3. We are also interested in improving our use of SAT solving for better plan generation, e.g., by studying how to encode forbidden behaviours as part of the SAT problem rather than filtering them out after the plan has been generated (see

Chapter 6, Section 6.4.2). One way of doing this is to use the safety properties to statically produce a new model of the application where behaviours that likely lead to property violation have been removed, and then use this model for computing plans. Another issue is the use of incremental SAT solving. We currently use an incremental SAT solver to generate multiple plans for a fixed plan length k. However, we do not reuse the results of this process when computing plans of length $k+1$. Since many plans at $k+1$ may be simple extensions of plans of length k, we expect that this improvement may significantly reduce plan computation times.

4. The work in this thesis concentrates on the efficient computation of recovery plans, and we have only included a preliminary study on plan quality and usability (see Chapter 6, Section 6.3.2). In future, we plan to conduct a more conclusive study about plan quality.

5. We are also currently using a simplistic metric to rank plans; we believe plan ranking can be greatly improved through analysis of user histories. For example, we can use machine learning techniques to classify users according to their histories, thus also ranking plans according to how popular these have been amongst previous users of the application.

6. The current version of RuMoR can only analyze BPEL applications running on the IBM WebSphere Process Server [51]. Recall from Chapter 5 that two RuMoR components interact directly with the BPEL engine: the Event Interceptor and the Plan Executor. Since the current implementation of these components includes IBM intellectual property, we cannot make them available online. We are in the process of designing stand-alone versions of them. For example, the Event Interceptor component can be implemented as a wrapper web service that records registered events between the application and the BPEL engine. The Plan Executor component is harder to generalize because of the static nature of BPEL

applications. We currently rely on dynamic workflows [99] to execute plans, but we are investigating whether aspect-orientation can be used instead (e.g., like are available in ActiveBPEL [1]).

7.2.2 Reasoning about Data-Aware Properties

Currently, our framework only permits the definition of properties that depend on the occurrence and order of system events. By monitoring the actual *data* exchanged by conversation participants, we could check richer properties that depend on such data. We cannot use the existing automata translations for data-exchange properties directly, because the resulting automata would be too large to be useful for monitoring. Hallé and Villemaire [42, 43] deal with this problem by generating data-aware monitors on-the-fly (see Chapter 4, Section 4.6). This approach is adequate for monitoring, but may significantly affect the scalability of the recovery plan generation process. Recall from Chapter 4 that we use monitors to compute goal transitions, and in Chapter 6, we use monitors to reduce the number of plans presented to the user. Runtimes for these two procedures will significantly increase if monitors need to be constantly regenerated.

The size of the monitors is not the only issue that arises from making our framework data-aware. We currently model compensation as going back to states visited earlier in the run. While this model is simple, clean and enables effective analysis, the BPEL compensation mechanism allows users to execute arbitrary compensation sequences, which may or may not restore the state of the application. For example, if we model the amount of money the user has as part of the state, then booking and then cancelling 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, since the application model is no longer statically available. In order to reason about data and more complex compensation, we need to augment the model of the application with state information collected at runtime. This directly affects the computation of goal transitions and change states, which can no longer be precomputed. We may be able to increase the efficiency of this process by carrying out some static analysis on the application model, e.g., if all the paths out a state definitively lead to an error, then this state cannot be a change state.

7.2.3 Monitoring and Recovery for Smart Web Service Interactions

In this thesis, we have described how to monitor and recover from violations in the traditional web setting, where applications are predefined and are deployed on the server. An emerging paradigm is that of the *smart internet*, which focuses on user-centric applications. Advances in web technologies have made possible web applications that combine the "media-rich power of the traditional desktop with the deployment and content-rich nature of web applications" [3]. New web applications are highly configurable, resulting in different user experiences for different users. For example, Gmail (Google's email service) introduces new features to different users at different times in order to analyze the usability of these features, and users can themselves choose to activate or deactivate certain features. The Smart Internet proposal by Ng et al. [74] attempts to formalize the concept of user-centric applications. In this proposal, users maintain lists of personal goals (defined in [74] as *matters of concern* (MOC)) that persist between individual sessions with various services. These constraints are used to customize existing applications in order to satisfy individual users' requirements.

Our proposed framework can be easily adapted to this paradigm, as we describe below:

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 are 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 cancelling a flight). Unfortunately, we are not aware of existing technology which allow such dynamic, user-centred 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. However, we cannot centrally monitor all these different versions of the application and MOCs and still expect to keep a low monitoring overhead. 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 [52], but we leave the

problem of distributed plan generation and execution for future work.

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 of ensuring quality of user-centric web systems where the level of customization does not allow effective testing.

Bibliography

[1] Active Endpoints. BPEL Open Source Engine. `http://www.activebpel.org`, Accessed August 2010.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[3] Jeremy Allaire. Macromedia Flash MX: A Next-Generation Rich Client. White paper, Macromedia, March 2002. Available online (14 pages).

[4] Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts. In *Proceedings of 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 114–129. Springer, 1999.

[5] Mohamed Ariff Ameedeen and Behzad Bordbar. A Model Driven Approach to Represent Sequence Diagrams as Free Choice Petri Nets. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC'08)*, pages 213–221. IEEE Computer Society, 2008.

[6] 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.

[7] Jorge A. Baier and Sheila A. McIlraith. Planning with First-Order Temporally Extended Goals using Heuristic Search. In *Proceedings of 21st National Conference on Artificial Intelligence (AAAI'06) and the 18th Innovative Applications of Artificial Intelligence Conference (IAAI'06)*. AAAI Press, July 2006.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Daniel Le Berre and Anne Parrain. SAT4J. `http://www.sat4j.org/`, Accessed August 2010.

[13] 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.

[14] 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.

[15] Tevfik Bultan. Modeling Interactions of Web Software. In *Proceedings of the 2nd International Workshop on Automated Specification and Verification of Web Systems (WWV'06)*, pages 45–52, 2006.

[16] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM.

[17] 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.

[18] 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.

[19] 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, Raffaela 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.

[20] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[21] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In Paolo Ciancarini and Herbert Wiklicky, editors, *Proceedings of the 8th International Conference (COORDINATION '06)*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006.

[22] Katherine E. Coons. Transactional Orc. Unpublished proposal, May 2008.

[23] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Journal of Formal Methods in System Design (FMSD)*, 19(1):45–80, 2001.

[24] Declan McCullagh. Amazon.com experiences hours-long outage. `http://news.cnet.com/8301-1023_3-20009241-93.html`, Accessed August 2010.

[25] 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.

[26] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP '98)*, pages 7–15. ACM, March 1998.

[27] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, pages 411–420. ACM, May 1999.

[28] Matthew B. Dwyer, Vicki Carr, and Laura Hines. Model Checking Graphical User Interfaces using Abstractions. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT International Sympo-*

sium on Foundations of Software Engineering (ESEC '97/FSE-5), pages 244–261. Springer-Verlag New York, Inc., 1997.

[29] 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.

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

[31] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David Rosenblum, and Sebastian Uchitel. Model Checking Service Compositions under Resource Constraints. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 225–234. ACM, 2007.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Yuan Gan. Runtime Monitoring of Web Service Conversations. Master's thesis, University of Toronto, Department of Computer Science, March 2007.

[38] Yuan Gan, Marsha Chechik, Shiva Nejati, Jon Bennett, Bill O'Farrell, and Julie Waterhouse. Runtime Monitoring of Web Service Conversations. In *Proceedings of the 2007 conference of the Centre for Advanced Studies on Collaborative Research (CASCON'07)*, pages 42–57, November 2007.

[39] Naghmeh Ghafari, Arie Gurfinkel, Nils 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. IEEE Computer Society, November 2007.

[40] Victor M. Glushkov. The Abstract Theory of Automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[41] Radu Grosu and Scott A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *Proc. of ACSD'05*, pages 6–14, 2005.

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

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

[44] David Harel and Shahar Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. In *Proceedings of ICSE'06 Workshop on Scenarios and State Machines (SCESM'06)*, pages 13–20, 2006.

[45] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming using LSCs and the Play-Engine*. Springer, 2003.

[46] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS: Towards Formal Design with Sequence Diagrams. *Journal of Software and System Modeling*, 4:355–357, 2005.

[47] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[48] George E. Hughes and Max J. Creswell. *An Introduction to Modal Logic*. Methuen, 1968.

[49] IBM. WebSphere Business Integration Software. `http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint`, Accessed August 2010.

[50] IBM. WebSphere Integration Developer. `http://www-306.ibm.com/software/integration/wid/`, Accessed August 2010.

[51] IBM. WebSphere Process Server. `http://www-306.ibm.com/software/integration/wps/`, Accessed August 2010.

[52] Paola Inverardi, Leonardo Mostarda, Massimo Tivoli, and Marco 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. ACM, 2005.

[53] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96). Technical report, ITU-TS, Geneva, 1996.

[54] 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.

[55] 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.

[56] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR '06)*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[57] Stephen C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies, Annals of Mathematical Studies*, 34:3–41, 1956.

[58] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[59] 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.

[60] Marc Lettrari and Jochen Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'01)*, pages 317–328. Springer-Verlag, 2001.

[61] 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.

[62] 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.

[63] Marc Lohmann, Leonardo Mariani, and Reiko Heckel. A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 173–204. Springer, 2007.

[64] Jeff Magee and Jeff Kramer. *Concurrency - State Models and Java Programs*. John Wiley, 1999.

[65] 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. ACM, 2004.

[66] 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.

[67] Drew V. McDermott. Estimated-Regression Planning for Interactions with Web Services. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS '02)*, pages 204–211. AAAI, 2002.

[68] Sheila A. McIlraith and Tran Cao Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR '02)*, pages 482–496. Morgan Kaufmann, 2002.

[69] Robert McNaughton and Hisao Yamada. Regular Expressions and State Graphs for Automata. *Electronic Computers, IEEE Transactions on*, EC-9(1):39–47, March 1960.

[70] Robin Milner. *Communication and Concurrency.* Prentice-Hall, New York, 1989.

[71] Jayadev Misra, William Cook, and David Kitchin. Orc Language. `http://orc.csres.utexas.edu/index.shtml`, Accessed August 2010.

[72] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In *MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53, New York, NY, USA, 2006. ACM.

[73] Srini Narayanan and Sheila A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the 11th International Conference on World Wide Web (WWW '02)*, pages 77–88. ACM, 2002.

[74] Joanna W. Ng, Mark Chignell, and James R. Cordy. The Smart Internet: Transforming the Web for the User. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '09)*, pages 285–296, 2009.

[75] OASIS. OASIS UDDI Specification TC. `http://www.oasis-open.org/committees/uddi-spec`, Accessed August 2010.

[76] 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 August 2010.

[77] Object Management Group (OMG). Unified Modeling Language (UML 2.0). `http://www.omg.org/spec/UML/2.0/`, Accessed August 2010.

[78] 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.

[79] Salvatore Orlando and Stefano Russo. Java Virtual Machine Monitoring for Dependability Benchmarking. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06)*, pages 433–440. IEEE Computer Society, 2006.

[80] Doron Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV '94)*, pages 377–390. Springer-Verlag, 1994.

[81] 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.

[82] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

[83] Murray Shanahan. The Event Calculus Explained. In *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *LNCS*, pages 409–430. Springer, 1999.

[84] 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.

[85] Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. Monitoring and Recovery of Web Service Applications. In Joanna W. Ng, Mark Chignell, and James R. Cordy, editors, *Smart Internet*, Lecture Notes in Computer Science. Springer, 2010. To appear.

[86] Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. Optimizing Computation of Recovery Plans for BPEL Applications. In *Proceedings of 2010 Workshop on Testing, Analysis and Verification of Web Software (TAV-WEB'10)*, 2010. To appear.

[87] Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. RuMoR: Monitoring and Recovery of BPEL Applications. In *Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, 2010. To appear.

[88] Jocelyn Simmonds, Marsha Chechik, Shiva Nejati, Elena Litani, and Bill O'Farrell. Property Patterns for Runtime Monitoring of Web Service Conversations. In *Proceedings of 8th International Workshop on Runtime Verification (RV '08). Selected Papers*, pages 137–157, 2008.

[89] 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*, 2(3):223–244, 2009.

[90] Simon Moser and Michal Chmielewski. BPEL Project home. `http://www.eclipse.org/bpel/index.php`, Accessed August 2010.

[91] Harald Störrle. Assert, Negate and Refinement in UML 2 Interactions. In *Proceedings of Workshop on Critical Systems Development with UML (CSDUML '03)*, pages 79–94, 2003.

[92] The Open Group. The SOA Work Group. `http://www.opengroup.org/soa/`, Accessed August 2010.

[93] Massimo Tivoli, Pascal Fradet, Alain Girault, and Gregor Goessler. Adaptor Synthesis for Real-time Components. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, pages 185–200. Springer-Verlag, 2007.

[94] Paolo Traverso and Marco Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proceedings of the International Semantic Web Conference (ISWC '04)*, pages 380–394, 2004.

[95] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In Grzegorz Rozenberg, editor, *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[96] Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and Eric Verbeek. Conformance checking of service behavior. *ACM Transactions on Internet Technology*, 8(3):1–30, 2008.

[97] Wil M. P. van der Aalst and Maja Pesic. 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.

[98] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003.

[99] 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.

[100] Moshe Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In *Proceedings of 8th Banff Higher Order Workshop*, volume 1043 of *LNCS*, pages 238–266. Springer, August 1996.

[101] W3C. SOAP Specifications. `http://www.w3.org/TR/soap/`, Accessed August 2010.

[102] W3C. Web Services Description Language (WSDL). `http://www.w3.org/TR/wsdl/`, Accessed August 2010.

[103] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. A timed semantics of Orc. *Theoretical Computer Science*, 402(2–3):234–248, Aug 2008.

[104] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Pattern-Based Analysis of BPEL4WS. QUT Technical report FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.

[105] 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.

Appendix A

QRE Property Patterns

1. Absence - event P does not occur:
   - **Globally**: $[-\mathsf{P}]*$
   - **Before** R: $[-\mathsf{R}] * |[-\mathsf{P},\mathsf{R}] * \cdot \mathsf{R} \cdot \Sigma*$
   - **After** Q: $[-\mathsf{Q}] * \cdot (\mathsf{Q} \cdot [-\mathsf{P}]*)?$
   - **Between** Q **and** R: $([-\mathsf{Q}] * \cdot \mathsf{Q} \cdot [-\mathsf{P},\mathsf{R}] * \cdot \mathsf{R}) * \cdot [-\mathsf{Q}] * \cdot (\mathsf{Q} \cdot [-\mathsf{R}]*)?$
   - **After** Q **until** R: $([-\mathsf{Q}] * \cdot \mathsf{Q} \cdot [-\mathsf{P},\mathsf{R}] * \cdot \mathsf{R}) * \cdot [-\mathsf{Q}] * \cdot (\mathsf{Q} \cdot [-\mathsf{P},\mathsf{R}]*)?$
2. Existence - event P occurs:
   - **Globally**: $[-\mathsf{P}] * \cdot \mathsf{P} \cdot \Sigma*$
   - **Before** R: $[-\mathsf{R}] * |[-\mathsf{P},\mathsf{R}] * \cdot \mathsf{P} \cdot \Sigma*$
   - **After** Q: $[-\mathsf{Q}] * \cdot (\mathsf{Q} \cdot [-\mathsf{P}] * \cdot \mathsf{P} \cdot \Sigma*)?$
   - **Between** Q **and** R: $([-\mathsf{Q}] * \cdot \mathsf{Q} \cdot [-\mathsf{P},\mathsf{R}] * \cdot \mathsf{P} \cdot [-\mathsf{R}] * \cdot \mathsf{R}) * \cdot [-\mathsf{Q}] * \cdot (\mathsf{Q} \cdot [-\mathsf{R}]*)?$
   - **After** Q **until** R: $([-\mathsf{Q}] * \cdot \mathsf{Q} \cdot [-\mathsf{P},\mathsf{R}] * \cdot \mathsf{P} \cdot [-\mathsf{R}] * \cdot \mathsf{R}) * \cdot [-\mathsf{Q}] * \cdot (\mathsf{Q} \cdot [-\mathsf{P},\mathsf{R}] * \cdot \mathsf{P} \cdot [-\mathsf{R}]*)?$
3. Bounded Existence - event P occurs at most k times:

- **Globally**: $(([-\mathsf{P}]*\cdot\mathsf{P})?)^k\cdot[-\mathsf{P}]*$
- **Before** R: $[-\mathsf{R}]*|((([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})?)^k\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R}\cdot\Sigma*$
- **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot(([-\mathsf{P}]*\cdot\mathsf{P})?)^k\cdot[-\mathsf{P}]*)?$
- **Between** Q **and** R: $([-\mathsf{Q}]*\cdot\mathsf{Q}\cdot(([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})?)^k\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
- **After** Q **until** R: $([-\mathsf{Q}]*\cdot\mathsf{Q}\cdot(([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})?)^k\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot(([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})?)^k\cdot[-\mathsf{P},\mathsf{R}]*)?$

4. Universality - specifying universality in an event-based formalism requires the identification of complementary events. The positive event, P, is defined such that after an occurrence of P, states have the property. The negative event, N, is defined such that after an ocurrence of N states fail to have the property. A Universality property holds on a scope if a P event is seen prior to the beginning of the scope, with no N event until after the end of the scope. We assume that the initial state has the desired property:

   - **Globally** $[-\mathsf{N}]*$
   - **Before** R: $[-\mathsf{R}]*|[-\mathsf{N},\mathsf{R}]*\cdot\mathsf{R}\cdot\Sigma*$
   - **After** Q: $[-\mathsf{Q}]*|([-\mathsf{Q}]*\cdot\mathsf{P})?\cdot[-\mathsf{N},\mathsf{Q}]*\cdot\mathsf{Q}\cdot[-\mathsf{N}]*$
   - **Between** Q **and** R: $(([-\mathsf{Q}]*\cdot\mathsf{P})?\cdot[-\mathsf{N},\mathsf{Q}]*\cdot\mathsf{Q}\cdot[-\mathsf{N},\mathsf{R}]*\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
   - **After** Q **until** R: $(([-\mathsf{Q}]*\cdot\mathsf{P})?\cdot[-\mathsf{N},\mathsf{Q}]*\cdot\mathsf{Q}\cdot[-\mathsf{N},\mathsf{R}]*\cdot\mathsf{R})*\cdot([-\mathsf{Q}]*|(([-\mathsf{Q}]*\cdot\mathsf{P})?\cdot[-\mathsf{N},\mathsf{Q}]*\cdot\mathsf{Q}\cdot[-\mathsf{N},\mathsf{R}]*)$

5. Precedence - event S precedes event P:

   - **Globally**: $[-\mathsf{P}]*|([-\mathsf{S},\mathsf{P}]*\cdot\mathsf{S}\cdot\Sigma*)$
   - **Before** R: $[-\mathsf{R}]*|([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R}\cdot\Sigma*)|([-\mathsf{S},\mathsf{P},\mathsf{R}]*\cdot\mathsf{S}\cdot\Sigma*)$
   - **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P}]*|([-\mathsf{S},\mathsf{P}]*\cdot\mathsf{S}\cdot\Sigma*)))?$

- **Between** Q **and** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{R}]*|(([-\mathsf{S},\mathsf{P},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{R}]*)\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
- **After** Q **until** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{R}]*|(([-\mathsf{S},\mathsf{P},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{R}]*)\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*|(([-\mathsf{S},\mathsf{P},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{R}]*)))?$

6. Response - event S follows event P:

- **Globally**: $[-\mathsf{P}]*\cdot(\mathsf{P}\cdot[-\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P}]*)*$
- **Before** R: $[-\mathsf{R}]*|[-\mathsf{P},\mathsf{R}]*\cdot(\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R}]*)*\cdot\mathsf{R}\cdot\Sigma*$
- **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P}]*\cdot(\mathsf{P}\cdot[-\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P}]*)*)?$
- **Between** Q **and** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{R}]*\cdot(\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R}]*)*\cdot\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
- **After** Q **until** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{R}]*\cdot(\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R}]*)*\cdot\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{R}]*\cdot(\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R}]*)*)?$

7. Precedence Chain

(a) Two cause events (S,T) precede one effect event (P):

- **Globally**: $[-\mathsf{P}]*|(([-\mathsf{P},\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{T}]*\cdot\mathsf{T}\cdot\Sigma*)$
- **Before** R: $[-\mathsf{R}]*|(([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R}\cdot\Sigma*)|(([-\mathsf{P},\mathsf{R},\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R},\mathsf{T}]*\cdot\mathsf{T}\cdot\Sigma*)$
- **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P}]*|(([-\mathsf{P},\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{T}]*\cdot\mathsf{T}\cdot\Sigma*)))?$
- **Between** Q **and** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*|(([-\mathsf{P},\mathsf{R},\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R},\mathsf{T}]*\cdot\mathsf{T}\cdot[-\mathsf{R}]*))\cdot\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
- **After** Q **until** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*|(([-\mathsf{S},\mathsf{P},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R},\mathsf{T}]*\cdot\mathsf{T}\cdot[-\mathsf{R}]*))\cdot\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*|(([-\mathsf{P},\mathsf{R},\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{P},\mathsf{R},\mathsf{T}]*\cdot\mathsf{T}\cdot[-\mathsf{R}]*)))?$

(b) One cause event (P) precedes two effect events (S,T):

- **Globally**: $[-\mathsf{P},\mathsf{S}]*\cdot((\mathsf{P}\cdot\Sigma*)|(\mathsf{S}\cdot[-\mathsf{T}]*))?$
- **Before** R: $([-\mathsf{R}]*)|(([-\mathsf{P},\mathsf{S},\mathsf{R}]*\cdot((\mathsf{P}\cdot[-\mathsf{R}]*)|(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*))?\cdot\mathsf{R}\cdot\Sigma*)$

- **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{S}]*\cdot((\mathsf{P}\cdot\Sigma*)|(\mathsf{S}\cdot[-\mathsf{T}]*))?))?$
- **Between** Q **and** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{S},\mathsf{R}]*\cdot((\mathsf{P}\cdot[-\mathsf{R}]*)|(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*))?\cdot\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
- **After** Q **until** R: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{P},\mathsf{S},\mathsf{R}]*\cdot((\mathsf{P}\cdot[-\mathsf{R}]*)|(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*))?\cdot\mathsf{R}\cdot[-\mathsf{Q}]*)*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{S},\mathsf{R}]*\cdot((\mathsf{P}\cdot[-\mathsf{R}]*)|(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*))?))?$

8. Response Chain

   (a) One reponse event (P) follows two stimuli events (S,T):

   - **Globally**: $([-\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{T}]*\cdot\mathsf{T}\cdot[-\mathsf{P}]*\cdot\mathsf{P})*\cdot[-\mathsf{S}]*\cdot(\mathsf{S}\cdot[-\mathsf{T}]*)?$
   - **Before** R: $[-\mathsf{R}]*|([-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T}\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})*\cdot[-\mathsf{S},\mathsf{R}]*\cdot(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*)?\cdot\mathsf{R}\cdot\Sigma*$
   - **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{T}]*\cdot\mathsf{T}\cdot[-\mathsf{P}]*\cdot\mathsf{P})*\cdot[-\mathsf{S}]*\cdot(\mathsf{S}\cdot[-\mathsf{T}]*)?)?$
   - **Between** Q **and** R: $([-\mathsf{Q}]*\cdot\mathsf{Q}\cdot([-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T}\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})*\cdot[-\mathsf{S},\mathsf{R}]*\cdot(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*)?\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
   - **After** Q **until** R: $([-\mathsf{Q}]*\cdot\mathsf{Q}\cdot([-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T}\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})*\cdot[-\mathsf{S},\mathsf{R}]*\cdot(\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*)?\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T}\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P})*)?$

   (b) Two reponse events (S,T) follow one stimulus event (P):

   - **Globally**:$([-\mathsf{P}]*\cdot\mathsf{P}\cdot[-\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{T}]*\cdot\mathsf{T})*\cdot[-\mathsf{P}]*$
   - **Before** R: $[-\mathsf{R}]*|([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T})*\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R}\cdot\Sigma*$
   - **After** Q: $[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P}]*\cdot\mathsf{P}\cdot[-\mathsf{S}]*\cdot\mathsf{S}\cdot[-\mathsf{T}]*\cdot\mathsf{T})*\cdot[-\mathsf{P}]*)?$
   - **Between** Q **and** R: $([-\mathsf{Q}]*\cdot\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T})*\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot[-\mathsf{R}]*)?$
   - **After** Q **until** R: $([-\mathsf{Q}]*\cdot\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T})*\cdot[-\mathsf{P},\mathsf{R}]*\cdot\mathsf{R})*\cdot[-\mathsf{Q}]*\cdot(\mathsf{Q}\cdot([-\mathsf{P},\mathsf{R}]*\cdot\mathsf{P}\cdot[-\mathsf{S},\mathsf{R}]*\cdot\mathsf{S}\cdot[-\mathsf{T},\mathsf{R}]*\cdot\mathsf{T})*)?$

Appendix B

Additional Tool Support Details

Listing B.1: Partial listing of add_comp.py

```
def add_comp(app, comp):
        # app = application LTS
        # comp = hash of {action => compensation} pairs

        # hash for storing compensation transitions
        comp_tt = {}

        # for every transition in tt
        for from_state in app.tt.keys():
                for sigma in (app.tt[from_state]).keys():
                        to_state = app.tt[from_state][sigma]

                        # if the transition action has a compensation action,
                        # add a transition to the compensation transition relation
                        if comp.has_key(sigma):
                                if not comp_tt.has_key(to_state):
                                        comp_tt[to_state] = {}
                                if not comp_tt[to_state].has_key(comp[sigma]):
                                        comp_tt[to_state][comp[sigma]] = from_state
        return comp_tt
```

Listing B.2: Partial listing of NFA.py

```
class NFA:
    def __init__(self, states, alphabet, tt, start, accepts):
        self.states = states
        self.start = start
        self.tt = tt
        self.delta = (lambda s, a: self.tt[s][a])
        self.accepts = accepts
        self.alphabet = alphabet
        self.current_state = start
        self.final = []
        self.comp_tt = {}
...
def cross_product(app, mon):
        # this method builds CP, the cross product app X mon
        # app = application LTS
        # mon = monitor LTS

        # auxiliary set of states
        states = []
        for s1 in app.states:
                for s2 in mon.states:
                        states.append((s1,s2))

        # CP start state
        start = (app.start, mon.start)
        # CP alphabet
        alphabet = copy(app.alphabet)
        # CP accepting states
        accepts = []

        reached = {}
        reached[start] = True
        # list of CP states to process
        to_process = [start]

        # list of reachable CP states
        new_states = [start]
        # CP transition relation
        tt = {}

        # check which CP states are reachable from the CP start state, and
        # construct tt during this check
        while len(to_process):
                # remove a state from the processing list
                q = to_process.pop()

                # for each symbol of the alphabet, check if we need to
                # add a transition to the CP
                for c in app.alphabet:
                        if str(q[0]) not in app.tt:
                                continue

                        if c in app.tt[str(q[0])]:
                                app_next = app.tt[str(q[0])][c]
                        else: app_next = None

                        # monitors have 'Sigma' and 'Sigma-{events}'
                        # transitions
                        if c in mon.tt[str(q[1])]:
                                mon_next = mon.tt[str(q[1])][c]
                        elif 'Sigma' in mon.tt[str(q[1])] or \
                                'Sigma-' + c not in mon.tt[str(q[1])] :
                                mon_next = q[1]
                        else:
                                mon_next = None

                        # if both the application and the monitor transition
                        # on a symbol, add transition to CP
                        if app_next is not None and mon_next is not None:
                                to_state = (app_next,mon_next)
                                if q not in tt:
                                        tt[q] = {}
                                        tt[q][c] = to_state
                                else:
                                        if c not in tt[q]:
                                                tt[q][c] = to_state
                                        else:
                                                tt[q][c].append(to_state)
                                if to_state not in reached:
                                        reached[to_state] = True
                                        to_process.append(to_state)
                                        new_states.append(to_state)
        return NFA(states=new_states, start=start, tt=tt, accepts=accepts, \
        alphabet=alphabet)
```

Listing B.3: Partial listing of compute_cp.py: using NFA.cross_product from Listing B.2

```
def get_app_states(cp_states, final_states):
        # final_states is a list of red monitor states or green monitor states

        if len(final_states) == 0: return []

        states = []
        for cp in cp_states:
                if int(cp[1]) in final_states:
                        if int(cp[0]) not in states: states.append(cp[0])

        return states

if __name__ == "__main__":
        ...
        # cp is the cross product of the application and monitor LTSs
        cp = NFA.cross_product(app_nfa, mon_nfa)
        ...
        # mon_nfa.final[0] is the set of green monitor states
        good_app_states = get_app_states(cp.states, mon_nfa.final[0])
        # mon_nfa.final[1] is the set of red monitor states
        bad_app_states = get_app_states(cp.states, mon_nfa.final[1])
        ...
```

Listing B.4: Partial listing of gen_safe_plan.py

```
def compute_plans(app, error_state, error_trace, max_plan, max_len, change):
        # computes saftey recovery plans
        # app = application LTS
        # error_state = application error state
        # error_trace = current error trace, list of (from_state, action, to_state)
        #   tuples, stored from last to first
        # max_plan = maximum number of recovery plans
        # max_len = maximum length of recovery plans
        # change = list of application change states

        # list of computed plans
        plans = []
        # current plan
        plan = []

        for trio in error_trace:
                # exit loop if enough plans have been computed
                if len(plans) == max_plan:
                        break

                # exit loop if current transition does not have compensation
                if not app.comp_tt.has_key(trio[1]):
                        break

                plan.append(trio)
                ctr += 1

                # exit loop if maximum plan length reached
                if ctr == max_len:
                        break

                # if the from_state is a change states
                if trio[0] in change:
                        # store a copy of the current plan
                        plans.append(plan[:])

        return plans
```

Listing B.5: Partial listing of gen_plan_prob.py

```
def write_adl(fa, filename, tt_type):
        # this method generates the ADL encoding of the LTS fa
        # fa = LTS that will encoded
        # filename = file where ADL encoding will be stored
        # tt_type = "nfa" or "dfa"

        file = open(filename, "wb")
        msg = """(define (domain domain_prop1)\n   (:requirements :adl)\n   (:predicates"""

        for state in sorted(fa.states):
                msg += " (s" + str(state) + ")"

        msg += ")\n\n"
        file.write(msg)

        # generates actions of the form
##  (:action displayResult
##      :precondition (or (s1) (s2))
##    :effect (and
##                (when (s1) (and (not (s1)) (s3)))
##                (when (s2) (and (not (s2)) (s4)))
##        )
##  )

        # collect all from-to pairs for each sigma, outgoing/incoming per state
        pairs = {}
        incoming = {}
        outgoing = {}
        if tt_type == "nfa":
                for from_state in fa.tt.keys():
                        for sigma in (fa.tt[from_state]).keys():
                                if sigma not in pairs:
                                        pairs[sigma] = []
                                pairs[sigma].append((int(from_state), \
                                fa.tt[from_state][sigma]))

        elif tt_type == "dfa":
                pass

        # write out formatted action statements
        for sigma in fa.alphabet:
                sigma_no_dots = sigma.replace(".", "_")
                file.write("   (:action " + sigma_no_dots + "\n")
                file.write("      :parameters ()")
                file.write("      :precondition (or")
                only_first = [item[0] for item in pairs[sigma]]
                for item in only_first:
                        file.write(" (s" + str(item) + ")")
                file.write(")\n")
                file.write("      :effect (and\n")
                for item in pairs[sigma]:
                        file.write("         (when (s" + str(item[0]) + ") \
                        (and (not (s" + str(item[0]) + ")) (s" + str(item[1]) + ")))\n")
                file.write("      )\n   )\n")

        file.write(")")
        file.close()

def generate_strips(script_cwd, dom_prefix, prob_prefix):
        # use adl2strips to convert ADL into STRIPS
        # script_cwd = current working directory
        # dom_prefix = filename prefix for domain file
        # prob_prefix = filename prefix for problem file

        # run adl2strips to ground the planning problem
        r = os.system(script_cwd + "/adl2strips-linux-static -p " + script_cwd + "/" + cwd + \
        "/ -o " + domain + " -f " + problem)

        # if adl2strips ran successfully, clean up filenames
        if r == 0:
                shutil.move(script_cwd + "/" + cwd + "/domain.pddl", dom_prefix + \
                "_strips.pddl")
                shutil.move(script_cwd + "/" + cwd + "/facts.pddl", prob_prefix + \
                "_strips.pddl")

def generate_cnf(script_cwd, dom_prefix, prob_prefix,save_vars):
        # use BlackBox to generate CNF encoding of the planning problem
        # script_cwd = current working directory
        # dom_prefix = filename prefix for domain file
        # prob_prefix = filename prefix for problem file
        # save_vars = boolean value that indicates whether action <-> cnf variable
        #   mapping must be stored

        cmd = [script_cwd + "/blackbox", "-printcnf", "-printmap", "-o", dom_prefix + \
        "_strips.pddl", "-f", prob_prefix + "_strips.pddl", "-t", str(t)]

        # run blackbox, opening a pipe in order to process blackbox output
        pipe = subprocess.Popen(cmd, stdout=PIPE).stdout

        flag = 0
        action_vars = ""
        action_var_map = ""

        # open file where CNF encoding will be stored
        file = open("sat__" + domain[0:len(domain)-5] + "__" + \
        problem[0:len(problem)-5] + "__t_" + str(t) + ".cnf", "wb")

        for line in pipe:
                # parse blackbox output

                # these lines contain the action <-> cnf variable mapping
                if flag == 0 and line.startswith("a "):
                        file.write(line.replace("a","c",1))
                        if line.find("noop") == -1 and line.find("c_"):
                                action_vars += line.split(" ")[1] + " "
                                action_var_map += line.split(" ")[1] + " " \
                               + line.split(" ")[2]

                # this line marks the beginning of the CNF encoding
                if line.startswith("Begin cnf") :
                        flag = 1
                        continue
                if flag == 1:
                        # this line marks the end of the CNF encoding
                        if line.startswith("End cnf") : flag = 2
                        else: file.write(line)

        file.close()

        if (save_vars == 1):
                file = open("av__" + domain[0:len(domain)-5] + "__" + \
                problem[0:len(problem)-5] + "__t_" + str(t), "wb")
                file.write(action_vars + "\n")
                file.write(action_var_map)
                file.close()
```

Listing B.6: Partial listing of GeneratePlans.java: using SAT4J to produce multiple plans

```
public static void main(String[] args) throws IOException {

        // Creating SAT4J instance
        ISolver solver = SolverFactory.newDefault();
        ModelIterator mi = new ModelIterator(solver);
        solver.setTimeout(3600); // 1 hour timeout
        Reader reader = new InstanceReader(mi);
        ...

        // Generated plans are stored in this array list
        ArrayList<String> plans = new ArrayList<String>();
        boolean unsat = true;

        // Reading in SAT instance produced by gen_plan_prob.py
        IProblem problem = reader.parseInstance(args[0]);

        // While the SAT instance is satisfiable ...
        while (problem.isSatisfiable()) {
                unsat = false;

                // ... get a new satisfying assignment
                int[] model = problem.model();

                // ... and use action mapping to convert it into a plan
                VecInt newClause = new VecInt();
                String newPlan = "";
                for (int i = 0; i < actionVars.length; i++) {
                        if (actionVars[i] == limit)
                                break;
                        if (problem.model(actionVars[i])) {
                                newClause.push(-1 * actionVars[i]);
                                newPlan += " " + var_mapping.get(actionVars[i]);
                        }
                }

                // add new plan to list of plans, and new constraint to SAT instance
                plans.add(newPlan);
                solver.addClause(newClause);
                if (plans.size() == maxPlans)
                        break;
        }
        ...
}
```

Appendix C

Case Studies

C.1 BPEL files

Listing C.1: Trip Advisor System

```
<process name="TripAdvisor" ...
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

    <bpel:partnerLinks>
        <bpel:partnerLink name="FlightService" ... />
        <bpel:partnerLink name="CarService" ... />
    </bpel:partnerLinks>

    <variables>
        <variable name="exp" type="xsd:boolean"/>
    </variables>

    <sequence name="main">
        <receive name="ri" operation="ri" createInstance="yes"/>
        <pick name="pickMode">
            <onMessage operation="onlyCar">
                <invoke name="bc" operation="bc" />
            </onMessage>
            <onMessage operation="carAndFlight">
                <flow name="bookTransport">
                    <sequence name="getFlight">
                        <invoke name="bf" operation="bf" />
                        <invoke name="cf" operation="cf" />
                        <if name="If expensive">
                            <condition>bpws:getVariableData('exp')=string(true())
                               </condition>
                            <invoke name="expF" operation="ef" />
                            <else>
                                <invoke name="cheapF" operation="cf" />
                            </else>
                        </if>
                    </sequence>
                    <pick name="getCar">
                        <onMessage operation="rental">
                            <invoke name="bc" operation="car" />
                        </onMessage>
                        <onMessage operation="limo">
                            <invoke name="bl" operation="limo" />
                        </onMessage>
                    </pick>
                </flow>
            </onMessage>
        </pick>
        <invoke name="rd" operation="rd" />
    </sequence>
</process>
```

Listing C.2: Travel Booking System

```
<bpel:process name="TBS" targetNamespace="tbs" ...
         xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

    <bpel:partnerLinks>
        <bpel:partnerLink name="client" />
        <bpel:partnerLink name="flightSys" />
        <bpel:partnerLink name="hotelSys" />
        <bpel:partnerLink name="carSys" />
        <bpel:partnerLink name="billing" />
    </bpel:partnerLinks>

    <bpel:variables>
        <bpel:variable name="travelRequest"
                  messageType="tns:TBSRequestMessage"/>
        <bpel:variable name="output"
                  messageType="tns:TBSResponseMessage"/>
        <bpel:variable name="reservationData"
                  messageType="tns:TBSReservation"/>
        <bpel:variable name="bookingData"
                  messageType="tns:TBSBooking"/>
        <bpel:variable name="tries" type="xsd:integer"/>
        <bpel:variable name="availableFlights" messageType="tns:TBSFlightInfo"/>
        <bpel:variable name="availableCars" messageType="tns:TBSCarInfo"/>
        <bpel:variable name="consistent" type="xsd:boolean"/>
    </bpel:variables>

    <bpel:sequence name="main">
        <bpel:assign>
            <copy>
                <from expression="0"/>
                <to variable="tries"/>
            </copy>
        </bpel:assign>
        <bpel:receive name="receiveInput" partnerLink="client"
                 portType="tns:TBS"
                 operation="initiate" variable="travelRequest"
                 createInstance="yes"/>

        <bpel:flow name="pickReservations">
            <bpel:sequence name="pickFlight">
                <bpel:invoke name="getAvailableFlights" inputVariable="travelRequest"
                    outputVariable="availableFlights" partnerLink="flightSys"
                    operation="gaf" />
                <bpel:while name="While_available_lte_0_and_tries_lt_3">
                    <bpel:condition>$availableFlights.availableFlights <= 0 and $tries < 3
                        </bpel:condition>
                    <bpel:sequence name="flight_while">
                        <bpel:assign>
                            <copy>
                                <from expression="$tries + 1"/>
                                <to variable="tries"/>
                            </copy>
                        </bpel:assign>
                        <bpel:invoke name="updateTravelDates" inputVariable="travelRequest"
                            outputVariable="travelRequest" partnerLink="client"
                            operation="utd"/>
                        <bpel:invoke name="getAvailableFlights" inputVariable="travelRequest"
```

```
                            outputVariable="availableFlights" partnerLink="flightSys"
                            operation="gaf" />
                    </bpel:sequence>
                </bpel:while>
                <bpel:if name="If_available_flights_gt_0">
                    <bpel:condition>$availableFlights.availableFlights > 0
                        </bpel:condition>
                    <bpel:invoke name="holdFlight" inputVariable="availableFlights"
                        outputVariable="reservationData" partnerLink="flightSys"
                        operation="hd_f" />
                </bpel:if>
            </bpel:sequence>
            <bpel:sequence name="pickHotel">
                <bpel:invoke name="holdRoom" inputVariable="travelRequest"
                    outputVariable="reservationData" partnerLink="hotelSys"
                    operation="hd_h" />
            </bpel:sequence>
        </bpel:flow>
        <bpel:sequence name="decideTransportation">
            <bpel:pick name="pickTransportation">
                <bpel:onMessage partnerLink="client" operation="carAirport">
                    <bpel:sequence name="getCar1">
                        <bpel:invoke name="getAvailableRentalsAirport"
                            inputVariable="travelRequest" outputVariable="availableCars"
                            partnerLink="carSys" operation="gara" />
                        <bpel:if name="If_available_cars_gt_0">
                            <bpel:condition>$availableCars.availableCars > 0
                                </bpel:condition>
                            <bpel:invoke name="holdCar" inputVariable="availableCars"
                                outputVariable="reservationData" partnerLink="carSys"
                                operation="hd_c" />
                        </bpel:if>
                    </bpel:sequence>
                </bpel:onMessage>
                <bpel:onMessage partnerLink="client" operation="carHotel">
                    <bpel:flow name="getCarShuttle">
                        <bpel:invoke name="holdShuttle" inputVariable="travelRequest"
                            outputVariable="reservationData" partnerLink="carSys"
                            operation="hd_s" />
                        <bpel:sequence name="getCar2">
                            <bpel:invoke name="getAvailableRentalsHotelZone"
                                inputVariable="travelRequest"
                                outputVariable="availableCars"
                                partnerLink="carSys" operation="garhz" />
                            <bpel:if name="If_available_hotels_gt_0">
                                <bpel:condition>$availableCars.availableCars > 0
                                    </bpel:condition>
                                <bpel:invoke name="holdCar" inputVariable="availableCars"
                                    outputVariable="reservationData" partnerLink="carSys"
                                    operation="hd_c" />
                            </bpel:if>
                        </bpel:sequence>
                    </bpel:flow>
                </bpel:onMessage>
            </bpel:pick>
        </bpel:sequence>

        <bpel:invoke name="displayTravelSummary" inputVariable="reservationData"
            partnerLink="billing" operation="dts" />
        <bpel:pick name="Pick">
            <bpel:onMessage partnerLink="client" operation="cancel">
                <bpel:flow name="cancelReservations">
                    <bpel:invoke name="releaseFlight" inputVariable="reservationData"
                        partnerLink="flightSys" operation="rl_f" />
                    <bpel:invoke name="releaseHotel" inputVariable="reservationData"
                        partnerLink="hotelSys" operation="rl_h" />
                    <bpel:invoke name="releaseCar" inputVariable="reservationData"
                        partnerLink="carSys" operation="rl_c" />
                </bpel:flow>
            </bpel:onMessage>
            <bpel:onMessage partnerLink="client" operation="book">
                <bpel:sequence name="bkRes_gi">
                    <bpel:flow name="bookReservations">
                        <bpel:invoke name="bookFlight" inputVariable="reservationData"
                            outputVariable="bookingData" partnerLink="flightSys"
                            operation="bk_f" />
                        <bpel:invoke name="bookHotel" inputVariable="reservationData"
                            outputVariable="bookingData" partnerLink="hotelSys"
                            operation="bk_h" />
                        <bpel:invoke name="bookCar" inputVariable="reservationData"
                            outputVariable="bookingData" partnerLink="carSys"
                            operation="bk_c" />
                    </bpel:flow>
                    <bpel:invoke name="checkDates" inputVariable="reservationData"
                        outputVariable="consistent" partnerLink="billing" operation="chD"/>
                    <bpel:if>
                        <bpel:condition>$consistent = true()</bpel:condition>
                        <bpel:invoke name="sameDates" partnerLink="billing"
                            operation="sameDates"/>
                        <bpel:else>
                            <bpel:invoke name="notSameDates" partnerLink="billing"
                                operation="notSameDates"/>
                        </bpel:else>
                    </bpel:if>
                    <bpel:invoke name="generateInvoice" inputVariable="bookingData"
                        partnerLink="billing" operation="geIn" />
                </bpel:sequence>
            </bpel:onMessage>
        </bpel:pick>
    </bpel:sequence>
    <bpel:invoke name="informCustomer" inputVariable="bookingData" outputVariable="output"
        partnerLink="client" operation="inCu" />
</bpel:process>
```

Listing C.3: Flickr Visibility

```
<bpel:process name="flickr_visibility" targetNamespace="flickr_visibility" ...
        xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

    <bpel:partnerLinks>
        <bpel:partnerLink name="client"
                    partnerLinkType="tns:flickr_visibility"
                    myRole="flickr_visibilityProvider"
                    partnerRole="flickr_visibilityRequester"
                    />
    </bpel:partnerLinks>

    <bpel:variables>
        <bpel:variable name="perm" type="xsd:string"/>
        <bpel:variable name="consistent" type="xsd:boolean"/>
    </bpel:variables>

    <bpel:sequence name="main">
        <bpel:receive name="receiveInput" partnerLink="client"
            portType="tns:flickr_visibility" operation="initiate" variable="perm"
            createInstance="yes" />
        <bpel:scope name="upload">
            <bpel:if name="if private">
                <bpel:condition>$perm = "priv"</bpel:condition>
                <bpel:invoke name="uploadPriv"></bpel:invoke>
                <bpel:else>
                    <bpel:if name="If family">
                        <bpel:condition>$perm = "fam"</bpel:condition>
                        <bpel:invoke name="uploadFam"></bpel:invoke>
                        <bpel:else>
                            <bpel:invoke name="uploadPub"></bpel:invoke>
                        </bpel:else>
                    </bpel:if>
                </bpel:else>
            </bpel:if>
        </bpel:scope>
        <bpel:scope name="change_perm">
            <bpel:while name="While">
                <bpel:sequence>
                    <bpel:if name="If private">
                        <bpel:condition>$perm = "priv"</bpel:condition>
                        <bpel:pick name="Pick">
                            <bpel:onMessage operation="change_pub">
                                <bpel:invoke name="setPermPub"></bpel:invoke>
                            </bpel:onMessage>
                            <bpel:onMessage operation="change_fam">
                                <bpel:invoke name="setPermFam"></bpel:invoke>
                            </bpel:onMessage>
                        </bpel:pick>
                        <bpel:else>
                            <bpel:if name="If family">
                                <bpel:condition>$perm = "fam"</bpel:condition>
                                <bpel:pick name="Pick1">
                                    <bpel:onMessage operation="change_priv">
                                        <bpel:invoke name="setPermPriv"></bpel:invoke>
                                    </bpel:onMessage>
                                    <bpel:onMessage operation="change_pub">
                                        <bpel:invoke name="setPermPub"></bpel:invoke>
                                    </bpel:onMessage>
                                </bpel:pick>
                                <bpel:else>
                                    <bpel:pick name="Pick2">
                                        <bpel:onMessage operation="change_priv">
                                            <bpel:invoke name="setPermPriv"></bpel:invoke>
                                        </bpel:onMessage>
                                        <bpel:onMessage operation="change_fam">
                                            <bpel:invoke name="setPermFam"></bpel:invoke>
                                        </bpel:onMessage>
                                    </bpel:pick>
                                </bpel:else>
                            </bpel:if>
                        </bpel:else>
                    </bpel:if>
                    <bpel:invoke name="checkPerm" inputVariable="perm"
                        outputVariable="consistent"></bpel:invoke>
                    <bpel:if name="If consistent">
                        <bpel:condition>$consistent = true()</bpel:condition>
                        <bpel:invoke name="permOk"></bpel:invoke>
                        <bpel:else>
                            <bpel:invoke name="permNotOk"></bpel:invoke>
                        </bpel:else>
                    </bpel:if>
                </bpel:sequence>
            </bpel:while>
        </bpel:scope>
    </bpel:sequence>
</bpel:process>
```

Listing C.4: Flickr Comments

```
<bpel:process name="flickr_comments" targetNamespace="flickr_comments" ...
         xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

    <bpel:partnerLinks>

        <bpel:partnerLink name="client"
                  partnerLinkType="tns:flickr_visibility"
```

```
                    myRole="flickr_visibilityProvider"
                    partnerRole="flickr_visibilityRequester"
                    />
    </bpel:partnerLinks>

    <bpel:variables>
        <bpel:variable name="perm" type="xsd:string"/>
        <bpel:variable name="comments" type="xsd:boolean"/>
        <bpel:variable name="consistent" type="xsd:boolean"/>
    </bpel:variables>

    <bpel:sequence name="main">
        <bpel:receive name="receiveInput" partnerLink="client"
            portType="tns:flickr_visibility" operation="initiate" variable="perm"
            createInstance="yes" />
        <bpel:scope name="upload">
            <bpel:sequence>
                <bpel:if name="if private">
                    <bpel:condition>$perm = "priv"</bpel:condition>
                    <bpel:invoke name="uploadPriv"></bpel:invoke>
                    <bpel:else>
                        <bpel:if name="If family">
                            <bpel:condition>$perm = "fam"</bpel:condition>
                            <bpel:invoke name="uploadFam"></bpel:invoke>
                            <bpel:else>
                                <bpel:invoke name="uploadPub"></bpel:invoke>
                            </bpel:else>
                        </bpel:if>
                    </bpel:else>
                </bpel:if>
                <bpel:if name="if comments">
                    <bpel:condition>$comments = true()</bpel:condition>
                    <bpel:invoke name="setComON"></bpel:invoke>
                    <bpel:else>
                        <bpel:invoke name="setComOFF"></bpel:invoke>
                    </bpel:else>
                </bpel:if>
            </bpel:sequence>
        </bpel:scope>
        <bpel:pick name="Pick">
            <bpel:onMessage operation="addComment">
                <bpel:sequence>
                    <bpel:invoke name="addCom" outputVariable="result"/>
                    <bpel:if name="If added">
                        <bpel:condition>$result = "ok"</bpel:condition>
                        <bpel:invoke name="addOk"></bpel:invoke>
                        <bpel:else>
                            <bpel:invoke name="addNotOk"></bpel:invoke>
                        </bpel:else>
                    </bpel:if>
                </bpel:sequence>
            </bpel:onMessage>
            <bpel:onMessage operation="delComment">
                <bpel:sequence>
                    <bpel:invoke name="delCom" outputVariable="result"/>
                    <bpel:if name="If deleted">
                        <bpel:condition>$result = "ok"</bpel:condition>
                        <bpel:invoke name="delOk"></bpel:invoke>
                        <bpel:else>
                            <bpel:invoke name="delNotOk"></bpel:invoke>
                        </bpel:else>
                    </bpel:if>
                </bpel:sequence>
            </bpel:onMessage>
        </bpel:pick>
    </bpel:sequence>
</bpel:process>
```

C.2 Plan Quality Study

Both subjects are currently Computer Science Ph.D students at the University of Toronto. The plans are presented per violation and subject, in the order in which they would be applied by the respective subject.

C.2.1 Trip Advisor System

Safety Property Violation: t_2

- Subject #1:
  1. Cancel limo, leaving system in state where user picks ground transportation.
  2. Cancel flight, leaving system in state where user picks flight.
  3. Cancel flight and limo, leaving system in state user picks transportation mode.
- Subject #2:
  1. Cancel limo, leaving system in state where user picks ground transportation.
  2. Cancel flight, leaving system in state where user picks flight.
  3. Cancel flight and limo, leaving system in state user picks transportation mode.

Mixed Property Violation: t_1

- Subject #1:
  1. Cancel flight, inform user that a flight could not be booked.
  2. Cancel flight, inform user that <flow> activity labelled ② must be re-executed.

3. Restart entire system, informing user of problem.

- Subject #2:
  1. Re-invoke cf service.
  2. Cancel flight, re-execute <flow> activity labelled ②, booking a car and then another flight.
  3. Cancel flight, execute onlyCar branch of the <pick> activity labelled ①.

C.2.2 Travel Booking System

Safety Property Violation: t_1^{TBS}

- Subject #1:
  1. Cancel the flight and hotel bookings, re-display the travel summary and inform the user that the dates are inconsistent.
  2. Cancel all bookings and reservations, and inform the user that there was a date inconsistency.
- Subject #2:
  1. Cancel hotel booking, without cancelling flight or car bookings.
  2. Cancel all bookings and reservations, attempt to make new, consistent reservations.

Mixed Property Violation: t_2^{TBS}

- Subject #1:
  1. Cancel bookings, go back to <pick> activity labelled ③ and suggest that user re-execute pickHotel branch of this <pick> activity.
  2. Cancel bookings, go back to <pick> activity labelled ③ and suggest that user execute pickAirport branch of this <pick> activity.

3. Cancel bookings, cancel hotel reservation, make new hotel reservation (without changing flight reservation), re-execute pickHotel branch of the <pick> activity labelled ③.

- Subject #2:

  1. Cancel shuttle booking, book car from airport. Do not cancel anything else.
  2. Cancel shuttle and hotel bookings, hotel reservation, make a new hotel reservation, attempt to book a car from new hotel.
  3. Cancel everything.