Symbolic Testing Approach of Composite Web Services

Lina Bentakouk

Laboratoire de Recherche en Informatique
Ecole doctorale Informatique de Paris-Sud

Thèse

Test symbolique de services Web composites

Présentée par:
Lina Bentakouk
pour l'obtention du
Doctorat du l'université Paris-Sud XI

Jury

Pr. Ana Rosa Cavalli	IT/Télécom SudParis	Rapporteur
Pr. Manuel Nunez	Universidad Complutense de Madrid	Rapporteur
Pr. Mohand-Said Hacid	Université Claude Bernard Lyon 1	Examinateur
Pr. Philippe Dague	Université Paris-Sud XI	Examinateur
Pr. Marie-Claude Gaudel	Université Paris-Sud XI	Directrice de thèse
Dr. Fatiha Zaïdi	Université Paris-Sud XI	Co-Encadrant
Dr. Pascal Poizat	Université Paris-Sud XI	Co-Encadrant

1 décembre 2007 – 16 décembre 2011

Résumé

L'acceptation et l'utilisation des services Web en industrie se développent de par leur support au développement d'application distribuées comme compositions d'entités logicielles plus simples appelées services. En complément à la vérification, le test permet de vérifier la correction d'une implémentation binaire (code source non disponible) par rapport à une spécification. Dans cette thèse, nous proposons une approche boîte-noire du test de conformité de compositions de services centralisées (orchestrations). Par rapport à l'état de l'art, nous développons une approche symbolique de façon à éviter des problèmes d'explosion d'espace d'état dus à la large utilisation de données XML dans les services Web. Cette approche est basée sur des modèles symboliques (STS), l'exécution symbolique de ces modèles et l'utilisation d'un solveur SMT. De plus, nous proposons une approche de bout en bout, qui va de la spécification à l'aide d'un langage normalisé d'orchestration (ABPEL) et de la possible description d'objectifs de tests à la concrétisation et l'exécution en ligne de cas de tests symboliques. Un point important est notre transformation de modèle entre ABPEL et les STS qui prend en compte les spécifications sémantiques d'ABPEL. L'automatisation de notre approche est supportée par un ensemble d'outils que nous avons développés.
Mots clès: services, orchestration, test formel, génération de cas de test, WS-BPEL, système de transitions, exécution symbolique, SMT solver.

Abstract

Web services are gaining industry-wide acceptance and usage by fostering the development of distributed applications out of the composition of simpler entities called services. In complement to verification, testing allows one to check for the correctness of a binary (no source code) service implementation with reference to a specification. In this thesis, we propose black box conformance testing approach for centralized service compositions (orchestrations). With reference to the state of the art, we develop a symbolic approach in order to avoid state space explosion issues due to the XML data being largely used in Web services. This approach is based on symbolic models (STS), symbolic execution, and the use of a satisfiability modulo theory (SMT) solver. Further, we propose a comprehensive end-to-end approach that goes from specification using a standard orchestration language (ABPEL), and the possible description of test purposes, to the online realization and execution of symbolic test cases against an implementation. A crucial point is a model transformation from ABPEL to STS that we have defined and that takes into account the peculiarities of ABPEL semantics. The automation of our approach is supported by a tool-chain that we have developed.

Keywords: services, orchestration, formal testing, test-case generation, WS-BPEL, transition systems, symbolic execution, SMT solver.

Acknowledgments

I wish to take this opportunity to express my gratitude to my supervisors Fatiha Zaïdi and PascalPoizat for their support, good guidance and their patience during all these four years working on this thesis topic. Without their knowledge, valuable suggestions and others efforts, this thesis would never been written.

Special thanks are dedicated to Marie-Claude Gaudel, Frédéric Voisin, Burkhart Worlff and Delphine Longuet that have spent a great deal of their precious time for given me advices, comments and positive attitude. It has been a great pleasure to work with them. I would like to thank the people with whom I shared the office specially Johan Oudinet and Abderrahmane Feliachi, they had always been very supportive and helpful, we have had some great time together, and I am sure that there is still a lot more to come.

Warm thanks go to all my friends for their support specially: Sonia Dahdouh, Rania Khefifi, Vincent Armand, Boubacar Diouf. I would like to thank Sandrine-Dominique Gouraud who introduced me to Fatiha Zaïdi. I am really thankful to all the persons that I worked with. I am also grateful to all the administrative staff for their help and their kindness.

Many thanks go to my reviewers and examiners as part of my PhD defense: Ana Rosa Cavalli, Manuel Nunez, Marie-Claude Gaudel again, Philippe Dague and Mohand-Said Hacid. It has been a great honor to have you all as part of the examination committee.

My warmest thanks naturally go to my family, my parents Razika and Mohamed-Chèrif for having always stood behind me through the good and the bad times. I would probably not have gone so far in my studies without their encouragements and the education that they gave me. It is their supporting love that have made me who I am and brought me this far.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Using computer systems in our daily lives has become almost instinctive. In order to satisfy consumers' requirements, software and hardware are becoming increasingly complex. Such complexity raises doubts and trust issues about these systems.

Finding faults in a system, then correcting them avoids a waste of money for enterprises and/or prevents any threat to the safety of consumers that use those systems, *e.g.* the medical equipments, industrial systems (as medicines or food production), planes, vehicle construction, etc.

To ensure the correctness and the correct behavior of complex systems, testing remains an incontrovertible step in systems development. In the testing literature there are several kinds of tests. Unit testing aims at checking independently all the modules that constitute the system, integration testing aims at verifying the integration of these modules, regression testing addresses the correctness of a system after some change has occurred, etc. Further testing details will be provided in the second chapter.

Conformance testing is the kind of test that interests us. It aims at verifying whether an implementation is correct with regards to the specification of this system. Our objective in this thesis is the conformance testing of an implementation of a composite system with regards to its specification.

More precisely, we focus on the more recent architecture for systems composition *i.e.* Web services. A Web service is a computer application that allows communication and data exchanges between disparate systems via the Internet. A composition of Web services allows one to implement a series of calls to these Web services and thus, provides a complete business process to a client (or user), whether this is a human being or another software system.

Through this manuscript, we will present our approach to automatically generate and execute conformance test cases on a composition of Web services in order to establish its conformance with regards to its specification.

In this chapter, we start by an introduction to the context of our research which is related to the french ANR WebMov Project. Then we will explain the difficulties of conformance testing approach for Web service compositions. The solution that we propose to overcome these difficulties will be presented in the contribution section. Afterward, we present the outline of the thesis manuscript and finally, we end with our publications.

1.1 Context

During the last few years, the notion of software architecture based on services that provide altogether a business process was largely widespread. Such an architecture is called Service Oriented Architecture (SOA). Web services (WS) are software applications that adhere to the SOA vision. A basic WS is an application which provides functionalities to its user. WS can be composed into composite services in order to provide more functionalities, or to base the ones they provide on simpler sub-services.

This thesis has been carried out in the context of the ANR WebMov[1] project. This project focuses on the composition and the validation of WS. The objectives of this project could be summarized as follows: testing interface and robustness for WS, testing the behavioral aspects of their composition, developing a platform for the verification of WS and finally, executing the proposed approaches on realistic test cases. To achieve this, the WebMov project includes partners from research laboratories and from the industrial field. Our focus within this project is on testing of Web services composition.

1.2 Issues

The testing process aims at finding system faults. Applying test methods to composite Web services raises several issues. The major issues that we face for testing such systems are:

- **Composition specification**: How could we describe this composition ? In other words, which specification language could we use ?
- **Formal model**: How could we represent this composition in a formal way while respecting the features of the specification language ?

[1] http://webmov.lri.fr/

- **Testing approach**: Which testing method could we reuse/adapt for such systems ?

- **Rich data context**: How are the tests generated and/or executed for data-dependent systems, avoiding state space explosion issues ?

1.3 Contributions

We propose a comprehensive end-to-end approach for conformance testing of composite WS, more especially orchestration *i.e.* centralized WS composition.
To do so, we begin by representing the functionalities expected from a composite WS using the Abstract Business Process Execution Language (ABPEL), from which we retrieve a formal model named Web Service Symbolic Execution Systems (WS-STS). This model is obtained from our transformation rules that supports the main features of the Business Process Execution Language (BPEL).
We also provide the possibility to describe a Test Purposes (TP) as a WS-STS model allowing one to specify which scenarios she/he wants to test. In this case a product of the WS-STS model representing the specification, *i.e.* the functionalities expected from a composite WS, with the WS-STS of the TP is computed and yields a WS-STS product model. From the latter, we apply Symbolic Execution (SE) in order to generate a Symbolic Execution Tree (SET). The SE technique allows us to avoid state space explosion problems in presence of unbounded data types, as used in full-fledged BPEL.
Each path of the SET represents a symbolic test case. Resolving the constraints of a SET path using the Z3 SMT solver allows us to generate concrete test cases on-line while interacting with the implementation under test. Finally, a verdict on the conformance of the composite service with regard to its specification is emitted.

1.4 Outline

Chapter 2 is an introduction to testing and to Web services. It also contains a state of the art on the testing of Web services. Our symbolic approach for the conformance testing of composite Web services, is exposed in Chapter 3. In Chapter 4, we present details on our framework and tool support that is used to generate and then execute tests against an implementation of a composite service. Finally, we end the manuscript by a conclusion and perspectives of our work.

1.5 Publications

- Lina Bentakouk, Pascal Poizat and Fatiha Zaïdi. A Formal Framework for Service Orchestration Testing Based on Symbolic Transition Systems. In Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop (TESTCOM '09/FATES '09). LNCS 5826, pages 16-32, Springer 2009

- Lina Bentakouk, Fayçal Bessayah, Mounir Lallali, Wissam Mallouli and Andrey Sadovykh. A Framework for Modeling and Testing of Web Services Orchestration. In MDA4ServiceCloud 2010 - The 4th workshop on Modeling, Design, and Analysis for the Service Cloud, Paris, France. 2010.

- Lina Bentakouk, Pascal Poizat and Fatiha Zaïdi. Checking the Behavioral Conformance of Web Services with Symbolic Testing and an SMT Solver. In TAP'11 Proceedings of the 5th international conference on Tests and proofs In TAP'2011. LNCS 6706, pages 33-50 Springer 2011.

Chapter 2

State of the Art on Web Services Verification

With the ten last years, the concept of Service Oriented Architecture (SOA) has emerged in the computer science area. The idea is to use or reuse existing services to create new ones which offer more functionalities with a lower cost.

Web services are the most popular approach of SOA, where a Web service is the elementary component which uses a standard communication infrastructure to communicate through the Internet. The description of a Web service also known as its interface is depicted using a WSDL file, and the messages exchanges are done using the SOAP protocol. Web services can be composed in order to perform more functionality. These latter constitute the business process.

The business process could also be expressed as a specification. Where a specification describes the expectations of future users from a system. In the context of some Web service, a specification describes the expectations of a user over this Web service.

In this chapter, we present at first Web services and their composition. Next we talk about the formal methods that are intended to express the Web services formally to remove any ambiguity. Relying on a formal model we can now apply various testing approaches (unit testing, functional testing, regression testing, etc). We will explain what is a black box testing, a white box testing and the gray box one. We will also discuss about existing works on testing and verifying Web services, within a state of the art.

2.1 Web Services

As pointed before, Web services have become the cornerstone of the Service Oriented Architecture (SOA). Web services provide goods and services to the customers, they also allow the customers to search for services that match their needs. A Web service is a loosely-coupled application that allows communication and data exchange with its user via the Internet. The major advantage of Web service is that it could use other web services regardless of their implementation (Java, C++,...), the underlying architecture (J2EE, .NET,...) and the platform (Windows, Unix,...). A simple Web service provides simple functionality such as booking a plane ticket, bank account management, weather forecasting, while complex Web services (also called composite Web services) combine those functionalities to offer a complete higher-order business. Let us assume unifying the booking plane ticket service with a booking service for hotel and another car rental service. This composition of web services named Travel planning offers more functionalities to a costumer and saves him time and money searching and invoking each service individually.

Two kinds of composition of web services exist: a centralized one called an Orchestration of Web services, and a distributed one called a Choreography of Web services. An orchestration of services implies the use of a web service which is in charge to organize and harmonize the invocation of other Web services in order to provide a business process. In a Choreography of Web services, each service must satisfy a received request, with no or a partial information dealing with the next invocations of partner services involved in achieving a business process.

Several languages for describing services composition exist. The forerunners languages for composing Web services are the Web Service Business Process Execution Language (WS-BPEL) [117] used for describing an orchestration of services, and the Web Services Choreography Description Language (WS-CDL) [15] which describes a choreography. Thereafter, other languages for service composition appeared, such as the service modeling language *SENSORIA Reference Modeling Language* (SRML) [14] or the *Orc Programming Language* [85].

2.1.1 Service Oriented Architecture

The Service-Oriented Architecture (SOA) [48], is a software architecture that describes the system components and the interconnections that may exist in a high level way. The idea of SOA is to use existing applications as an enterprise system that provides services. Each service is self-described and aims to fulfill specific functionalities. Those services can then be combined to realize a business process.

An SOA provides a standard framework to represent services interactions as shown in Figure 2.1. A *service provider* publishes a description about his service in a public registry. The *service broker* manages the registry that contains information

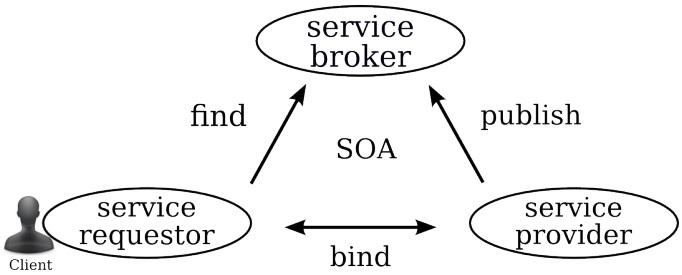

Figure 2.1: The SOA functional model

on other services, and allows the *service requestor* to find the service that suits its requests.

2.1.2 Presentation and features of Web Services

The main pillar of SOA is the use of Web services. They are widely deployed and easily integrated into users lives. Web services are applications that can be described, published, discovered and composed. They aim to provide goods and services according to users needs. This popularity is due to the fact that Web services are loosely coupled, Internet-enabled applications that process business activities performed by a single service or by interacting with other web services in order to fulfill the users demands.

The W3C[1] defines a Web service as: *A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

As we said previously Web services are loosely coupled. It means that they interact with each other dynamically using Internet more freely. It also implies that a change in the implementation of a web service functionality does not induce change on the client program that invokes the Web service.

2.1.3 Involved Technologies

Testing a system means interacting with it, *i.e.*, submitting input data to it then analyzing its responses. To do so, we must know the basic features and conditions of using this system. In the following we present the basic employed technologies for web services:

[1]World Wide Web Consortium (W3C) http://www.w3.org/

- **XML**. The *Extensible Markup Language* (XML) is a prevailing markup language that is used for transmitting structured data between applications. The widely use of XML is also due to its validating format. It means that XML must satisfies specific grammatical rules. XML Schema (also known as XSD for *XML Schema Definition* [157]) is the most powerful describing XML language. In the context of web services, XML documents allow services which are implemented differently to communicate and share data using the XML Schema format.

- **HTTP**. The *Hypertext Transfer Protocol* (HTTP) [75] is the usually used networking protocol for distributed and collaborative information systems.

- **SOAP**. The *Simple Object Access Protocol* (SOAP) [2] is a protocol specification that allows exchanging XML-based messages representing operations calls between the requester(s) and the Web service provider in decentralized and distributed environment like Internet or a Local Area Network (LAN). SOAP is not a new technological advancement it is just a simple way to codify the usage of existing Internet technologies in order to standardize distributed communications over the Internet. The structure of a SOAP messages is shown in Figure A.7. The SOAP envelop, wraps the start and the end of the message. The SOAP header part (with an optional contents) contains processing or control information such as information about authentication, session management, etc. The XML data that are exchanged between the applications are detailed in the SOAP body part.

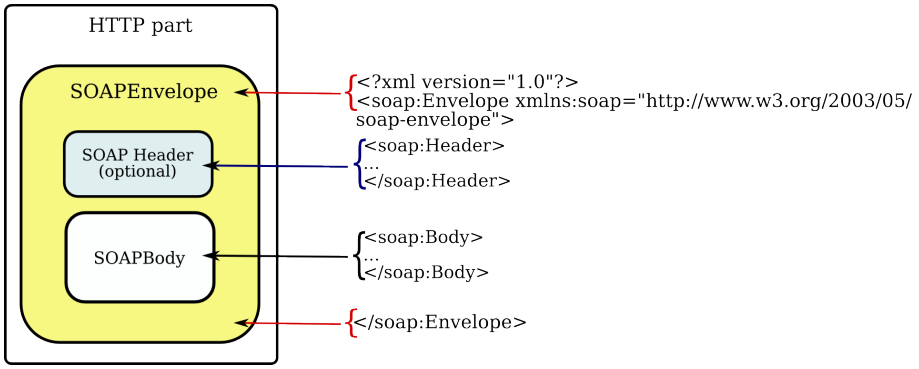

Figure 2.2: SOAP message structure

- **WSDL**. The *Web Services Description Language* (WSDL) [7], also known as the static interface is an XML-based language that provides a description of the functionalities of a Web service and how to access to it. A WSDL document can be divided into two parts [129]:

- a *service-interface definition* (or abstract section) that describes all the operations, their parameters and the abstract data types used
- a *service implementation* (or concrete section) that binds the first part (service interface definition) to a concrete network address with a specific protocol and exchanging concrete data structures.

Figure 2.3 presents a WSDL document structure where:

- **Types**: the types element contains XML schema or external references to XML schema that describe the data type used in the WSDL document
- **Message**: the message element is an abstract description of typed information used to communicate information between the invoker and the service.
- **PortType**: the portType element contains a set of operations. The portType binds the operation to a transport protocol such as SOAP. It represents the link between the service-interface and the service implementation
- **Operation**: represents the action exhibited by the service
- **Binding**: the binging element contains information that allows converting abstract service (the portType element) into a concrete representation. In other words, the binding element formats an operation and bounds it to a specific protocol
- **Service**: the service element contains a collection of ports elements. It associates an endpoint (such as an URL) with a WSDL binding element.
- **Port**: the port element defines an individual endpoint by specifying a single address for a binding.

WSDL document represents an operation with an input and/or an output. This representation induces four types of possible interactions with a service:

- **One way operation**: is an operation where the service receives a message without sending back a response. This kind of operation is used when a potential user of the service does not require an immediate response or no response at all.
- **Request/response operation**: is an operation where the service receives a message then it returns a message in response
- **Solicit-response operation**: is an operation where the service sends a message and expects to receive a replying message.

– **Notification operation** is an operation in which the service sends a message without expecting to receive a response. This type of operation could happen when a given service provides periodically a report to a given user.

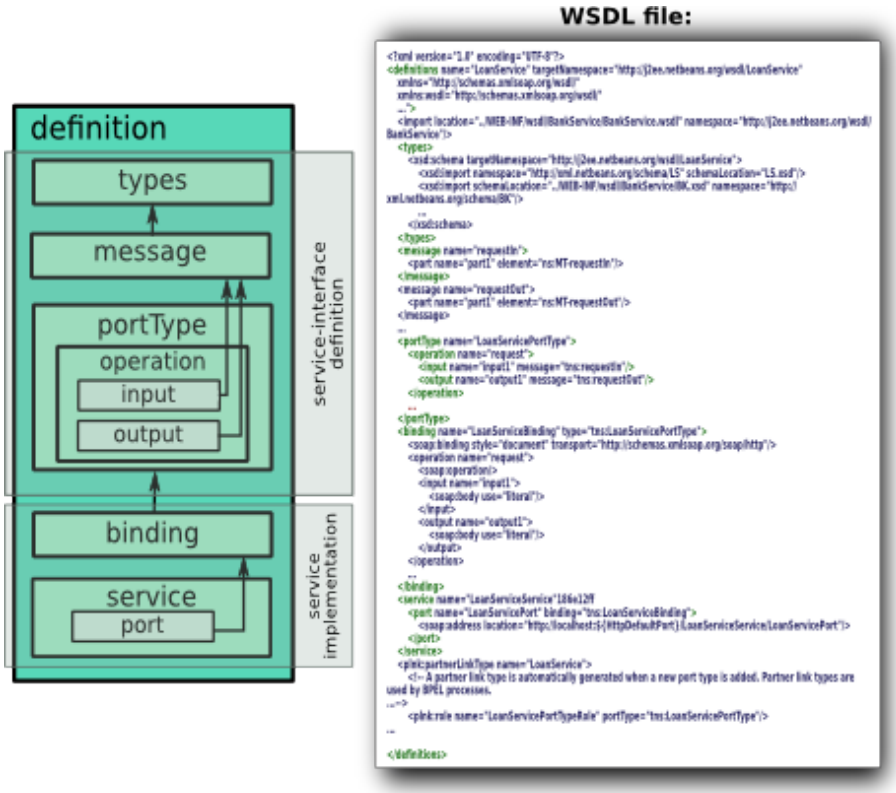

Figure 2.3: WSDL 1.1 structure

- **UDDI**. The *Universal Description Discovery and Integration* (UDDI) [6] is a register which contains information about Web services. It works as a catalog offering business functionalities and characteristics of different Web services. Each Web service features is obtained from the WSDL description. UDDI allows to [129]: (i) discover information about enterprises offering Web services, (ii) find description of Web services and (iii) find technical information about a Web service interface description. XMethods[2] site is an example of an UDDI.

[2]XMethods. http://www.xmethods.net/ve2/index.po

2.1.4 Web Services Composition

Many organizations adopted the concept of Web services (as *E-Learning* composition services or the *Travel Reservation Service*), this interest is due to their benefits. Indeed the idea with Web services, is to use or reuse existing services, it is also possible to compose them in order to build a new system that meets a specific business process.

A composition of services combines the use of multiple services in order to perform a particular complex task. Those services communicate by exchanging messages in a specific order. The procedure or sequence in which tasks are executed is known as the business process.

Such composition of web services may itself be provided by another service. Two approaches exist for designing a services composition the first one called a *Choreography* also known as a *Coordination* and the second an *Orchestration*:

- A *Choreography* of Web services describes a collaborative communication between Web services in order to achieve a specific goal. Such configuration looks like a distributed composition of systems. The global view that a choreography provides is a communication without any dominance by a participating service. The most popular language for service choreography is called *Web Services Choreography Description Language* (WS-CDL) [15]. The Figure 2.4, inspired from [99], depicts a graphic representation of a choreography.

- An *Orchestration* of Web services focuses on one Web service named *service orchestrator*, which manages the interaction with other service partners in order to accomplish a task. Such configuration of Web services represents a centralized composition of systems. The *Web Service Business Process Execution Language* (WS-BPEL) [117] is known as a standardized language for orchestration and the most used for this kind of composition. The graphic view associate to an orchestration, also inspired from [99], is presented in Figure 2.5.

Note that, among the services described within a choreography, it is possible to find a service orchestrator as one of the collaborating services.
Our testing approach focuses on an orchestration of services based on Web Service Business Process Execution Language (WS-BPEL). This language has emerged as a cornerstone to develop added-value distributed applications out of reusable and loosely coupled software pieces.
In the next part we present some features of the WS-BPEL to describe how an orchestration of service is implemented.

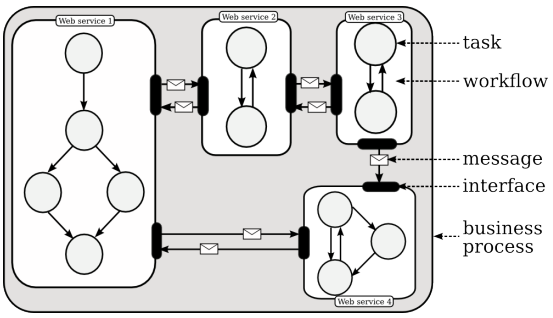

Figure 2.4: Service choreography

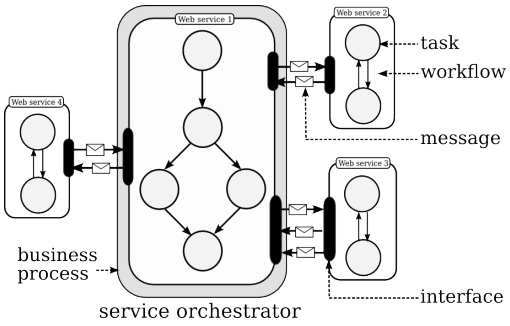

Figure 2.5: Service orchestration

2.1.5 Web Service for Business Process Execution Language

The *Web Service for Business Process Execution Language* (WS-BPEL) becomes the de facto standard for Web service orchestration. The WS-BPEL models the workflow of Web services composition by exporting and importing their functionalities, using their exposed interfaces. The description provided by this language can be employed in two ways. As an executable language that designs the exact behavior of the business process. Or as an abstraction of the process that provides a high level description in which roles of the services partners are depicted but details of concrete operations are hidden. The asset offered by the WS-BPEL with its executable and abstract process, advances the development of automated composition process. However, with such popularity, verifying WS-BPEL correctness becomes a topical issue, all the more because of BPEL complexity.

WS-BPEL (or simply BPEL) relies on WSDL, XML schema and XPath expressions. The WSDL interface of services is used in the composition and the WS-BPEL itself, offers a WSDL interface to describe the orchestration functionalities. The XML schema data are handled using the XPath expressions.

A Business process is described within a process tags as shown below. The attributes of the first process tag specify the name of the process, the target namespace to identify the namespace in which new elements are created, and the namespace that works as a reference to an element or a type.

```
1 <process
2         name="BPELprocessName"
3         targetNamespace="http://enterprise.netbeans.org/bpel/restOfURI"
4         xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
5         ...>
6         ...
7 </process>
```

All the following tags will be nested inside the process tag. Let us begin with the

import tag, as its name indicates, this activity is used to import the interface of the service orchestrator *i.e.* the WSDL file.

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
        location="BPELprocessServiceInterface.wsdl"
        namespace="URIlocation">
```

The involved Web services in the composition are then specified. These partner services are specified using the notion of *partner link* (partnerLink). Among the attributes of a partnerLink, we find its name , the *partner link type* that represents the functionality provided by the interaction between service orchestration and service partner. The function of the business process is indicated using *myRole* and the one of the partner using *partnerRole* keywords.

```
<partnerLinks>
        <partnerLink name = "partner1"
                partnerLinktype="partnerLT"
                myRole="BPELprocessRole"
                partnerRole="partner1Role"/>
        <partnerLink name = "partner2"
                .../>
        ...
        <!-- other partners -->
</partnerLinks>
```

In order to manipulate data, BPEL uses variables. These variables are used whether internally in the process or in messages exchanges. As said before, WS-BPEL relies on WSDL interface and this one handles data using XML schema. Consequently, the data types of WS-BPEL variables will be the same as those of WSDL messages, an XML schema element or an XML schema type (complex or simple types).

```
<variables>
        <variable name = "variable1" messageType="ns:messageType" />
        <!-- WSDL message type -->
        <variable name = "variable2" element="ns:elementName" />
        <!-- XML schema element -->
        <variable name = "variable3" type="ns:complexType" />
        <!-- complex type specified with its namespace -->
        <variable name = "variable4" type="string" />
        <!-- simple type -->
        ...
</variables>
```

WS-BPEL describes the process workflow using activities. We can distinguish two kinds of activities, the basic ones that describes atomic actions and the structured activities that describe complex actions by combining the simple activities.

Among the basic activities we find the *assign* activity. This activity is used to manipulate variables. Data stored within variables may be sent to a partner, received

from a partner or processed internally. This manipulation is done by storing a value into a variable of the same type.

Depending on the structure of a variable which can be of: (i) WSDL message type generally used for communicating activities with partners (these activities are the invoke, receive and reply. We explain them later). (ii) The variables may reference to an XML element. This element is mostly typed with a complex XML schema type. (iii) or XML schema types. To browse through variables of complex type, WS-BPEL uses XPath 1.0 expressions for that.

```
<assign>
        <copy>
        <from variable = "var1" />
        <!-- var1 and var2 are of the same type  -->
        <to variable= "var2"  />
        </copy>
        <copy>
        <from> $message/ns:subpart1/ns:subpart2 <from/>
        <!-- from the XPath expression to variable2 -->
        <to variable= "var3"  />
        </copy>
</assign>
```

One of the important aspects of an orchestration is the communication between Web services, for this matter simple activities were defined. The *receive* and *reply* are activities used for receiving and sending back messages between one or more services that have initiated the business process and the orchestrator. Among the attributes of the *receive* activity, one is dedicated to create a new instance of the process, the rest of them provide information about :

- partnerLink indicates the name of the partner link to use,
- portType indicates the WSDL port type in order to invoke the operation,
- operation indicates the operation to invoke,
- variable indicates the received variable.

```
<receive name = "nameOftheReceiveActibity"
        createInstance="yes"
        partnerLink="User"
        operation="operationUsed"
        portType="ns:servicePortType"
        variable="inputVariable">
</receive>
```

In response to the received message the *reply* activity is used. Its attributes are much like those of the receive activity, except for the values and the attribute for instance creation which is no more needed.

```
<reply name = "nameOftheReplyActibity"
        partnerLink="User"
        operation="operationUsed"
        portType="ns:servicePortType"
        variable="outputVariable1">
</reply>
```

The receive-reply activities are used for synchronous communication. The receive plays the role of request when the reply activity plays the role of the response. The asynchronous communications are done using the *invoke* activity which can be followed directly or not by a receive activity.

Invoking another Web service using the invoke activity for a synchronous communication implies the use of an input variable and an output variable. In this case the invoke activity will block until a response is provided from the invoked service. For an asynchronous communication only the input variable is used without need for the output variable. Example sending a receipt message for the web service partner.

```
<invoke name = "nameOftheInvokeActibity"
        partnerLink="servicePartner"
        operation="operation"
        portType="ns:servicePortType"
        inputVariable="VariableIn"
        outputVariable="VariableOut">
</invoke>
```

WS-BPEL defines also more complex activities named *structured* activities. The structured activities describe the organization of other activities. In the following, the described activities are those belonging to structured ones. To represent a sequential execution, the activity *sequence* is used.

```
<sequence>
        <!-- other activities nested inside -->
</sequence>
```

Parallel execution is represented using the *flow* activity. The flow activity is using links to define a synchronization dependence, in this case WS-BPEL support such constraint using *source* and *target* tags to satisfy the condition. For sake of simplicity we do not exemplify such case (more information can be found in the WS-BPEL standard [117]).
The next example describes the link "*rendez-vous*" that specifies a relation between the "*task 1*" as the source and the "*task2*" as the target. Nested within the "task 1" a synchronization between the first and the second services is required. Then the receive activity will be executed.

```
<flow name="flowActivityName">
        <links>
```

```
3                <link name= "rendez-vous" />
4                <!-- a link had only one source and one target  -->
5        </links>
6        <sequence name="task1" >
7                <source linkname="rendez-vous"/>
8                <invoke name="firstService"/>
9                <invoke name="secondService"/>
10       </sequence>
11       <!-- an activity inside the flow may have source and target for
             sevral links -->
12       <sequence name="task2">
13               <target linkname="rendez-vous"/>
14               <receive name = "thirdService"/>
15               ...
16       </sequence>
17 </flow>
```

Repeated actions could be expressed using several activities like *while, forEach,* or *repeatUnitl*. We exemplify the fist one *i.e. while*, that focuses on repeating the nested activities as long as the condition is satisfied.

```
1  <while>
2        <condition> $variable==true </condition>
3        <sequence>
4                <!-- do activities  -->
5        </sequence>
6        ...
7  </while>
```

Selective actions are represented using the *if* and *else* activities.

```
1  <if>
2        <condition> $variable==true </condition>
3                <!-- do some activities  -->
4        <elseif>
5                <condition> $variable2 &gt 100 </condition>
6                <!-- do other activities  -->
7        </elseif>
8        <else>
9                <!-- do activities  -->
10       </else>
11 </if>
```

However, WS-BPEL describe another sort of selective actions relying on message-dependent branching (at least one message) and an optional timeout. Such behavior is described using the *pick* activity. This activity describes the future behavior of the process, according to the received message (onMessage). It is possible to specify the waiting time for a receiving message using the *onAlarm* activity. This activity is mostly used for asynchronous operations.

```
1  <pick createInstance="no">
2        <onMessage partnerLink="User"
```

```
3                 operation="op1Name"
4                 portType="ns:userPortType1"
5                 variable="variable1">
6                 <!--  activities  -->
7         </onMessage>
8          <onMessage partnerLink="User"
9                 operation="op1Name"
10                portType="ns:userPortType2"
11                variable="variable2">
12                <!--  activities  -->
13        </onMessage>
14        <onAlarm>
15                <for> 'P0Y0M0DT0H10M0S' </for>
16                <!-- for a duration equivalente to 10 min  -->
17                <!--  then do activities  -->
18        </onAlarm>
19 </pick>
```

Using the activity *scope*, it is possible to specify a collection of actions. As for the other structured activities, simple or structured activities can be nested within a scope. The scope allows to define a part of a service behavior, with its own local variable, correlation and handlers.

```
1 <scope name ="scopeName">
2         <!-- do activities  -->
3 </scope>
```

Once, the service orchestration is made available on the Internet, it can be used by several users or services. Such scenario implies that there is a track for each process initialized instance. For this purpose, WS-BPEL proposes to use parts of transmitted data along with messages to correlate the exchanged messages for an instance. A correlation had a unique name, associate attributes and is defined within correlation set tag. The properties attributes are associated to operations in the WSDL file.

```
1 <correlationSets>
2         <correlationSet name= "UserCorrel"
3                 properties="UserToken"/>
4         ...
5 </correlationSets>
```

With these main activities we stop our introduction to the BPEL elements, more details are available on BPEL documentation [117]. We provide an example of an orchestration of services in the next part.

2.1.6 The xLoan Case Study

To help the understanding of our approach we introduce our *xLoan* case study.

It is an extension of the well-known loan approval example presented in the BPEL standard [117], which usually serves for demonstration purposes in articles on BPEL verification.

The Figure 2.6 presents an overview of this services orchestration. Where a user sends a loan request to the xLoan orchestrator service, this request includes the amount asked for and user personal information. According to those information and by invoking partner services, the xLoan service will provide an *approval* response with loan proposals or a *reject* response.
The specified partner services respectively deal with loan approval (BankService) and black listing (BlackListingService), with users not being blacklisted asking for low loans ($\leqslant 10000$) getting loan proposals without requiring further approval. Yet, once a loan is accepted, proposals may be sent to the requester. Further communication then takes place, letting the requester select one proposal or cancel, which is then transmitted to BankService. If the selected offer code is not correct the requester is issued an error message and may try again (select or cancel). Timeouts are also modeled, and the bank is informed about canceling if the requester does not reply in a given amount of time (2 hours).
Our extensions are targeted at demonstrating our support for BPEL important features: complex data types, complex service conversations including message correlation, loops and alarms. Hence, more complex and realistic data types are used, to model user information, loan requests and loan proposals.

2.2 Basic Notions of Testing

To ensure the good functioning and to increase trust in a system, the verification and the validation activities are inescapable. These two activities are complementary and jointly used all along the software development process (software requirements, design and programing).

The verification activity focuses on proving the correctness of a specification and/or an implementation of a system. To do so, the verification checks the specification and/or the program by applying proof and test techniques. By contrast the validation activity focuses on establishing that the system intends its requirements. The query "*Are you building the right thing?*" is usually applied for the validation and the query "*Are you building it right?*" is the one used for the verification.

2.2.1 Verification and Testing

Some researchers associate the testing process to the verification. Others claim a distinction between verification and testing. The aim of verification, also known as formal verification, is to proof the correctness of a system by analyzing one or more

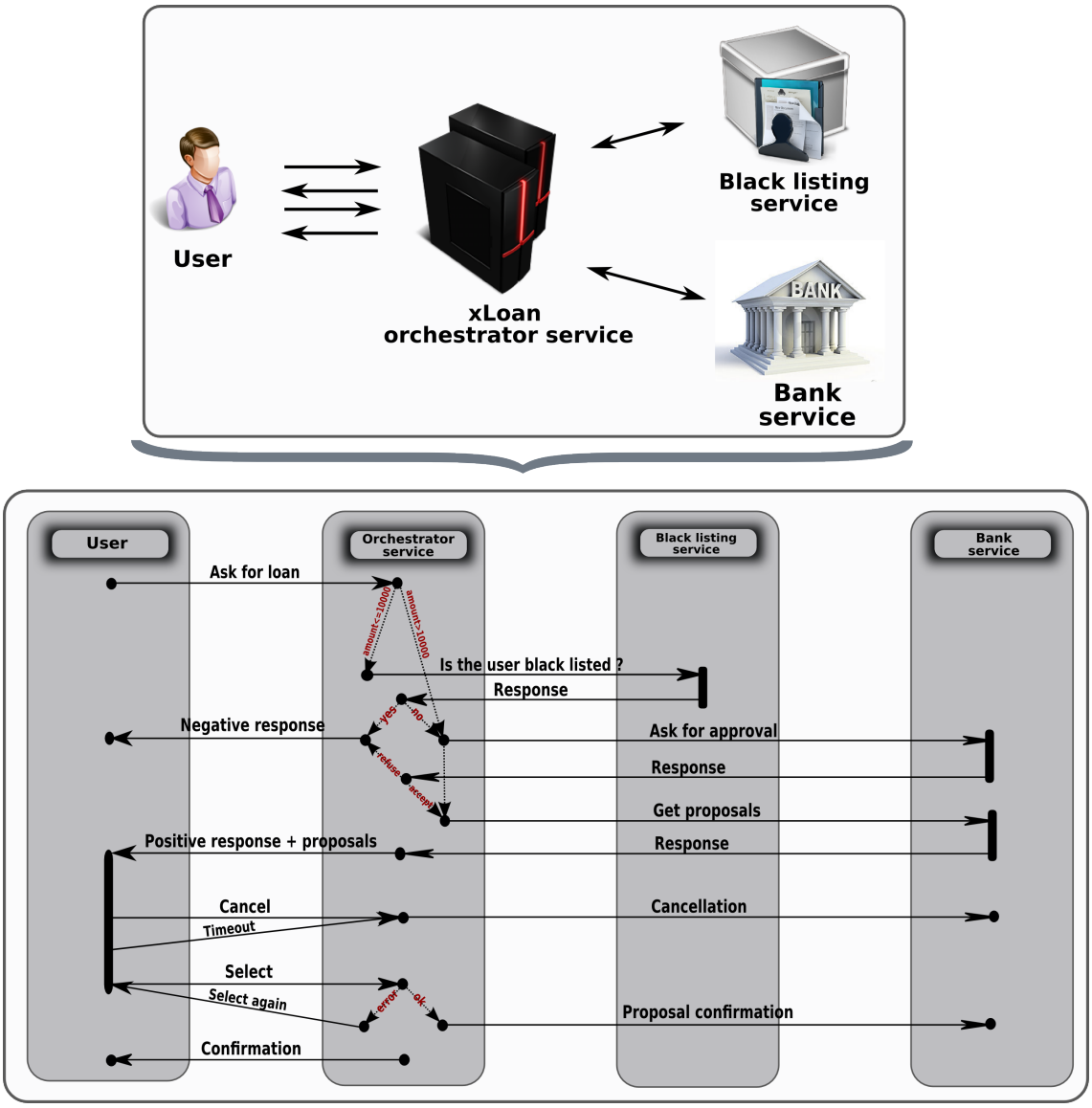

Figure 2.6: Overview of the xLoan services orchestration

properties of the system model. Still, testing is the process of executing a program in order to find errors.

Among the main techniques of formal verification we cite the *Model-checking* technique. The model checking algorithm takes as input an abstraction of the behavior of a system (*i.e.*, a transition system) and a formula of some temporal logic, and responds whether or not the abstraction satisfies the formula. We then say that the transition system is a model of the formula. The big advantage of model checking is that it is completely automatic, and usually a counterexample is returned when

the property is not checked.

In a testing domain, Abran et al [17] define the testing process as follows: *Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.*
The testing process is also defined as the process that detects the system errors regarding to its specification, and thus enhance the reliance in it (LAPRIE, J.C. [89]).

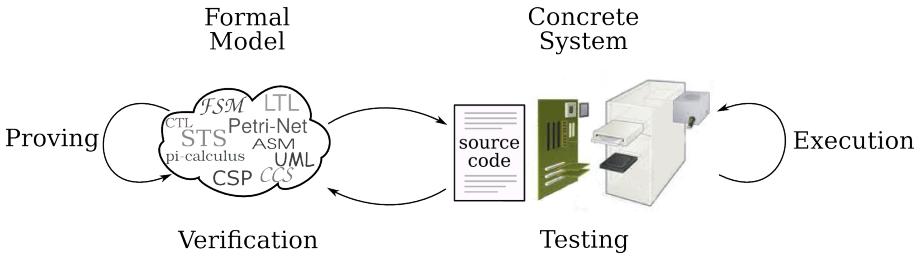

Figure 2.7: Verification and test

The Figure 2.7, provides a standard view of the verification and the testing process. Since we are interested in finding errors due to a non conformance of a composite Web service implementation regarding to its specification, the work presented in this thesis focuses on the testing process of an orchestration of Web services.

2.2.2 Levels of Testing

There are several levels of test, depending on the detail level of the system that a tester wants to verify, such as: unit testing, integration testing, system testing, etc. Furthermore, those tests could be applied at different levels of abstraction: (i) *Black box* testing, consists in making tests on the external observable behavior of the system without having implementation code knowledge. (ii) *White box* testing is based on the internal details and the structure of the code. A third method that mixes the previous ones is called (iii) *Gray box* testing. We provide more detail in the following: It is possible to apply a test at different levels of details for a system as depicted in the Figure 2.8:

- Unit testing: it verifies a small module (i.e. a class) of a software independently of other modules. This kind of test is used with white box testing,
- Integration testing: it verifies that the interaction of two or more modules belonging to the same system produces the expected results. It is used with White or black box testing,

- System testing: it verifies the entire behavior of the system. It includes other kinds of testing such as: performance, stress, scalability testing, etc. Used with Black box testing

- Acceptance testing: generally conducted by a customer. This testing verifies that the system satisfies the acceptance requirements. It is used for Black box testing

- Regression testing: this test is applied when new functionality are added to the system. It uses all the previous tests applied and run them again to verify that the new functionality does not affect the old ones,

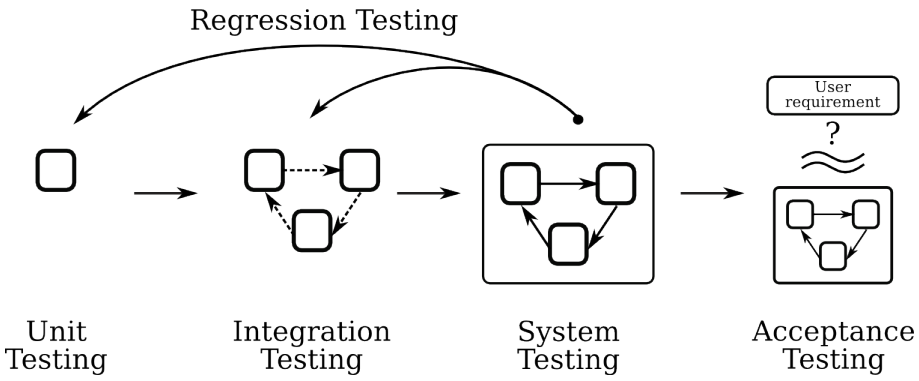

Figure 2.8: Evolution of the testing process

As said above, performing a test depends also on the level of abstraction. In the following paragraphs we provide a definition of the white an the black box testing.

- White box testing. Also referred as *structural* testing. It is a test method that relies on the knowledge of the internal structure of the program, *i.e.* the tester has access to the source code of the program. This method verifies that a functionality is correct by using code coverage. Among the techniques used in this method we cite: the *Data-Flow* testing that analysis whether the variables of the program are bound to a value or not and how these variables are used; the *mutation* testing aims to test faulty hypothesis by introducing small transformation to the original program.

- Black box testing. Also referred as *functional* testing. It is a test method that verifies that the functionalities of a program are in accordance with the user requirement. Such test is done without any information on how the system was created. In other words, the tester uses the black box testing to make sure that the program behaves correctly by sending inputs data to the program and

comparing its outputs with those specified by the user.

The white and the black box testing are represented in Figure 2.9.

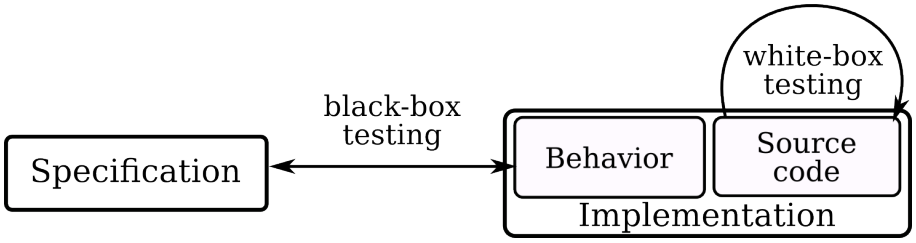

Figure 2.9: Test abstraction

When a tester chooses the testing type and the associate level of abstraction, he aims to test a specific aspect or characteristic. We describe some aspects as follows:

- Conformance testing: it determines whether a system meets the functionalities described in the specification,
- Robustness testing: it focuses on the resistance aspect of a system implementation against external events or unanticipated errors in the specification of this system,
- Performance testing: it determines the performance of a system or checks whether the announced performances in the specification are achieved,
- Security testing: it determines if the data and the functionalities of a system are safe and well protected against any intrusion.

The Figure 2.10 is reused form Tretmans's tests classification. It summarize the kind of testing that one may perform.

In the next part we provide basic test notion in order to apply conformance testing of an orchestration of services.

2.2.3 Conformance Testing

As mentioned before, conformance testing also called *Compliance* testing is process of checking if the implementation of a system adheres to its specification. In other words, the conformance notion is a formal relation between a specification and an implementation of it. This relation evaluates whether the implementation satisfies the properties described in the specification. The Figure 2.11 is an overview of the conformance testing process for which we present the basic notions:

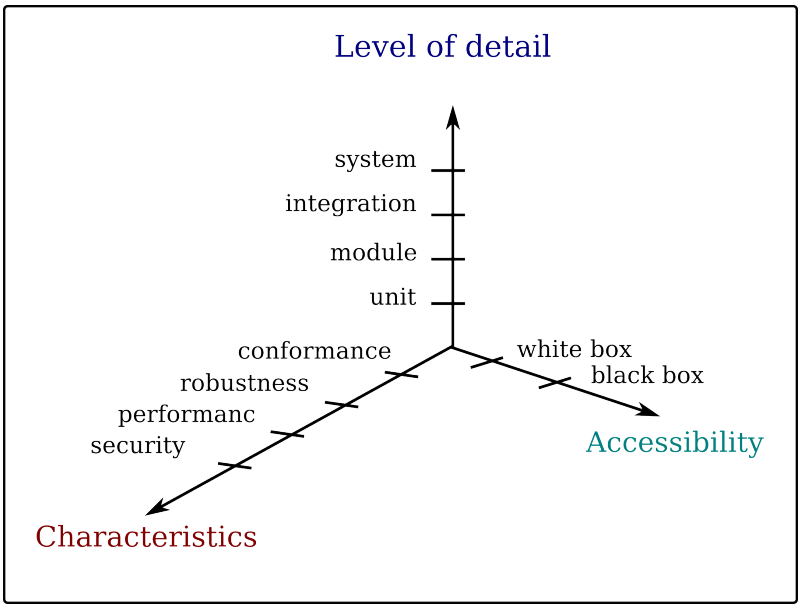

Figure 2.10: Tests classification

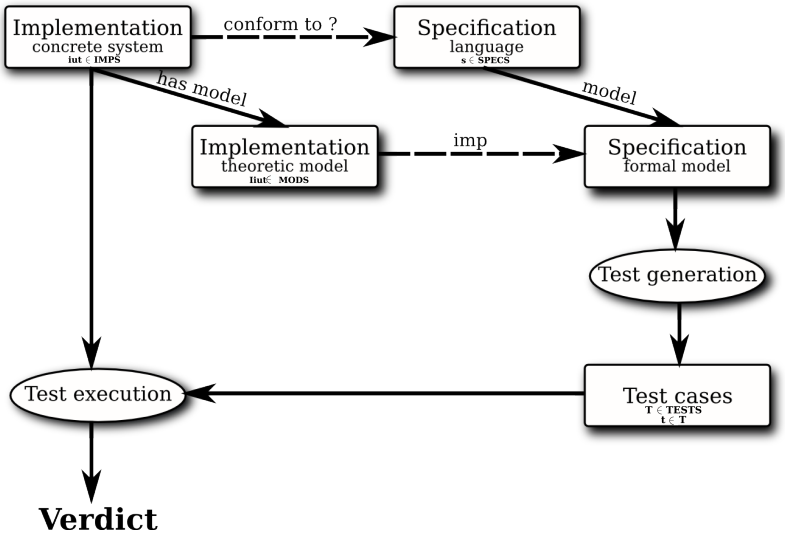

Figure 2.11: The formal specification based testing process.

- *Specification.* A system specification is a formal or an informal description of what the implementation should do. It is a way to express the properties of the system (see Figure 2.12). Generally, the specification is established from the user requirements it can be expressed as natural language e.g. French, English, or as specialized description languages, e.g. LOTOS [ISO/IEC 8807]

or SDL [ITU-T]. The semantics of the specification can be represented using Temporal Logic [127], algebraic specifications, extended finite state machines, transition systems, etc. In the following we will use the same formalization as J. Tretmans [146, 147] to express the notion of conformance.
Let $SPECS$ denote the set of all valid expressions of formal specifications and s a single specification such that $s \in SPECS$.

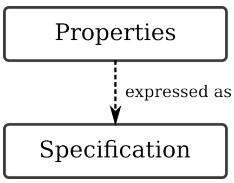

Figure 2.12: Properties as a specification

- *Implementation.* The Implementation Under Test (*iut*) represents the system to be tested. The *iut* can be a piece of hardware like a single chip or a software component like a piece of executable code.
  Let $IMPS$ denote the set of all possible implementations and *iut* is a one of them such that $iut \in IMPS$.

- *Conformance.* To check the conformance of an implementation with respect to its specification we need to find a formal relation between the implementation and its specification. A correct implementation *iut* ($iut \in IMPS$) with respect to its specification s ($s \in SPECS$) is represented with the following relation:

  conform-to $\subseteq IMPS \times SPECS$

  However, whereas the specification s is represented formally, the implementation under test *iut* is a real object. In order to formally reason about a concrete implementation (*iut*), it is assumed that $MODS$ is a set of all the possible implementation models. We assume I_{iut} to be a formal model describing the *iut* such that $I_{iut} \in MODS$. In the testing theory domain, this assumption is called *test hypothesis* [28, 67, 66]. The test hypothesis allows us to reason on implementations under test as if they were formal implementations in $MODS$. As described in Zinovieva thesis [176] we can express the test hypothesis in a semi-formal manner as follows:

  $\forall\ iut \in IMPS \Rightarrow \exists\ I_{iut} \in MODS$. Where *iut* is modeled by I_{iut}.

  Once the implementation is represented formally, the conformance between the model of the implementation and the specification could be expressed by a formal relation called *implementation relation*:

  imp $\subseteq MODS \times SPECS$.

In other words, the implementation model I_{iut} is correct with respect to the specification s if I_{iut} **imp** s

- *Testing.* Actually it exists two kinds of testing : *active* testing and *passive* testing.
  - Active testing implies performing experiments on an implementation under test and observing the corresponding reactions,
  - Passive testing also called *monitoring*, it implies observing then analyzing the inputs and the outputs of an implementation under test without disturbing its normal execution.

  In our approach we are interested by the active testing, thus we will interact with the implementation in order to check its response with regard to the expected one.When a specification depicts a single experiment, it is called *test case* and the process of its execution is called *test execution.* Let $TESTS$ denote the domain of all the experimentations and t a single one such that $t \in TESTS$.

  If the response observed from the system is the expected one then the implementation has passed the test. Otherwise, it is considered as an erroneous response so, the implementation has failed. However, a third case may happen, when the response is an unexpected one. It means that the response belongs to another scenario depicted by the specification, in this case the verdict is called *inconclusive* and the test case will be re-executed. Yet at the end of the test the final verdict will be either a pass or a fail one.

  For an implementation *iut*, we could express such observation as follows:
  iut **passes** t for a successful test execution, *iut* **fails** t for unsuccessful one and *iut* **inconclusive** t for an unexpected response observation. The extension of this definition to a set of test cases called *test suite* T such that $T \subseteq TESTS$ so:

  $$iut \textbf{ passes } T \Leftrightarrow \forall\ t \in T : iut \textbf{ passes } t.$$

  $$iut \textbf{ fails } T \Leftrightarrow \exists\ t \in T : iut \textbf{ fails } t.$$

- *Test generation.* Producing tests from a specification for a given implementation is called *test generation* or *test derivation.* It is expressed as:
  $SPECS \rightarrow P(TESTS)$, where $P(TESTS)$ is a function that refers to the set of all subsets of TESTS.

- *Conformance Testing.* An implementation is conform to its specification if the responses produced by the implementation during the test execution are correct. This is formalized as:

$\forall\ iut \in IMPS$ and $\forall\ T \in TESTS$:

$$iut \textbf{ conform-to } s \Leftrightarrow iut \textbf{ passes } T$$

This suggests that the test suite T is *complete*. The completeness requirement means that all the possible test cases generate successful answers. Nonetheless, in a realistic point of view, it is impossible to support an infinite number of test cases. Therefore, a weaker requirement is posed: the test suite must be *sound*.

The soundness means that if an implementations is considered as conform to its specification then these implementation must pass the test suite. Sill, it is possible that some non-conform implementation pass the test suite and therefore will also be considered as conform. In other word, if an implementation is identified as non-conform to its specification it involves that it will not pass the test suite but the contrary is not true.
The sound requirement is deducted by reading the completeness formula form the left to the right:

$\forall\ iut \in IMPS$ and $\forall\ T \in TESTS$:

$$iut \textbf{ conform-to } s \Rightarrow iut \textbf{ passes } T$$

Reading the other way, form the right to the left, is known to be the *exhaustiveness* requirement. This means that the implementation that passes the test suite is conform, and any non-conform implementation is detected. It corresponds to :

$\forall\ iut \in IMPS$ and $\forall\ T \in TESTS$:

$$iut \textbf{ passes } T \Rightarrow iut \textbf{ conform-to } s$$

In a practical testing even the exhaustiveness property could not be applied because it is not possible to execute an infinite number of tests in an amount of time. Thus the property that must be hold is the one of soundness.

- *Test Execution.* As defined by Elena Zinovieva [176], it refers to the process of interacting with the implementation under test *iut* ($iut \in IMPS$). This interaction is described as : (1) submitting test cases to the *iut*, (2) observing the produced response from the *iut* and (3) emitting a test verdict.
The execution process implies applying a test case t ($t \in TESTS$) on the model representing the implementation under test I_{iut} ($I_{iut} \in MODS$) which produces a subset of observations OBS. A formal interpretation of an observation OBS after an execution *exec* is represented as follows:

  exec: $TESTS \times MODS \longmapsto P(OBS)$

  The verdict indicates if the implementation is conform or not to its specification, i.e. if the implementation passes the tests then it is conform to the specification

otherwise, it is not. We present v_t as the *verdict function*, we represent it formal as:

$$v_t\colon OBS \mapsto \{Fail, Pass\}$$

Finally, we formalize an execution e of a test cases $t \in TESTS$ against an implementation $iut \in IMPS$ that may produce a fail or a pass verdict as follows:

$$iut \textbf{ fails } t \iff \exists\ e \in \textbf{exec}(t, I_{iut}),\ v_t(e) = Fail$$

$$iut \textbf{ passes } t \iff \neg\ (iut \textbf{ fails } t)$$

The generalization of such rules to a test suit $T \in TESTS$ is represented as:

$$iut \textbf{ fails } T \iff \exists\ t \in T,\ iut \textbf{ fails } t$$

$$iut \textbf{ passes } T \iff \neg\ (iut \textbf{ fails } T) \iff \forall\ t \in T,\ iut \textbf{ passes } t$$

After this introduction to the conformance testing, in the next section we present the existing specifications for an orchestration of services.

2.3 Specification for Web Services Orchestration

Composing Web services provides the possibility to accord and organize the functionalities offered by these Web services. So one may achieve his purpose, without losing time searching for which service satisfies his request. For this goal, an elaborate description of those functionalities is required.

Different specifications in the context of Web services exist to depict the Web service composition. The most popular language for specifying an orchestration of services is the (WS-BPEL). This language is an executable one, it implies that WS-BPEL describes code details on how the interactions between services are done. It exists other languages more abstract to describe a specification of services composition like: SENSORIA Reference Modeling Language (SRML) and the Business Process Modeling Notation (BPMN) or the Unified Modeling Language for Service Oriented Architecture (UML4SOA).

In the following we provide a brief description of those specification languages:

2.3.1 Unified Modeling Language for Service Oriented Architecture

Relying on the standardized Unified Modeling Language (UML), the UML for Service Oriented Architecture (UML4SOA) [111] is an extension that supports behavioral description of Service Oriented Software (SOA). This model had been defined in the context of the *Software Engineering for Service-Oriented Overlay Computers*

(SENSORIA) project [12]. Form UML4SOA specification model transformer are used to generate executable code as WS-BPEL and WSDL files. The UML4SOA profile allows its users to model an orchestration of services as UML2 diagrams in a simple way using small set of model elements.

2.3.2 Business Process Modeling Notation

We can say that the Business Process Modeling Notation (BPMN) [8] is the most popular specification used in the industrial fields due to its user friendly representation. BPMN is a graphical representation of business process of a system. This notation is considered as a standardized bridge between a business process design and a process implementation. With the popularity of using services orchestrated, a lot of work [160, 119, 9] focus on extending BPMN to supports the generation of executable WS-BPEL code.

2.3.3 SENSORIA Reference Modeling Language

The SENSORIA Reference Modeling Language (SRML) [52, 18, 14] supports the design of services at a high level of abstraction. It has also been defined for the SENSORIA project [12]. SRML is inspired by the Service Component Architecture (SCA) [11] composition. The SCA architecture provides several technologies in order to model services components and the way to interact with them.
In addition to the functional properties of a service composition, SRML focuses also on run-time discovery and services binding.

2.3.4 Discussion

In order to apply our conformance testing approach, we are interested in a specification that describes the global conversation protocol (the business process) of services orchestration. But also the operations used, the data exchanged between the services and some special properties like the correlation that links the messages with the appropriate process instance.

The UML4SOA model provides a complete description of the operations and the messages exchanged within a service orchestration. However, the conversation protocol that defines the ordering of those exchanged messages are still not well represented. In contrast, the BPMN notation provides a good description of the conversation process, but describes the data traffic using a high level of abstraction. As for the SRML, the associate model for an orchestration supports the operations, data exchanges description and the conversation protocol. Yet this description focuses on the architectural aspect of the services components.

For our approach we rely on the (A)BPEL which is an *abstraction* of the WS-BPEL. The ABPEL describes the conversation protocol with the operations and the exchanged data without details on the implementation code. Since the WS-BPEL or the ABPEL do not have a graphical representation and for sake of pedagogy we chose to use BPMN for graphic representation of BPEL process which is independent form the tool used to describe a BPEL service orchestration. We extended the BPMN notation to highlight the data handling in an orchestration. We also used and extended the UML diagram in order to represent the structure of transfered data.

In the following section, we present how based on a specification, a formal model is used to represent an orchestration of services and then apply verification and testing techniques in order to check its correctness.

2.4 Formal Methods for Web Services

A formal model allows the representation of behavioral characteristics of a system according to precise rules, thus any ambiguity or repetitions are prevented. In order to gain a thorough understanding and reliability by avoiding ambiguity or misunderstanding of what Web services must provide, the services are expressed using formal models.

This section provides an overview of the existing approaches to verify or test Web services using formal models.

We can divide the methods of verifying Web services in two sorts: The *control flow* verification and the *data flow* verification. The control flow verification uses temporal logic model checking to prove properties of services, bi-simulation to check the behaviors equivalence between two services or two versions of the same service, simulation to check whether the behavior of a service is included within the behavior of other interacting services, or execution traces of the service, to understand the behavior of the service. The data flow verification uses data type checking, in the case of LOTOS and other process algebras allowing data handling.

2.4.1 Models of Web Services

Within this part, we present the main models adopted in web services formalization as shown in the Figure 2.13.

- Petri-Nets [167]. They are particularly useful for modeling concurrent systems and asynchronous processing. A Petri-net is a directed bipartite graph in which nodes are either "*places*" (represented by circles) or "*transitions*" (represented by bars). A *place* represents a state or a condition to satisfy and a *transition*

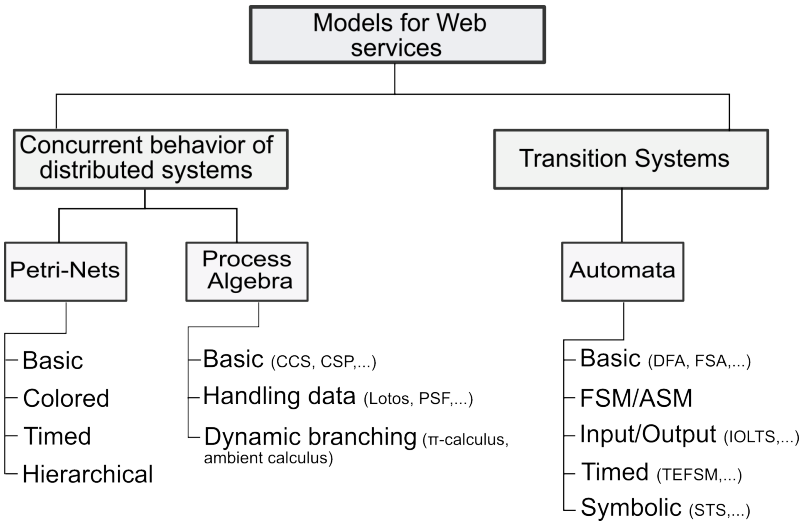

Figure 2.13: Formal models classification

represents an action. *Places* may also contain tokens, these represent the available resources. The token moves form a place to an other by executing actions. The Petri-nets are widely used for modeling the control flow of Web services composition [69, 72, 142, 154, 101]. In [69] the authors provide a Petri net-based algebra, for modeling the control flows, as a necessary constituent of reliable Web service composition process. The proposed algebra is expressive enough to capture the semantics of complex Web service combinations. As for Christian Stahl et al. [142, 121] they present a pattern-based Petri net semantics for the Business Process Execution Language for Web Services (WS-BPEL). This semantics is complete, and covers the standard behavior of BPEL as well as the exceptional behavior (e.g. faults, events, compensation). The tool BPEL2PN [142] is a parser that takes a BPEL process as an input, and transforms it into a Petri net according to the associate Petri net semantics. The output of the tool is a Petri net expressed as a data format of the model checker LoLA [137].
Other research works propose the use of colored Petri nets, as the work of Yingmin et al. [94] where they use colored tokens to represent the faults in a BPEL process. Kang et al. [78] also use colored Petri net (CPN), their model allows efficient composition of Web services and validates the correctness of composition using formal verification methods. As for Timed Petri nets, they were used by Valero et al. [132] to handle priorities and time constraint for services choreography. Other works like [166] use hierarchical Petri nets for the verification of composite services.

- Process Algebra [144]. They describe a diverse family of related approaches to formally model concurrent systems. Several process algebra are defined as: the Calculus of Communicating Systems (CCS) [114], Communicating Sequential Processes (CSP) [73], π-calculus, the Language Of Temporal Ordering Specification (LOTOS) [36], etc. The Algebra of Timed Processes (ATP) is used in [108] to formally model the behavior of web services described in BPEL language. Driven by the ATP rules the WSMod tool, in this work, produces a model represented as a Label Transition System (LTS) which will be analyzed by model checking using the Construction and Analysis of Distributed Processes (CADP) [10, 61] toolbox. CCS was used in [171] as a formal tool to specify and model web services behavior using weak bi-simulation relation to discover services that satisfies a given requirement. W.L. Yeung [168] uses CSP to express the behavior of services composition either as a centralized composition described using WS-BPEL or as distributed one described using WS-CDL. The generated model is verified using the FDR2 [103] model-checking tool. Foster et al. [53, 54] present an approach to specify, model, verify and validate web services compositions. To do so, they proposed the use of Finite State Processes (FSP) with the Labeled Transition System Analyzer (LTSA) tool to produce and analyze Labeled Transition Systems (LTS). In [159] M. Weidlich et al. present a partial formalization of BPEL using π-calculus for process verification. Finally LOTOS and its CADP toolbox are used in several work [134, 51] to verify the BPEL process.

- Automata. They consists of a set of states related by transitions that can be labeled using labels from predefined set. Zhang et al. [173, 174] define an automata-based model named Web Service Automaton (WSA) to capture the most features of BPEL without handling complex data or predicates. The WSA model is then transformed into input languages as PROcess MEta LAnguage (Promela) or SMV for the model checker tools SPIN or NuSMV respectively, to apply coverage criteria methods. The authors of [80, 81] model and analyze time-related properties of a web service composition expressed using BPEL. The formalism they choose for representing a BPEL process is called Web Service Timed State Transition Systems (WSTTS). After that model checking techniques are applied on WSTTS formalism to verify the BPEL composition. Another automata called the annotated deterministic finite state automata (aDFA) model is presented in [161] to allow discovering other web services. The Annotated finite state automata (aFSA) model presented in [106], is used for service discovery by indexing and matching the modeled business processes. Pistore et al. [126] use a different automata model named the State Transition System (STS). Given an STS obtained form the translation of BPEL and the formalization of the requirements obtained using Extended Goal Language

(EaGLe) [86], the authors describe how to generate an STS that encodes a process behavior which satisfies these requirements. The latter STS is then transformed into an executable BPEL program. In order to trace faults when a business process fails, the work presents in [123] proposes an automatic method to model Web service behaviors and their interactions using model-based reasoning approaches on Discrete-Event Systems (DES).
In [59, 60] Bultan et al. present a set of tools and techniques for analyzing interactions of composite web services which are specified in BPEL and communicate by exchanging asynchronous XML messages. To do so, the authors first of all present a framework that translates BPEL into an intermediate representation named Guarded Finite State Automata (GFSA) then into the Promela verification language. The translation steps are done using their tool named the Web Service Analysis Tool (WSAT). Finally, they use the generated Promela with the SPIN tool for the verification of the BPEL composition. For their part, Garcia-Fanjul et al. [63] were interested in transforming BPEL into the Promela language then submit it to SPIN in order to derive a test suite for the composition.
For a functional testing of service composition, M. Lallali [87] uses the Web Service Timed Extended Finite State Machine (WS-TEFSM) model as the formal description of a BPEL process. The WS-TEFSM model handles the temporal activities, the termination and the exceptions expressed in the BPEL process. Yuan et al. [170] define a graph structure to represent BPEL, then search concurrent test paths with a matrix-based algorithm. This approach defines an extension of Control Flow Graph (CFG) named BPEL Flow Graph (BFG) that represents a BPEL program as a graphical model. Then concurrent test paths can be generated by traversing the BFG model, and test data for each path can be generated using a constraint solving method.

- Other formalisms exist, Butler et al. [37] described the Structured Activity Compensation (StAC) language used to specify BPEL composition activities in long running business transactions. StAC supports sequential and parallel behavior as well as exception and compensation handling. Moreover, the StAC language can also be combined with the B notation to specify the data aspects of transactions. This combination provides a rich formal notation which authorizes for succinct and precise specification of business transactions.

2.4.2 Discussion

As presented above numerous model-based verification and testing approaches have been proposed for BPEL as translating BPEL to automata, Petri nets or process algebras. These approaches are especially valuable to check if an orchestration

specification is correct. Still, as far as the correctness of an implementation wrt. a specification is concerned, these approaches fall short as, *e.g.*, one may expect service providers to publicize (Abstract) BPEL descriptions of *what* their services do, but not *how* they do it. Here, testing comes as a solution to ensure (i) that some sub-service used in an orchestration really conforms to its publicized behavioral interface, and (ii) that the service orchestration itself conforms to the behavioral interface to be publicized after its deployment.

With reference to the previous works, our interest was in a model that depicts the control flow of a services orchestration and supports the rich XML-based data types available in BPEL. This is achieved thanks to the STS model which has already proven to be valuable, *e.g.*, for UML [19], and is here used for services orchestration. From ABPEL as specification language and using BPMN and UML for a graphical representation of the business process and handled data, we compute the associate Symbolic Transition System (STS) model according to process algebraic transformation rules, more details are provided in the next chapter.

2.5 Verifying and Testing Web Services

Using formal models of Web services, several approaches and techniques are used for testing and verifying them. However, before talking about such approaches, we present some issues faced in the test community in order to test Web services.

2.5.1 Web Services Testing Issues

Web services are considered as a new challenge in the test field. The difficulty in testing the Web Services is due to their complexity (for composition) and to the notion of control and observability services. Inspired from [116] we describe these challenges as follows:

- The activity of a Web service (publishing, finding, binding) makes it difficult to test a service due to its dynamic aspect. Suppose a user who asks to have a service that meets a specific request. For that a Web service *WS1* can be offered at time *t1*, and another Web service *WS2* at time *t2* because the service *WS1* is no longer available. Such a scenario makes testing *WS1* according to its activity difficult.

- The choice of the specification. Indeed, if we consider the point of view of Specification-Based Testing, such issue is very relevant. From which specification, should a tester begin ? a static one (*e.g.*, WSDL) or a dynamic one. Moreover for the latter choice which language should the tester selects (WS-BPEL, BPMN, SRML or UML for SOA, etc) ?

- The control and observability of Web services also complicates their test. In general the web service provider does not expose their source code which is understandable for a confidential and secure point of view. But testing a source code with a formal white box approach is also important because it increases the trust level of the service. Complementary to the white box testing the black box testing will rely on the specification of the service to test it.

- Testing the composite Web services. Both for the black or the white testing have to manage the complexity of composite services due to the multiple parties involved in business process. This complexity also includes how to handle complex data transmitted between the different services partner and how to associate each exchanged message to its corresponding process instance.

2.5.2 How to Verify and Test Web Services ?

As pointed out before (in section 2.4), significant research effort have been produced in the last years in order to propose formal models for orchestration verification and testing. Numerous work in this area have addressed model-checking, to check if properties are verified by an orchestration specification. Still, in presence of a black box implementation, one cannot retrieve a model from it. To help establishing the conformance wrt. a specification, testing should be used. However, to the contrary of model-checking, testing is incomplete. One rather focuses on generating the *good* test cases to search for errors. Testing enables one to ensure both that sub-services participating to an orchestration conform to their publicized behavioral interface, and that the orchestration itself conforms to the behavioral interface to be publicized after its deployment. Orchestration testing has mainly been addressed from a white-box perspective, assuming that the orchestration implementation source code is available. In practice, the source code is often not available as it constitutes an added-value for the service providers. In this section we present an overview of some works done in Web Services verification and testing domain.

2.5.2.1 Unit Testing of Web Services

This is the most basic technique applicable to any system. Each individual component of the system is verified independently. In the case of Web services each operation can be considered as a basic unit. Unit testing of web service consists of sending and receiving SOAP messages between the tester and the service under test. The message exchange is done through information extracted from the WSDL file. Thus, this type of test allows to verify the proper functioning of the operation and to check the WSDL file. Automated testing tools exist for unit testing of Web services such as Parasoft SOAtest [4], SOAPSonar [3], HP Service Test Software and Oracle Application Testing Suite [1]. However, these tools do not automate all the testing process: a

tester provides the test cases then the tools generate the appropriate SAOP message for each test case. Unit testing was also addressed by the research community. Mayer and Lubke [110] propose a framework for testing BPEL composition with a white box testing method. This framework called BPELUnit [109] can replace participating web services using service mocks, exchanging SOAP messages with other services and collecting test results. BPELUnit also manages synchronous or asynchronous BPEL process that interacts with several partners. Li et al. [96] propose another approach for testing BPEL process. They present a BPEL4WSunit test framework that includes an abstract model of a BPEL process, a test architecture and stub process to simulate services partners.

2.5.2.2 Testing the Interface of Web Services

This refers to the test of WSDL file. To achieve this goal Salva et al. [135] propose an automatic testing method of WSDL interface. This method allows the automatic generation of test cases, and to verify the handling of sessions, exceptions and operations existence. As well Bartolini et al. [24] define an automatic approach for testing WSDL descriptions, which combines the coverage of WS operations with data-driven test case generation. They use the popular tool soapUI [13] with their developed tool named TAXI [23, 31, 32] to generate test suites. The TAXI tool derives automatically XML instance from the XML schema. Their methodology for the generation of test cases, uses basic coverage criteria and some heuristics. Those heuristics aim to combine the generated instance elements in different ways, and sometimes change the cardinalities and the data values used for the generated instances. In [148], Troschutz provides a framework for testing Web services using the Testing and Test Control Notation version 3 (TTCN-3) [5]. TTCN-3 is a standardized test specification and implementation language dedicated to black-box testing of computers and telecommunication systems. Using specific mapping rules, an abstract test suite is derived from a WSDL description of a Web service then executed against the Web service by the TTCN-3 tool. In [20], Bai et al. present an automatic method for generating test cases form WSDL interface. They begin by parsing WSDL in order to generate the corresponding DOM tree, then test cases are derived from the messages data types and operation dependency. Bertolino et al [30] focus on testing the interoperability between Web services. In this work, the authors strengthen a WSDL description with a corresponding UML2.0 diagram and Protocol State Machine (PSM) model. The PSM describes the required conditions and how a customer can access to the service. From the PSM a Symbolic Transition System (STS) is generated on which formal testing theory and tools are applied for conformance evaluation.
Generally, the interface testing of Web services is included within the black box testing approach. We apply such separation to distinguish between testing the

interface of Web services composition and testing their behavior.

2.5.2.3 Behavioral White Box Testing

Several works interested in structural testing of BPEL were performed. Liu et al. [98] present a test model called WS-BPEL Control Flow Graph (BCFG) that represents the control flow of a BPEL process. The test paths are then derived form the BCFG model and path conditions are collected and analyzed to keep only satisfiable paths and thus produce test cases to execute against the service composition.
Using the BPEL Graph Flow (BGF) model Yuan et al [170] generate concurrent test paths then test data are obtained from a constraint solving method. The produced test paths and test data are finally combined to produce complete test cases. The eXtended Control Flow Graph (XCFG) model was used to represent a BPEL program, and from it sequential test paths are generated, then combined to form concurrent test paths, in the last step of this method a constraint solver named BoNuS [172], which is an extension of a Boolean satisfiability checker, is used to solve the constraints of these test paths and generate feasible test cases. Dong et al. [46] analyze the structure of a service composition to generate the corresponding High-level Petri Nets (HPN) model. Testing coverage and reduction techniques are applied on the HPN model to generate test cases.

2.5.2.4 Behavioral Black Box Testing

The functional testing of Web services aims to verify the accordance of the services with regard to the requirements of the specification. Based on the specification of a service Kaschner et al. [79] introduce their approach that automatically generates test cases in a purpose of conformance testing. From a specification expressed using BPMN or abstract BPEL they translate it in an Open Work Flow Nets (oWFNs), which is a particular class of Petri Nets. This approach focuses on detecting the non existence of deadlocks between the different services partners. They consider each service partner as a test case, *i.e.*, if the specification describes the interaction with a service partner, it must be the case in the implementation. In [55] Frantzen et al. attend to the coordination protocol. First of all the authors use the UML diagram representing the communication between a client and a Web service as a specification, then the Symbolic Transition Systems (STS) is generated and testing techniques are applied according to the ioco [145] conformance relation.

An offline approach for testing Web services is exposed by Frantzen et al. [56]. The authors present their tool named *Jambition* to test Web services modeled as Symbolic Transition System (STS). Relying on the *sioco* testing relation, they provide a random on-the-fly testing approach. This work is very close to ours, the main differences are in the generation of test cases, while we do generate test cases then execute them against the implementation, Frantzen et al. [56] do it on-the-fly. They

execute random testing while we already support the test purposes and we use the symbolic execution in order to avoid state space explosion problem.

A similar approach as our is described in Escobedo Del Cid [50] thesis. Where the author focus on conformance testing of an orchestration in context. By the notion of context the author presents a strongly coupled conformance relation with the testing architecture. After that Escobedo uses the symbolic execution and a rule-based online testing algorithm, in order to detect errors resulting from the interaction between the tester and the orchestrator, according to a specific context. A second contribution depicted within this work is to define Web service testing techniques, and thus offering the possibility for one to determine if a Web service can interact with the service orchestrator without leading this new composition into a deadlock situation. As for the execution aspect, the author provides a prototype implementing the online testing algorithm in order to apply contextual conformance testing on a service orchestrator.

2.5.2.5 Testing Web Services Approaches

We report the main research works on testing Web service in the Table 2.5.
The first column refers to the authors of the work. Then the approach used for the verification or the testing process is defined in the second column. The described approach can be:

- Specification-Based, this approach resets on documents that describes a Web service like WSDL, or Web Ontology Language for services (OWL-S). Where the OWL-S is a semantic markup for composing Web services.
- Fault-base, it aims to prove the non existence of predefined errors
- Model-Based, is a technique for deriving test cases form formal models. Those models represent the Web service or their composition.

The abstract testing level applied for the chosen approach is depicted in the third column, where the white square represents the white box testing and the black one represents the black box testing. The gray box testing is represented with a star.
Each approach aims to check a specific type or point of view, it could be unit testing, robustness testing, model-checking, etc, as indicate in the fourth column. And each type focuses on precise criterion, these criteria are indicated in the fifth column. The next column points to the input specification used. The seventh column presents the formal model used for the approach. Finally, if any of the mentioned research work uses a tool, this is indicated in the last column.
We used abbreviations in the representation of the formal models, here are their complete definition:
Petri-nets (PN), Timed Predicate Petri-Nets (TPPN), Unified Modeling Language

(UML), Web Service Automaton (WSA), WS-BPEL Control Flow Graph (BCFG), eXtended Control Flow Graph (XCFG), BPEL Graph Flow (BGF), High-level Petri Nets (HPN), Symbolic Transition System (STS), ASynchronous Extended Hierarchical Automata (ASEHA), Task precedence graph (TPG), Timed Labeled Transition Systems (TLTS), XPath Rewriting Graph (XRG), Data Flow Diagrams (DFDs), Timed Exetended Finite State Machine (TEFM), Test Ontology Model (TOM), Java Interclass Graph, Protocol State Machine (PSM).

2.5.2.6 Discussion

Our objective is to apply conformance testing on an orchestration of services with a black box approach. It means that we are interested in testing the behavior of the service orchestrator, which implies that the order of services invocations and the order of the called operations are very important, and not only its WSDL interface.

To do so, we formally represent the orchestration from an ABPEL specification. We chose to use the Symbolic Transition System (STS), to support the data exchanged between the involved services. As the work of Escobedo et al. [49], we reuse the symbolic execution technique, to bypass the states explosion problem. Unlike the works of Lallali et al. [88], Cao et al. [39], or Morales et al. [115] where they face this kind of problem since they use concrete data in their approaches. However, as the previous approaches we use test purpose criterion too. A test purpose aims to guide and focus the generation of test cases on a specific aspect.

Nevertheless, the work of Escobedo et al. [49] and the one in Escobedo Del Cid [50] thesis do not target the same goal as our. While, the authors focus on the testing architecture for the conformance testing of an orchestration by simplifying data exchanges, we are interested in handling the characteristics of an orchestration, *e.g.*, structured exchanged messages and correlation property, in order to instantiate then execute test cases against the service orchestrator.

To provide the appropriate input data for testing the orchestration implementation, we use an SMT solver during the generation of test cases. These data inputs are submitted to the service orchestrator during the several interactions with it. According to its replying messages, we check if the service is conform to its specification or not.

The added value of our work compared to the other conformance testing of BPEL orchestration using a black box approach, is the fact that (i) it is a complete approach (also called end to end approach). Where we begin by an ABPEL specification, until the execution of the generated test cases.(ii) Within our STS model we also support the exchanged data between the services and the correlation property. (iii) Using the symbolic execution and an SMT solver we generate test cases with input data that we use for testing the orchestration service.

2.6 Conclusion

In this chapter we introduced the context of our work. We presented the SOA, as well as the main pillar of this architecture, which are web services. The composition of web services allows one to produce more efficient and dedicated systems, but also more complex and less secured ones. Software testing focuses on the detection of errors that can occur using such systems. Through this chapter we have introduced the main concepts of testing, with more details on conformance testing approach which is our objective. We have presented existing specification of composed services, formal models for Web services and testing approaches for them.

In the next chapter we deeply present our contribution for testing the orchestration of web services. We will describe there the formal model of the orchestration, the generation of test cases and the process of execution those test cases against the orchestrator service.

Authors	Approach	B/W	Type	Criterion	From	Model used	Tools
Sneed and Huang [141]	Specification-based	□	unit testing	—	WSDL	—	WSDLTest
Bartolini et al. [27, 24]		■	test data generation	Coverage criteria some heuristics	WSDL	?	WS-TAXI
Ma et al. [104]		■	test data generation	—	WSDL	Model for XML schema	—
Bai et al. [20]		■	test data generation	From XML data Types and dependency analysis	WSDL	—	—
Li et al. [95]		■	test data generation	?	WSDL	—	WSTD-Gen
Hanna and Munro [70]		■	fault-based robustness	—	WSDL		—
Offutt and Xu [118]		■	fault-based generation	Data perturbation	WSDL	—	—
Heckel and Lohmann [71]		■	interoperability conformance	—	WSDL	UML	—
Xu et al. [163]		■	Fault-based	Coverage criteria	WSDL	tree model	—
Mei and Zhang [112]		■	Fault-based	?	WSDL	—	—
Mayer and Lübke [110, 109]	Testing	□	unit testing	—	BPEL, WSDL	—	BPELUnit
Chan et al. [40]		□	unit testing integration	Metamorphic relations	Metamorphic service	—	—
Tsai et al. [149]		■	integration	XML-based	WSDL	—	Coyote
Peyton et al. [124]		★	integration	—	SOAP, WSDL	—	TTCN-3
Tsai et al. [150]		□	collaborative	Test scripts	UDDI	Sequence diagram	—
Pau [16]		■	regression	—	—	virtual	JOpera

Table 2.1: Related work on Web Services Testing

Authors	Approach	B/W	Type	Criterion	From	Model used	Tools
de Almeida and Vergilio [44]	Fault-based	■	—	Data perturbation	SOAP	—	SMAT-WS
Martin et al. [107]		□	—	?	WSDL	—	WebSob
Looker et al. [102]		■	—	Fault injection	SOAP	fault-model ontology	WS-FIT
Siblini and Mansour [138]		□	mutation	Mutant operator	WSDL	—	—
Lee et al. [90]		■	mutation	Mutant operator	OWL-S	OWL-S ontology model	—
Vieira et al. [155]		■	robustness benchmark	—	WSDL SOAP	—	—
Wang and Huang [158]		■	mutation	—	OWL-S	WSA	—
García-Fanjul et al. [63]	Model-based	□	model checking	Transition coverage	BPEL	Promela model	SPIN
Zheng et al. [174] and Zheng and Krause [173]		□	model checking	Coverage criteria in temporal logic	BPEL	WSA	SPIN, NuSMv
Huang et al. [76]		□	model checking	Check concurrency	OWL-S	?	BLAST
Dai et al. [43]		□	model checking	Reachability safety	BPEL	TPPN	MCT4WS
Betin-Can and Bultan [33]		■	model checking	Interaction properties, conformance	BPEL	HSM	JavaPathfinder, SPIN
Fu et al. [59, 60]		■	model checking	LTL properties	BPEL	Guarded	SPIN
Ramsokul and Sowmya [131] and Ramsokul and Sowmya [130]		■	model checking	interoperability conformance	WS protocols	ASEHA model automata, kripke structure	SPIN
Schlingloff et al. [136]		■	model checking	usability with ATL	BPEL	PN	Lola
Tsai et al. [152]		■	model checking	data selection	OWL-S	Swiss Cheese	—

Table 2.2: Related work on Web Services Testing - Suite

Authors	Approach	B/W	Type	Criterion	From	Model used	Tools
Mateescu and Rampacek [108]	Model-based	■	model checking	discrete-time properties	BPEL	LTS	WSMod CADP
Yeung [168]		■	model checking	trace refinement	WS-CDL BPEL	CSP	FDR2
Foster et al. [53, 54]		■	model checking	deadlocks safety property	BPEL	LTS	—
Weidlich et al. [159]		■	model checking	compatibility consistency	BPEL	π-calculus	ABC - checker
Salaün et al. [134] Kazhamiakin et al. [80, 81]			model checking	time requirement	BPEL	WSTTS	—
Kang et al. [78]		■	formal verification	correctness reliability	BPEL	CPN	—
Ouyang et al. [120]		□	formal verification	?	BPEL	PN	BPEL2PNML, WofBPEL
Dong et al. [46]		□	formal verification	coverage and reduction techniques	BPEL	HPN	Pose++
Xu et al. [162]		□	formal verification	soundness effectiveness	BPEL	PN	BPEL2PNML
Lohmann et al. [100]		■	formal verification	controllability operating guidelines	BPEL	PN	BPEL2oWFN, Fiona
Yang et al. [165, 166]		■	formal verification	syntactic correctness termination, etc	BPEL WSCI	PN	CPNTools
Yi and Kochut [169]		■	formal verification	—	BPEL	PN	CPNTools
Conroy et al. [41]		□	unit testing	?	WSDL	—	—
Yan et al. [164]		□	unit testing	coverage criteria	BPEL	XCFG	—
Li et al. [92]		□	unit testing	flow analysis using PageRank	RDF schema	RDF graph	WS-StarGaze
Tsai et al. [151]		□	unit testing integration	reliability model	WSDL	ASTRAR	—
Li et al. [96]		□	unit testing	Input/output business process	BPEL	ABPEL	—

Table 2.3: Related work on Web Services Testing - Suite

Authors	Approach	B/W	Type	Criterion	From	Model used	Tools
Lenz et al. [91]	Model-based	□	test data generation	—	—	Extended UML	—
Paradkar et al. [122]		■	test data generation	?	OWL-S	IOPE paradigm	—
Blanco et al. [35]		□	search-based methods	scatter search	BPEL	State graph	—
Guangquan et al. [68]		□	test	coverage criteria	BPEL	UML 2.0	—
Yuan et al. [170]		□	test	coverage criteria	BPEL	BFG	—
Hou et al. [74]		□	test	path exploring	BPEL	MSG	—
Ma et al. [105]		■	test	W-method	BPEL	stream X-machines	—
Dong and YU [45]		■	test	fault-coverage	WSDL	HPN	—
Endo et al. [47]		□	integration	coverage criteria	BPEL	PCFG	CValidBPEL
Mei et al. [113]		□	integration	rewriting rules, data flow criteria	BPEL	XRG	—
Ruth and Tu [133]		□	regression	coverage information	WSDL	CFG	—
Lin et al. [97]		■	regression	RTS techniques	WSDL	JIG composition language	—
Fu et al. [58]		□	robustness	fault injection	Java code	?	—
Salva and Rollet [135]		■	robustness		WSDL		—
Bai et al. [21]		□	collaborative	Constraint-guided	OWL-S	PN	—
Bartolini et al. [26]		□	collaborative	coverage information	WSDL	sequence diagram	—

Table 2.4: Related work on Web Services Testing - Suite

Authors	Approach	B/W	Type	Criterion	From	Model used	Tools
Bai et al. [22]	Model-based	■	partition testing	—	OWL-S	TOM	TCGen4WS
Tarhini et al. [143]		■	interaction	reliability	WSDL	TPG	—
						TLTS	
Bertolino and Polini [29]		■	interoperability	?	WSDL	UML	—
					UDDI	PSM	
Dai et al. [42]		■	interoperability	?	OWL-S	PN	CBT4WS
Smythe [140]		■	interoperability	—	WSDL	UML	—
Li et al. [94], Li [93]		■	monitoring	quality of service	BPEL	CPN	—
Pencole et al. [123]		■	monitoring	—	BPEL	DES	—
			conformance				
Bartolini et al. [25]		□	conformance	coverage criteria	WSDL	DFD	—
Kaschner and Lohmann [79]		■	conformance	—	BPEL	Open nets	—
Sinha and Paradkar [139]		■	conformance	predicate coverage	WSDL-S	EFSM	—
				mutation-based			
				test purpose			
Lallali et al. [88], Lallali [87]		★	conformance	timed trace	BPEL	TEFSM	BPEL2IF,
				equivalence			TestGen-IF
				test purpose			
Cao et al. [39]		■	conformance	trace equivalence	BPEL	TEFSM	TGSE
				test purpose	WSDL		
Morales et al. [115]		■	conformance	invariant	BPEL	TEFSM	TIPS
				properties			
				test purpose			
Bertolino et al. [30]		■	conformance	IOCO-	PSM	STS	—
				relation	UML		
Frantzen et al. [56]		■	conformance	SIOCO	UML	STS	Jambition
Escobedo Del Cid [50]		■	conformance	IOCO	—	STS,LTS	Tool prototype
			in context	test purpose			
Our approach	Model-based	■	conformance	trace equivalence	ABPEL	STS	a tool chain
				test purpose	WSDL		

Table 2.5: Related work on Web Services Testing - Suite

CHAPTER 3

A Symbolic Approach for Composite Web Service Conformance Testing

In this chapter, we present the theoretical aspects of our end to end symbolic testing approach of a composite service. We clearly explain our steps, which start with the definition of a formal model of the specification, namely the Web Service Symbolic Transition System (WS-STS), by means of our transformation rules written in a process algebraic style.

We also provide the possibility to use test purposes (TP). A TP allows us to focus on a particular behavior to test. By computing a product between the specification and the TP models, we obtain a WS-STS model that supports both the specification behavior and the behavior to test. The product is computed according to our defined rules. Using Symbolic Execution to avoid the unfolding of the model that would yield state space explosion, we generate the Symbolic Execution Tree (SET). Each path of the SET is a potential test case. We present an (online) algorithm that allows us to interact with the Z3 SMT solver in order to realize and then execute the test cases against the implementation service. Finally, depending on the response returned by the service under test, a verdict is emitted indicating whether the service is correct or not with regard to its specification.

3.1 The Proposed Framework

In this section, we present an overview of our formal framework for the testing of service orchestrations.

While most model-based (for verification or testing) approaches targeted at orchestration languages either (i) ignore the retrieval of the formal model from the orchestration specification, (ii) ignore or over-simplify the rich XML-based data types of services, or (iii) do not tackle the execution of test cases against a running service implementation, our approach follows a language-to-language (or end to end) model-based testing framework that tackles the conformance testing process for a service orchestration from the specification language until the execution of the test cases.

As described in the Figure 3.1, we start from the orchestration specification, which is translated into a formal model, namely a Symbolic Transition System (STS) [128] rather than on labeled transition systems (LTS) usually used, either directly or indirectly from process algebraic or Petri net descriptions, as BPEL models.
The LTS are known to cause over-approximation or unimplementable test cases (when data are simply abstracted away), and state explosion problems (when message parameters or variables are flattened wrt. their infinite domains). Both are avoided using STS. Besides, we handle the possibility of exploiting Test Purposes (TP) to check a particular behavior. For this goal, a test purpose is modeled with the same formal model as the specification one *i.e.* as an STS model. A product model is computed based on the specification model and the test purpose model to obtain a model that represents the orchestration behavior and handles test objectives.

Then, the Symbolic Execution (SE) technique [84] is applied to compute a Symbolic Execution Tree (SET) from this STS. The SET represents (a finite subset of) the STS execution semantics, while avoiding the usual state-explosion problem in presence of unbounded data types, as used in full-fledged BPEL. This state explosion is avoided using message parameters and variables as symbolic values instead of concrete data. Given some criteria, we generate from the SET a set of execution paths which are finally run by a test oracle against the orchestration implementation and a verdict is produced.

3.2 Composite Web Services specification

As exposed before for service composition, we are interested in testing a WS-BPEL service orchestration. This executable language describes the behavior of services through the interactions of a centralized service with other ones, according to the basis of an orchestration. Recall that WS-BPEL is based on basic activities and structured activities. Among the basic activities we mention the ones that can be used to communicate with other services by messages (invoke , receive, reply),

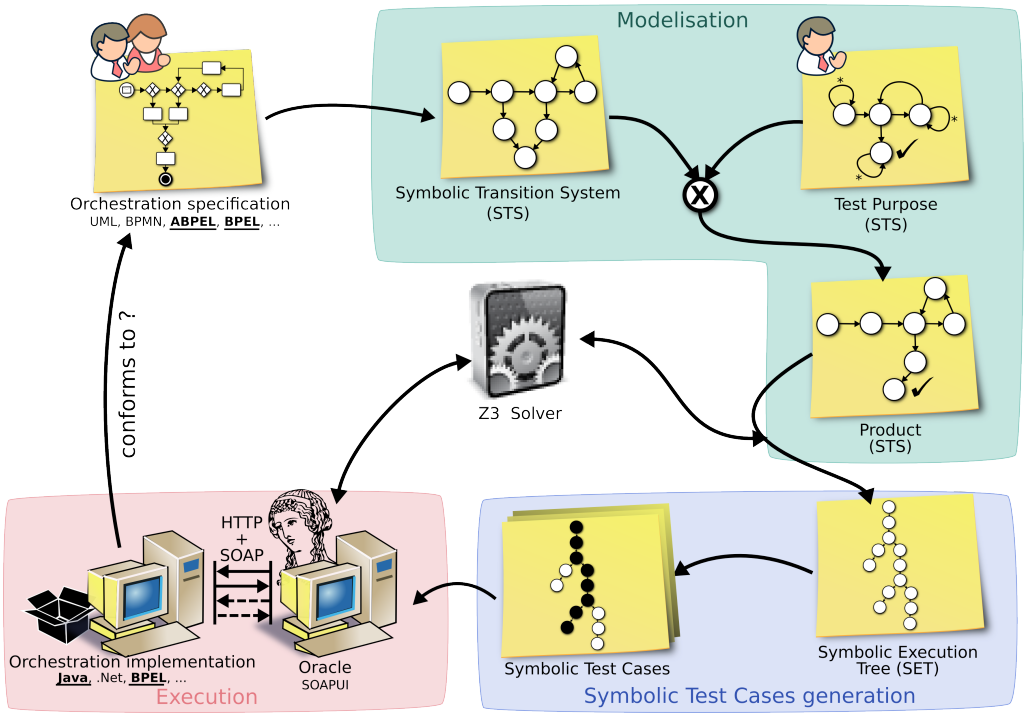

Figure 3.1: Overview of the proposed framework

manipulate data (assign), wait for some time (wait), indicate faults (throw), do nothing (empty), or finish the entire process instance (terminate).
The structured activities can depict a sequential execution (sequence), parallel execution (flow), data-dependent branching (switch), timeout or message-dependent branching (pick), or repeated execution (while). The structured activity scope allows to link activities and provides fault, compensation, and event handling.
However, for a black box testing approach its not allowed to use the source code of WS-BPEL as a specification; and since our concern is on the behavior of an orchestration, not on its execution details, we consider *Abstract BPEL* ((A)BPEL) as our initial specification.

Abstract WS-BPEL as a Specification. (A)BPEL process shares a common syntactic base with WS-BPEL, and it is not necessarily executable. It describes the public interface of the orchestration and thus, it may hide some operational requirements details by either omitting them or replacing them with opaque construct (more details are available on the WS-BPEL manual [117]). (A)BPEL is used for both describing in precise terms the behavior expected form the services partners and expressing the constraints on the roles of each of them. In other words, an Abstract BPEL publicize *what* a composed service do, but not *how* it do it.

3.3 From Language to Model

A formal model of a specification is a graphical representation of it. This representation must describe accurate details on the characteristics that one wants to test or verify. In this section we first provide our formal model named Symbolic Transition System (STS) for a service composition. Then, we present and explain the rules which are used to obtain a service model from a (A)BPEL description.

3.3.1 Service Model

A Web service allows to specify the functionalities that it offers in order to exploit them by other services or human clients. The service may expose information at different interface description levels in order to support automatic discovery, composition, verification and adaptation of Web services.

Two kinds of descriptions could be distinguished. The first one is the *static* description or the interface (*i.e.* WSDL) which focuses on how to use the service and the second one in the *dynamic* description or the conversation (*i.e.* (A)BPEL) which focuses on how the service behaves. For a composition of Web services, such descriptions are more complex due to interactions of several services expressed through the static description, and their organization expressed through the dynamic description. In order to support a precise description of an orchestration with its static and dynamic aspects, and thus avoiding any misunderstanding or ambiguity, we present how these two descriptions are taken into account using the STS formal model.

3.3.1.1 Static Description

The static description provides information details on how to use and communicate with a Web service. Since we are interested in an orchestration of web services, we present the important definitions that describe how a service orchestrator works.

Signatures. The required information needed to interact with a services are provided through the *signature level*.
The *signature level* for a Web service must describe the features that it provides or requests. Among these features we find the used operations. Each operation describes the messages that allow the data exchange for passing operation parameters and getting result(s). A set of domains noted $\mathcal{D}$ can represent the allowed data type specification.
In practice data types are declared in the WSDL file. The later do not propose a new data type definition but, uses XML schema (XSD) to define canonical type system and considers it as the fundamental type system. $\mathcal{D}$ includes simple data types as integers, boolean, etc. However, WDSL also includes complex types. Complex types

represent XML schema data structure. Within the context of Web services complex data types are expressed using XPath expressions.

We suppose that the implicit semantics that we attribute to the data types corresponds to the one used in the BPEL engine, and also used in the SMT solver which we will call later. In other words we suppose domains convey a semantic information so that we can apply evaluation and satisfaction solving. Example: the *int* type for the BPEL engine corresponds to the *Integer* type for the SMT solver.

In order to hold this exchange, the signature must support a set of operations and the required type of these data. Moreover, we associate three functions to the set of operations. To get the input, the output and the fault message of an operation. Another feature of a Web service deals with its set of properties. The properties, together with property aliases and correlation sets, are important BPEL features that support the definition of sessions (see [117] and below, *Message Correlation*). In other words, the correlation represents the link between the message and the corresponding instance of the process.

We also introduce a feature named π that specifies what two-way operations in sub-services are supposed to do. π is a boolean formula relating the inputs and outputs of an operation o, denoted as $\pi(o)$, where $o \in \mathcal{O}$ and $\mathcal{O}$ is a set of operations.

The *signature* of a service corresponds to its description using a combination of XML schema (exchanged data structures) and WSDL (operations and messages). Formally we define signatures as follows:

Definition 3.3.1. ***(Signatures)***
A signature is a tuple $\Sigma = (\mathcal{D}, \mathcal{O}, in, out, err, \pi, \mathcal{P}, \downarrow)$ *where:*

- $\mathcal{D}$ *is a set of domains and* $dom(x)$ *denotes the domain of* x*,*
- $\mathcal{O}$ *is a set of (provided) operations defined within the WSDL file.*
- $in, out, err : \mathcal{O} \rightarrow \mathcal{D} \cup \{\perp\}$ *denote respectively the input, output, or fault message of an operation,*
- π *is used to specify what two-way operations in sub-services are supposed to do:* $\pi(o)$*,* $o \in \mathcal{O}$*, is a boolean formula relating* o *inputs and outputs.*
- $\mathcal{P}$ *is a set of property names,*
- $\downarrow$ *is used to define property aliases for messages: for a message type* m*,* $m \downarrow_p$ *denotes the part in* m *messages that corresponds to property* p*.*

Correlation is a specific property of BPEL business process. It aims to decide which part of the exchanged message between several services, represents the identifier (Id). Thus, each message is mapped correctly to its business process instance. For example, when an employee applies for a loan, his Social Security number may be

regarded as its Id. This Id is then used to correlate messages exchanged during the conversation between the employee and the service orchestrator.

The correlation consists of two parts: (i) a property which has a name and a type. The property refers to parts of the exchanged messages during services conversation. (ii) a correlation set which define a set of properties. The latter represents the complete structure of the Id in our example.

$\perp$ corresponds to an undefined operation message, *e.g.* $out(o) = err(o) = \perp$ for any one-way operation named o. In other words, if o is a one-way operation no reply message (output message) is expected.

Example. The signature associate to the xLoan orchestration service is provide in the Appendix D.1. Among the allowed operation we find :

- {tns:MT-requestIn,tns:MT-requestOut, tns:MT-selectIn, tns:MT-selectOut, tns:MT-cancelIn, xsd:string, xsd:long, xsd:boolen, ns1:userInfo, ns1:loanInfomation, ns1:loanRequest } $\subseteq \mathcal{D}$
- $\{request, select, cancel\} \subseteq \mathcal{O}$,
- $in(resquest)$ = tns:MT-requestIn, $out(request)$ = tns:MT-requestOut, $in(select)$= tns:MT-selectIn, $out(select)$= tns:MT-selectOut, $in(cancel)$=tns:MT-cancelIn,
- suppose a loan application with an amount lower than 10000 then $\pi = MT - approveIn/amount < 10000 \wedge rtr == true$,
- $\mathcal{P}$ = LS-PROP, the name of the property is LS-PROP
- The correlation value corresponds to the fileNumber part for each sent or received message variable as follows: $requestOut \downarrow_{LS-PROP} = ns : fileNumber$, $selectIn \downarrow_{LS-PROP} = ns : fileNumber$, $cancelIn \downarrow_{LS-PROP} = ns : fileNumber$

Partnership. One of the assets characterizing Web services is the fact that they are loosely coupled applications. Thus a service can be used either as a single application or it can be combined with other services. Generally, Web services are not executed in isolation but they communicate with other services or users through a partnership.

A partnership, is a set of partner signatures, corresponding to required and provided operations, including the signature of the service orchestrator itself. The required operations are those needed by the service orchestrator to accomplish his business process. They correspond to the signatures of the invoked Web services. On the other hand, provided operations are the ones exposed by the service orchestrator.

For the need of our approach, we suppose, without loss of generality, that an orchestration has only one of these partners, named USER. The USER will use the

provided signature *i.e.* orchestrator signature, to interact with the service orchestrator. As for the service orchestrator, it will uses the requested signatures to perform his business process and thus, satisfy the USER request. Note that, each service partner including the service orchestrator is recognized using a unique identifier.

Definition 3.3.2. ***(Partnership)***
A partnership ρ *is an* ID*-indexed set of signatures:* $\rho = \{\Sigma_i\}_{i \in ID}$*, where* ID *is a set of names and* $\mathsf{USER} \in ID$*.*

Two domains with the same name, including the namespace part, in two signatures correspond to the same thing. More specifically, we suppose they have the same formal semantics.

Example. The partnership of the xLoan includes the signatures of the USER, the BlackList service and the Bank service.

3.3.1.2 Dynamic Description

The dynamic description depicts the behavior of the orchestration. This description provides details on the order of the interactions between the service orchestrator and the service partners involved in the orchestration.

Events. A stateful Web service or a service orchestrator interacts via conversation protocol. This protocol specifies the order in which the operation calls must occur. The semantics of a service conversation depends on message-based communication, which is modeled using *events* denoted Ev. We can define an event for Web services as an occurrence that take place at a significant point of time. This occurrence could be an action with or without an associated condition, allowing to perform the business process.

We classify these events as: *input events* denoted $pl.o?x$, *output events* denoted $pl.o!x$, waiting for period of time denoted as a χ, or the event indicating an expected termination of a conversation denoted $\surd$. We also discern another kind of event. It corresponds to an internal event of the orchestration service denoted τ. From the viewpoint of a USER partner, such internal events are non-observable, they represent internal computations or conditions in the service orchestrator.
For the input event $pl.o?x$ respectively the output event $pl.o!x$, pl corresponds to the partner link used for the communication with the service, o is the operation used and x is the exchanged message. Note that the operation must be defined by the signature of the corresponding partner link, $o \in \Sigma_{ID_{pl}}$, and the message variable x must correspond to the reception of the input message, $dom(x) = in(o)$, respectively

an output message must correspond to an emission of a message, $dom(x) = out(o)$.

We highlight the fact that in our approach time constraints in services are generally soft hence, discrete time is a valid abstraction.

A service call may also yield errors *i.e.* message faults. This is modeled with fault input events, $pl.o??x$ and fault output events, $pl.o!!x$, where $dom(x) = err(o)$.

We omit BPEL port types here for simplicity reasons, the full event prefixes would be, *e.g.* $pl.pt.o?x$ for the first example above with input on the port type pt.

$Ev^?$ (resp. $Ev^!$, $Ev^{??}$, and $Ev^{!!}$) is the set of input events (resp. output events, fault input events, and fault output events). We note Ex to be the set of internal fault events, that correspond to faults possibly raised internally (not in messages) by the orchestration process.

Definition 3.3.3. ***(Events)***
We define Ev as the set of events where $Ev = Ev^? \cup Ev^! \cup Ev^{??} \cup Ev^{!!} \cup Ex \cup \{\tau, \chi, \surd\}$. We also define hd as $\forall * \in \{?, ??, !, !!\}$, $hd(pl.o * x) = pl.o$, and $hd(e) = e$ for any other e in Ev.

Example. When the xLoan service orchestrator receives a message from the USER service, it is represented as follows:

$$\underbrace{\mathsf{USER}}_{\text{pl}}.\ \underbrace{request}_{\text{operation}}\ \underbrace{?}_{\text{input event}}\ \underbrace{vans2 : requestIn}_{\text{the received variable}}\ \underbrace{/vns2 : requestIn := vans2 : requestIn}_{\text{action associate}}$$

We use a *transition system* to model the orchestration behavior according to the process algebra rules. We first of all, present a simple introduction to *Labeled Transition Systems* (LTS) model. Then we present the model that we use called *Web Service Symbolic Transition System* (WS-STS). The WS-STS is a model very close to the LTS. The difference between the two models is the addition of more details concerning data exchanges in the WS-STS than in the LTS model.

An LTS is a structure connected by transitions. States represent the actual state of the system, and transitions, which are labeled with actions, represent the action that a system may perform. The formal presentation of the LTS as defined by Tretmans [147] is provided in the following:

Definition 3.3.4. ***(Labeled Transition Systems)***
A Labeled Transition Systems *(LTS), is a tuple $(\mathcal{S}, \mathcal{L}, \mathcal{T}, s_0)$, where:*

- *$\mathcal{S}$ is a finite and non empty set of states;*
- *$\mathcal{L}$ is a finite set of labels;*

- $\mathcal{T} \subseteq \mathcal{S} \times (\mathcal{L} \cup \{\tau\}) \times \mathcal{S}$, *with* $\tau \notin \mathcal{L}$, *is the transition relation;*
- $s_0 \in \mathcal{S}$ *is the initial state.*

The idea associated with such representation is that the system evolves from a state to another by executing an action. The representation $s \xrightarrow{l} s'$ *i.e.* $(s, l, s') \in \mathcal{T}$ describes a transition labeled l from the state s to state s'.

We rely on the Mateescu and Rampacek [108] transformation, that defines a formal semantic of BPEL. To represent the dynamic description of an orchestration the authors use a *discrete-time Labeled Transition Systems* (dtLTS) model obtained from their defined Algebra of Timed Processes (ATP) rules.

This formalism had a good coverage of the main BPEL language constructs. Its process algebraic style for transformation rules enables a concise yet precise and operational model, which is, through extension, amenable to symbolic execution.

With reference to the work of Mateescu and Rampacek [108], we reuse, extend and add some of their transformation rules, in order to support data (in computation, conditions and messages), message faults (enabling a service to inform its partners about interval errors), message correlation (enabling BPEL engines to correlate messages in-between service instances), flows (parallel processing) and the until activity. More specifically, support for data yields grounding on (discrete time) Symbolic Transitions Systems and their symbolic execution, rather than on (discrete time) Labeled Transition Systems (dtLTS).

We need to reuse the information describing data structure and the interface of the service orchestrator, *i.e.* the signature level of the service orchestrator, in order to incorporate them in our WS-STS model. After, we apply the transformation rules on an ABPEL specification we obtain the Web Service Symbolic Transition System (WS-STS) model. We present the features of the WS-STS as follows:

Definition 3.3.5. ***(Web Service Symbolic Transition System)***
A Web Service Symbolic Transition System *(WS-STS), is a tuple* $(\mathcal{D}, \mathcal{V}, S, s_0, T)$, *where:*

- $\mathcal{D}$ *is a set of domains,*
- $\mathcal{V}$ *is a set of variables with domain in* $\mathcal{D}$,
- S *is a non empty set of states,*
- $s_0 \in S$ *is the initial state,*
- T *is a (potentially nondeterministic) transition relation,*
  $T \subseteq S \times \mathcal{T}_{\mathrm{Bool},\mathcal{V}} \times Ev \times seq(Act) \times S$, *with:*

- $\mathcal{T}_{\text{Bool},\mathcal{V}}$ *denoting boolean terms possibly with variables in* $\mathcal{V}$*, it represents a guard i.e. the condition that must be satisfied,*
- *Ev a set of events represents the messages communication and internal events.* $\mathcal{D}$ *and* $\mathcal{V}$ *are often omitted in when clear from the context (e.g.* $\mathcal{V}$ *are variables used in transitions).*
- *seq(Act) is a sequence of actions denoting computation (data processing) that will be executed in a sequential way (of the form* $v := t$ *where* $v \in \mathcal{V}$ *is a variable and* $t \in \mathcal{T}_{\mathcal{D},\mathcal{V}}$ *is a term).*

The transition system is called symbolic as the guards, events, and actions may contain variables. $(s, g, e, A, s') \in T$ is also written $s \xrightarrow{[g] \quad e \quad / \ A}_T s'$ or simply $s \xrightarrow{[g] \quad e \quad / \ A} s'$ when clear from the context.

The WS-STS describes the behavior of an orchestration. Thus, the progress from the state s to the state s' is done when the event e occurs, the guard g is satisfied and some actions are performed. In case, there is no guard (*i.e.* it is true) associate to the transition, it is omitted. The same yields for the actions.

From the above definitions, we can represent an orchestration as a WS-STS model representing the behavior of a service orchestrator and a set of partner services:

Definition 3.3.6. ***(Orchestration)***
An orchestration $\mathcal{O}rch$ *is a couple* $(\rho, \mathcal{B})$ *where:*

- ρ *is a partnership,*
- $\mathcal{B}$ *is a WS-STS.*

We impose that $\mathcal{B}$ is correct wrt. ρ, *i.e.* its set of events correspond to partner links and operations defined in ρ, which can be syntactically checked.

Notice that each data type (structured or scalar) of BPEL corresponds to an element of $\mathcal{D}$ and each variable of BPEL corresponds to a variable in $\mathcal{V}$. STS have been introduced under different forms (and names) in the literature [128], to associate a behavior with a specification of data types that is used to evaluate guards, actions and sent values. Transformation rules from BPEL to WS-STS are provided in the following subsection.

3.3.2 (A)BPEL to STS Transformation Rules

In this subsection we present a syntactical abstraction of the BPEL activities as a Backus Normal Form (BNF) that represent simple (or basic) and structured activities, then we present the associate transformation rules.

The BNF form that describes the main BPEL activities [117], is presented as follows:

P,Q,R	::=	basic | struct
basic	::=	receive(pl,op,var) | reply(pl,op,var) | invoke(pl,op,inputvar[,outputvar]) | time | throw e | x[/path]:=Expr | empty | 0
struct	::=	P;Q | if c then P[else Q] | while c {P} | repeat {P} until c flow($\{P_i\}$) | scope(P,EH^d) | pick(EH^d)
EH^d	::=	[{((pl_i,op_i,var_i),P_i)},(d,Q),{(e_j,R_j)}]

Here, P, Q or R represent BPEL activities that could be either basic or structured activities. As basic activities we can find:

- Communication activities (receive, reply, and invoke), such as the attributes **pl**, **op** and **var** for respectively partnerLink, operation and variables are present in the receive and reply activities.
  Instead of the variable attribute, the invoke activity uses an input variable and the optional (expressed using the square brackets) output variable. Remember that an invocation can be execute as a one-way operation that why the output variable is optional
- Time activities (timeouts or watchdogs) can be reduced to a time passing activity, time, and the use of scopes
- Faults are raised using throw
- Assignment activities (:=), support data and computation, and operate between an XPath [156] expression (a variable and an optional path over it) and any expression (including a simple use of XPath)
- empty and 0 denote respectively an empty and a terminated process.

On top of these *basic activities*, BPEL defines workflow-based *structuring activities*:

- Sequence (;), where P;Q describes here that the P activity followed by the Q activity
- Conditional using (if) to express if the constraint c is satisfied then execute P else execute Q.
- Loops:
  - The while loop verify the constraint c then executes P as long as the c is satisfied

  - However until execute at least once the activity P then verifies the satisfaction of the condition c

- The flow describes the parallel execution of several P activities (P_1, P_2,...)

- scope(P,EH^d) encapsulates an activity P with an event handler EH^d. An event handler is made up of sets of events, one for each type of event and associated activities. The event could be associate to a reception, a time out our faults, where received messages ((pl_i,op_i,var_i),P_i), timeouts related to a duration (d), or faults (e_j). These correspond respectively in BPEL to onEvent and onAlarm in event handlers, and to fault handlers.
  scope behaves as P if none of the events happens and as a given sub-activity (some P_i, Q, or R_j) if the corresponding event happens.

- pick is treated as a scope. We abstract from concrete syntax differences, *e.g.* between onMessage in a pick and onEvent in scopes

Transforming (A)BPEL into a state transition could be done directly with structural BPEL to the state transition rules as in M. Lallali thesis [87].
Let us illustrate this with an example. Consider the sequence activity: P;Q where the activity P must be followed by Q. The associate model will be as presented in the Figure 3.2, where P-automata and Q-automata are the transformation of the activities P and Q respectively.

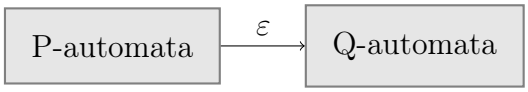

Figure 3.2: Transformation of P;Q sequence into an automata according to [87]

Such a kind of transformation becomes complex when we deal with structured activities. Here an example: let us consider the flow activity. Its transformation involves the product computation of all the automaton (activities) implied within the flow process. Such computation quickly becomes difficult.

In order to avoid such complexity, we prefer the use of process algebra (PA) semantics style that allows more expressiveness.
Let us back to the P;Q sequence example. We have to consider the case were P is a structured activities, then P has to perform its nested activities before executing Q. Using process algebra semantic, we express such case using rule hypothesis and rule conclusion where:

$$\forall a \in Ev \backslash \{\surd\},\ \frac{P \xrightarrow{[g]\ \ a\ \ /\ A} P'}{P;Q \xrightarrow{[g]\ \ a\ \ /\ A} P';Q} \quad \begin{matrix} \} \text{ Rule hypothesis} \\ \} \text{ Rule conclusion} \end{matrix}$$

When P activity ends its execution, or P is a basic activity, it is represented using the $\surd$, then Q activity must be processed. The representation of this case is:

$$\forall a \in Ev,\ \frac{P \xrightarrow{\surd} P' \wedge Q \xrightarrow{[g]\ \ a\ \ /\ A} Q'}{P;Q \xrightarrow{[g]\ \ a\ \ /\ A} Q'} \quad \begin{matrix} \} \text{ Rule hypothesis} \\ \} \text{ Rule conclusion} \end{matrix}$$

Our transformation rules (BPEL to STS) are presented in the Table 3.3 for basic activities. The Tables 3.5 and 3.6 contain transformation rules for structured ones. Note that the $^*/^+$ symbols represent extended / added rules from Mateescu and Rampacek [108] (BPEL to dtLTS) transformation.

Basic activities	
BPEL	STS
empty	$\mathsf{empty} \xrightarrow{\surd} 0$
time	$\mathsf{p} \xrightarrow{\chi} \mathsf{p}$ with $\mathsf{p} \in \{\mathsf{time}, \mathsf{rec(pl,o,v_{in})}, \mathsf{send(pl,o,v_{out})}\}$
assign$^+$	$\mathsf{p1{:=}p2} \xrightarrow{\tau\ \ /\ \ p1:=p2} \mathsf{empty}$
throw	$\forall e \in Ex$ throw $\mathsf{e} \xrightarrow{e} 0$
rec$^+$	$\mathsf{rec(pl,o,v_{in})} \xrightarrow{pl.o?va_m\ \ /\ \ v_{in}:=va_m} \mathsf{empty}$ with $\exists o \in \mathcal{O}(\Sigma_{pl}),\ in(o) = m$
send$^+$	$\mathsf{send(pl,o,v_{out})} \xrightarrow{\tau\ \ /\ \ va_m:=v_{out}} \_ \xrightarrow{pl.o!va_m} \mathsf{empty}$ with $\exists o \in \mathcal{O}(\Sigma_{pl}),\ out(o) = m$
receive*	$\mathsf{receive(pl,o,v_{in})} = \mathsf{rec(pl,o,v_{in})}$
reply*	$\mathsf{reply(pl,o,v_{out})} = \mathsf{send(pl,o,v_{out})}$
invoke$^+$	$\mathsf{invoke(pl,o,v_{in})} = \mathsf{send(pl,o,v_{in})}$ $\mathsf{invoke(pl,o,v_{in},v_{out})} = \mathsf{send(pl,o,v_{in});rec(pl,o,v_{out})}$

Figure 3.3: Transformation rules for basic activities

The transformation rules describe how the BPEL activities are represented in an WS-STS model. In the following we explain transformation rules presented in the Table 3.3.

The empty activity semantics is to do nothing, its execution in the WS-STS will be represented as an event that terminates correctly. It implies that the out transition associate to an empty event is labeled with a $\surd$ (tick).

Time activity is transformed to represent a wait for a period of time denoted χ. Meanwhile, the behavior of the orchestrator service does not change. In our model, we treat time passing as discrete time.

Data manipulation are managed through the assign activity. This activity aims at associating a value to a variable. It is represented as an internal event denoted τ with the description of its associate action, *i.e.* $p1 := p2$.

An exception is raised in BPEL with the throw activity, such event is represented in the WS-STS as a transition labeled with the name of the exception.

The transformation rules for receive and reply communication activities, were established from the following reasoning. A message is received through the partner link pl according to the operation o that manipulates the anonymous message variable va_m. The anonymous message variables are used to represent data exchange between the orchestrator and partner services. The action associated with this input event (receive activity) allows to attribute the incoming variable to the corresponding orchestration variable.
The output event (reply activity) is executed in two steps, first an internal action (τ) assigns the data of the orchestrator to the anonymous variable of the corresponding partner. Then the message is sent according to the operation and the partner link of the service.

We impose that variables used in the WS-STS transitions are variables from ABPEL and we add anonymous variables used for interacting with other services. As said before the anonymous message variables are used as intermediate variables for data exchange between the service orchestrator and the other partner services according to BPEL communication semantics.

Example. Consider a simple service which receives a variable increases it with 1 then returns the result. The model associate to this scenario is represented in the Figure 3.4. This model describes the reception of the *anonymous* message variable x_a according to the operation named *addition*. The variable x_a is copied into the orchestration variable x, in order to be processed, then the result is returned via the anonymous message variable x_a.

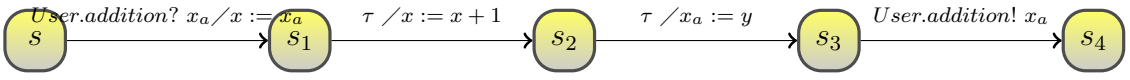

Figure 3.4: Simple service with an anonymous message variable

The Tables 3.5 and 3.6 describe transformation rules for structured activities.

Intuitive rules were defined for the if, while and until activities. For these activities, conditional branches are considered as internal events. If the associated condition (c) is satisfied then business process will execute the correspondent activities else it will preform the activities of the other branch.

Structured activities	
BPEL	STS
sequence*	$\forall a \in Ev \backslash \{\surd\}$, $\frac{P \xrightarrow{[g] \quad a \quad / \quad A} P'}{P;Q \xrightarrow{[g] \quad a \quad / \quad A} P';Q}$ $\forall a \in Ev$, $\frac{P \xrightarrow{\surd} P' \wedge Q \xrightarrow{[g] \quad a \quad / \quad A} Q'}{P;Q \xrightarrow{[g] \quad a \quad / \quad A} Q'}$
if*	if c then P else Q $\xrightarrow{[c] \quad \tau}$ P if c then P else Q $\xrightarrow{[\neg c] \quad \tau}$ Q
while*	while c {P} $\xrightarrow{[c] \quad \tau}$ P;while c {P} while c {P} $\xrightarrow{[\neg c] \quad \tau}$ empty
until+	repeat {P} until c = P;while c {P}
flow+	
flow internals	$\forall a \in Ev \backslash \{\chi, \surd\}$, $\frac{\exists j \in I, P_j \xrightarrow{[g] \quad a \quad / \quad A} P'_j}{\text{flow}(\{P_{i,i \in I}\}) \xrightarrow{[g] \quad a \quad / \quad A} \text{flow}(\{P_{i,i \in I \backslash \{j\}}\} \cup \{P'_j\})}$
flow termination	$\frac{\forall i \in I, P_i \xrightarrow{\surd} P'_i}{\text{flow}(\{P_{i,i \in I}\}) \xrightarrow{\surd} \text{empty}}$
time passing	$\frac{\exists J \neq \emptyset, J \subseteq I, \forall i \in J, P_i \xrightarrow{\chi} P'_i \wedge \forall i \in I \backslash J, P_i \xrightarrow{\surd} P' i}{\text{flow}(\{P_{i,i \in I}\}) \xrightarrow{\chi} \text{flow}(\{P_{i,i \in I \backslash J}\} \cup \{P'_{i,i \in J}\})}$

Figure 3.5: Transformation rules for structured activities

The flow activity, supports the concurrent execution aspect of a business process. All the activities included in the flow are executed simultaneously. The execution of the flow activity implies the union of the execution of all sub activities. We can distinguish 3 kinds of flow activity depending on event that will occur.
The *internals flow* transformation rule, describes the execution of the simultaneous events, except for the termination $\surd$ and the time passing χ events, each event will be executed one after the other in the WS-STS. When the flow execution ends with a *flow termination* ($\surd$), it leads him to an empty activity. Finally, if a *time passing* is defined in the flow, it corresponds to a wait for an amount of time before continuing its execution.

Structured activities - suite

BPEL	STS
scope*	let $EH^d = [\{((pl_i, o_i, v_i), P_i)_{i\in I}\}, (d, Q), \{(e_j, R_j)_{j\in J}\}]$, $O_I = \{(pl_i, o_i, v_i)_{i\in I}\}$, $\overline{O_I} = \{pl_i.o_i \mid (pl_i, o_i, v_i) \in O_I\}$, $E_J = \{e_{j,j\in J}\}$ in:
event handler	$\forall (pl_i, o_i, v_i) \in O_I$, $\dfrac{\forall a \in Ex \cup \{\chi, \surd\}, \neg(P \xrightarrow{a})}{\mathsf{scope}(P, EH^d) \xrightarrow{pl_i.o_i?vam \;/\; v_i := vam} P_i}$ with $\exists o_i \in \mathcal{O}(\Sigma_{pl_i}),\ in(o_i) = m$
time passing	$\forall d > 1$, $\dfrac{P \xrightarrow{\chi} P' \wedge \forall a \in Ex \cup \{\tau, \surd\}, \neg(P \xrightarrow{a})}{\mathsf{scope}(P, EH^d) \xrightarrow{\chi} \mathsf{scope}(P, EH^{d-1})}$
alarm	$\dfrac{P \xrightarrow{\chi} P' \wedge \forall a \in Ex \cup \{\tau, \surd\}, \neg(P \xrightarrow{a})}{\mathsf{scope}(P, EH^1) \xrightarrow{\chi} \mathsf{Q}}$
fault handler	$\forall e_j \in E_J$, $\dfrac{P \xrightarrow{e_j}}{\mathsf{scope}(P, EH^d) \xrightarrow{\tau} R_j}$
unsupported fault	$\forall e \in Ex \backslash E_J$, $\dfrac{P \xrightarrow{e}}{\mathsf{scope}(P, EH^d) \xrightarrow{e} \mathsf{0}}$
scope termination	$\dfrac{P \xrightarrow{\surd}}{\mathsf{scope}(P, EH^d) \xrightarrow{\surd} \mathsf{0}}$
scope internals	$\forall a \in Ev$, $\dfrac{hd(a) \notin (\{\chi, \surd\} \cup Ex \cup \overline{O_I}) \wedge P \xrightarrow{[g] \;\; a \;\; / \;\; A} P'}{\mathsf{scope}(P, EH^d) \xrightarrow{[g] \;\; a \;\; / \;\; A} \mathsf{scope}(P', EH^d)}$
pick	pick(E) = scope(time,E)

Figure 3.6: Transformation rules for structured activities - suite

The scope(P,EH^d) (see Table3.6) encapsulates an activity P with an event handler EH^d. An event handler is made up of sets of events, one for each type of event –received messages, timeouts related to a duration (d), or faults (e_j). These correspond respectively in BPEL to onEvent and onAlarm in event handlers, and to fault handlers. – and associated activities. scope behaves as P if none of the events happens and as a given sub-activity (some P_i, Q, or R_j) if the corresponding event happens.

Finally, the pick activity is treated as a scope activity. We abstract from concrete syntax differences, *e.g.* between onMessage in a pick and onEvent in scopes.

Other important features of composite Web services are the correlation and the message faults. In the following we present how these two characteristics are

represented within the formal model.
Transformation for message correlation. We support correlation as an extension of the transformation rules Tables. Firstly, we modify the orchestration model to be $(\rho, \mathcal{B}, \mathcal{C})$ where $\mathcal{C}$ are correlation sets, *i.e.* a name and a set of associated properties denoted with *props*. These are used to correlate messages in-between service instances [117].

Sometimes a single property is used (*e.g.* an identifier), but more generally this is a set (*e.g.* name and surname). A correlation value is a value of a structured domain with items corresponding to the properties. For each correlation set c in $\mathcal{C}$, we have two variables, the correlation value vcs_c and the correlation initiation $vcs_c init$, among the set of variables belonging to the model $\mathcal{B}$ ($\mathcal{V}(\mathcal{B})$). The communication activities parameter lists (pl,o,v) (in receive, reply, invoke, pick using onMessage and scope using onEvent) are extended to (pl,o,v,i,c) where c is the correlation name and i corresponds to correlation initiation (yes, no, or join). The STS semantics of communication activities is then extended in the following way:

On the initial transition we add action $vcs_c init := false$. It means that the correlation variable ($vcs_c init$) is not initialized yet.
We also define the *correlation consistency constraint* (ccc) as a function that verifies the property p of a received message va_m with regards to it correlation set c.
Note that if the correlation set includes multiple properties all those properties must be verified. Since the input message is received from an other service the associate variable is an anonymous one. The formalization of this idea is expressed as follows:

$$ccc(c, va_m) = (\bigwedge_{p \in props(c)} va_m/[m{\downarrow}p] = vcs_c/p).$$

Among the attributes of a correlation we find an attribute named *initiate*. As said earlier, only three possible values may be assigned to this attribute a *yes, no* or *join*.
The *yes* implies that the property of the first message that initiates the conversation is the one that should be taken as the correlation of the process. The *no* implies that the correlation is already defined. In such a case, if the value of the received message property is no the same as the defined correlation then a fault is raised. The last option is the *join* value. It implies that if a correlation set is not defined yet, then it will corresponds to the property of the received message.

In our formal model (WS-STS) we represent the violation of these attribute with a guard G. Consequently, we have three guards G^{yes}, G^{no}, and G^{join} that check the correlation c according to the received message supporting the anonymous message variable va_m. Formally, we express the above guards as follows:

- $G_c^{\text{yes}}(va_m) = (vcs_c init = true)$, the correlation guard with a yes attribute is

triggered when the instantiation variable had a true value instead of a false,

- $G_c^{\text{no}}(va_m) = (vcs_c init = false \vee \neg ccc(c, va_m))$, this guard implies that the $vcs_c init$ variable had not a defined correlation yet, which should not be the case because the *no* indicates that the process had already a defined correlation, or that the correlation consistency constraint is not satisfied

- $G_c^{\text{join}}(va_m) = (vcs_c init = true \wedge \neg ccc(c, va_m))$, the join guard is triggered when the correlation is already defined however, the correlation consistency constraint with the received message is not satisfied

When the correlation property is satisfied, it implies the execution of the some actions associate to their corresponding attribute, where:

- $A_c^{\text{yes}}(va_m) = A_c^{\text{join}}(va_m) = \{vcs_c/p := va_m/[m \downarrow p]\}_{p \in props(c)} \cup \{vcs_c init := true\}$, when a correlation must be defined, *i.e.* yes attribute, or need to be defined in case it is not done yet, *i.e.* join attribute, then (i) the correlation variable vcs_c will have the same value as the properties of the received message va_m, and (ii) the variable $vcs_c init$ will be set to true, thus indicating that the correlation had been defined

- $A_c^{\text{no}}(va_m) = \emptyset$, when a correlation do not have to be defined, then no action will be executed.

From now on, in order to take into consideration the correlation, we replace each transition $s \xrightarrow{pl.o*va_m \quad / \ A} s'$ $(* \in \{?, ??\})$ by:

$s \xrightarrow{pl.o*va_m \quad / \ A} s''$, the reception of a message or an exception,
$s'' \xrightarrow{[\neg G_c^{\text{i}}(va_m)] \quad \tau \quad / \ A_c^{\text{i}}(va_m)} s'$, when the correlation is ok, then we execute the associate action
and
$s'' \xrightarrow{[G_c^{\text{i}}(va_m)] \quad \tau}$ throw bpel:correlationViolation, if the correlation guard is satisfied, it implies a correlation error and thus a bpel exception must be thrown.

We also replace each transitions $s \xrightarrow{\tau \quad / \ A} s' \xrightarrow{pl.o*va_m \quad / \ A} s''$ $(* \in \{!, !!\})$ by:

$s \xrightarrow{\tau \quad / \ A} s'$, assignment activity, where we allocate a variable value to the anonymous variable to be send,
$s' \xrightarrow{[\neg G_c^{\text{i}}(va_m)] \quad pl.o*va_m \quad / \ A \cup A_c^{\text{i}}(va_m)} s''$, the correlation is ok, then we emit the message after executing the transition actions and the actions of the correlation

$s' \xrightarrow{[G^{\mathrm{i}}_{\mathrm{c}}(va_m)]\ \ \tau}$ throw bpel:correlationViolation, in this case the correlation guard is satisfied, indicating a correlation error and thus a bpel exception must be thrown.

Transformation for message faults. Faults may occur during the execution of a business process. A fault indicates an abnormal behavior of the process. However, we can distinguish three kind of faults:

- a fault raised during an interaction with a service partner, such a fault is known as a WSDL faults,
- a fault raised by the BPEL engine due to an unsatisfied action, such as specifying an incorrect XPath expression,
- a fault explicitly defined in the business process using the throw activity.

Faults are generally managed within the BPEL process. Nevertheless, it is possible to communicate these errors to the client through the reply activity and message faults.

As for correlation sets, we support messages faults with the extension of the rules presented in the transformation Tables. The concerned activities are the reply and the invoke.
For reply, we add the reply(pl,o,fn[,$\mathsf{v}_{\mathrm{err}}$]) form, where fn is the fault name and $\mathsf{v}_{\mathrm{err}}$ is an (optional) fault variable. This form is transformed as follows:

$$\mathsf{reply(pl,o,fn[,v_{err}])} = \mathsf{send(pl,o,fn[,v_{err}])}, \text{ with}$$
$$\mathsf{send(pl,o,fn[,v_{err}])} \xrightarrow{\tau \ / \ v_{\mathrm{m}}:=v_{\mathrm{err}}} \_ \xrightarrow{pl.o!!_{fn}v_{\mathrm{m}}} \mathsf{empty} \text{ with } \exists o \in \mathcal{O}(\Sigma_{pl}),\ err(o) = m.$$

Faults for synchronous invoke are represented by first sending the message to the corresponding service partner then the reception of the answer. The synchronous invoke is interpreted as follows:
invoke(pl,o,v_{in},$\mathsf{v}_{\mathrm{out}}$)= send(pl,o,$\mathsf{v}_{\mathrm{in}}$);rec+(pl,o,$\mathsf{v}_{\mathrm{out}}$), where there are two rules for rec+: one for the reception of a correct message, rec+(pl,o,v_{in}) = rec(pl,o,v_{in}), and one for the reception of a fault message. The transformation rule is given as follows:

$$\mathsf{rec+(pl,o,v_{in})} \xrightarrow{pl.o??_{\mathrm{fn}}v_{\mathrm{m}} \ \ / \ \ v_{\mathrm{in}}:=v_{\mathrm{m}}} \mathsf{throw\ fn} \text{ with } \exists o \in \mathcal{O}(\Sigma_{pl}),\ err(o) = m.$$

The catch construct of synchronous invoke (as in Figure 3.7) is not directly supported but it can be simulated using a fault handler in a scope around the invoke.

Application on the xLoan case study: Using the transformation rules on the xLoan orchestration we obtain the formal model depicted in Figure 3.8, where tau (resp. tick, term) denote τ (resp. χ, $\surd$). The zoom corresponds to the while part.

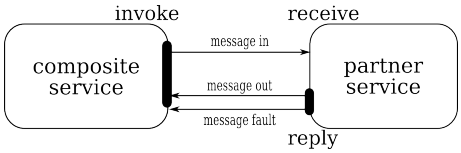

Figure 3.7: Catch construct of invoke activity

One may notice states 16 (while condition test), 17/33 (pick), 34 (onAlarm timeout), and 18/23 (correlation testing with the attribute *no*)

3.3.3 Test Purpose Model

In order to generate test cases for a service orchestrator, one may want to focus on a particular aspect of the process behavior. A Test Purpose (TP) is a set of functional properties allowed by the specification and that one is interested to test. Generally, the formal model for TP follows the one used for the system specification. Hence, LTS is the most popular model for TP. In our case, since STS are our formal model. TP will be formalized as an STS.
Note that a TP could also be modeled using Linear Temporal Logic (LTL) to be more abstract. The average user may prefer more user friendly notation *e.g.* MSC or UML sequence diagrams [153, 175] that describe the interactions between system components. In both case we can get back to transition system model: LTL can be transformed in Buchï automata [64] while, MSC and UML sequence diagrams can be transformed in LTS [125].

To formally represent requirements as a test purpose we were inspired by the work of Jéron et al. [77]. However, the way to express a test purpose is simpler because we don't need *reject* states to specify an undesired behavior. Thus, the WS-STS resulting from the product of the specification model $\mathcal{B}$ and the TP model, contains only the paths that run through an accept state.
In our context TP models are defined according to the orchestration (specification) models they refer to.

Definition 3.3.7. ***(Test Purpose)***
Given an orchestration model $\mathcal{B} = (\mathcal{D}_\mathcal{B}, \mathcal{V}_\mathcal{B}, S_\mathcal{B}, s_{0_\mathcal{B}}, T_\mathcal{B})$*, a TP for* $\mathcal{B}$ *is a WS-STS* $TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$.

TP may use a set of additional variables $\mathcal{V}_I$ for expressiveness, disjoint from $\mathcal{B}$ variables, *i.e.* using a variable in the TP model to limit the number of iterations in a loop described by the specification model. $\mathcal{V}_{TP} = \mathcal{V}_I \cup \mathcal{V}_\mathcal{B}$ where $\mathcal{V}_I \cap \mathcal{V}_\mathcal{B} = \emptyset$, accordingly $\mathcal{D}_{TP} \supseteq \mathcal{D}_\mathcal{B}$, with $\forall t\ s \xrightarrow{[g] \quad e \quad / \ v:=t} s' \in T_{\mathcal{TP}}\ v \in \mathcal{V}_I$. Assignments in

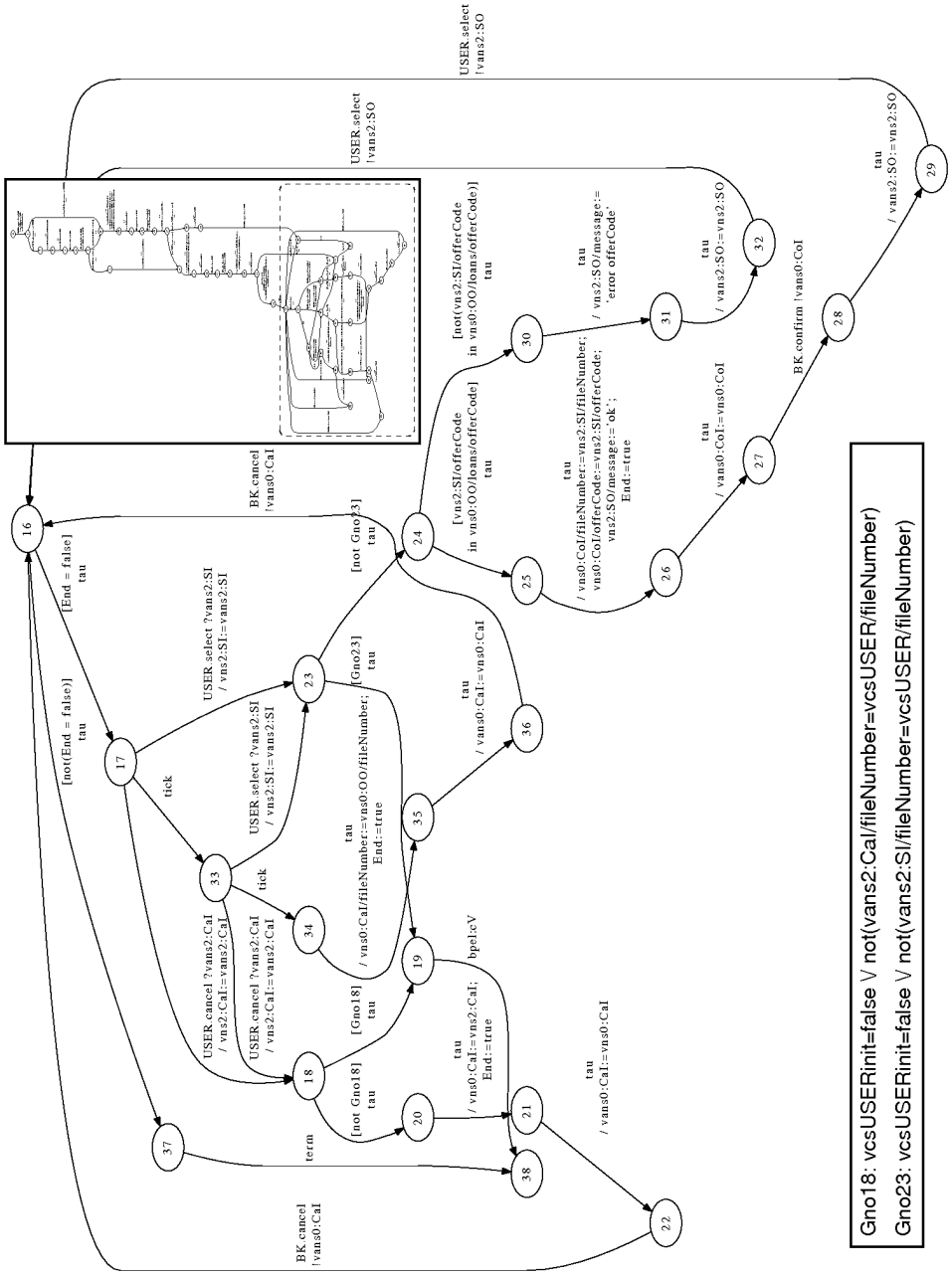

Figure 3.8: The formal model of the xLoan orchestration

TP can only operate on $\mathcal{V}_I$.

The events labeling in the TP transitions correspond to the $\mathcal{B}$ ones. More specifically, we impose for simplicity sake that variables used in message exchanges

(events of the form pl.op ...) correspond to the ones in $\mathcal{B}$. This constraint can be lifted using substitutions.
TP also introduce a specific event, *. Transitions labeled with * may neither have a guard, nor actions, and are used to abstract in TP one or several $\mathcal{B}$ transitions that are not relevant for the expression of the requirement.

Example: The Figure 3.9 is a simplified example of using the * transition within a specific TP. The focus of the test purpose is on the input and the output transitions, all the other activities that may occur in between these two transitions are abstracted using the *.

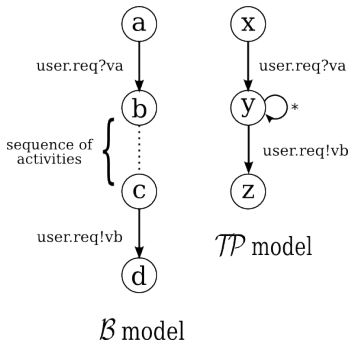

Figure 3.9: A test purpose with a * transition

A TP defines a specific set of states, *Accept* ($Accept \subseteq S_{TP}$), that denotes TP satisfaction. In order to handle acceptance states, while we compute the WS-STS product, we add transitions labeled by an event # in the TP model leading to the accept states. Finally, we impose that TP is consistent with $\mathcal{B}$, *i.e.* TP symbolic traces are included in $\mathcal{B}$ ones. This can be checked using symbolic execution (see Sect. 3.4), where we also have to check that the path condition corresponding for the TP trace implies the path condition of the $\mathcal{B}$ trace.

Application on the xLoan case study: We provide the a Test purpose model for the xLoan orchestration that imposes that the USER selects an offer and this offer is proposed by service orchestrator as shown in Figure 3.10.

3.3.4 WS-STSs Product

We reached the point where we have the specification model on one side and the TP model in another. The model of the specification, describes the complete expected behavior of the service. The TP model represents only certain aspects of the behavior, the one to test, the rest is abstracted using * transitions.

In order to, support the scenarios described by the TP during the generation of test cases, we must compute a product between the specification model and the TP

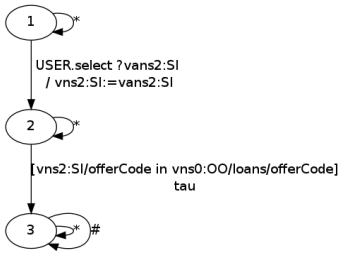

Figure 3.10: A test purpose for the xLoan orchestration

model. Thus, the generated test cases will be dedicated to aspects of the behavior described in TP. The computed WS-STS product is defined as follows:

Definition 3.3.8. ***(WS-STSs Product)***
Given a specification model $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$*, and a test purpose* $TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$*, their product,* $Prod = \mathcal{B} \otimes TP$*, is the WS-STS* $(\mathcal{D}_{Prod}, \mathcal{V}_{Prod}, S_{Prod}, s_{0_{Prod}}, T_{Prod})$ *where:* $\mathcal{D}_{Prod} = \mathcal{D}_{TP}$*,* $\mathcal{V}_{Prod} = \mathcal{V}_{TP}$*,* $S_{Prod} \subseteq S_{\mathcal{B}} \times S_{TP}$*,* $s_{0_{Prod}} = (s_{0_{\mathcal{B}}}, s_{0_{TP}})$*, and* T_{Prod} *is the transition relation.*

Note that each state of the product model consists of a specification model state and another of the TP model. The transition relation of the product (T_{Prod}) is built using four rules. This (see Table in Figure 3.11) rules represent the different scenarios that may happen either in the specification model $\mathcal{B}$ or in the test purpose model TP, during the computation of the product. These rules describe how the computation is done according to each transition.

The two first rules express the independent evolution of a behavior owing to a non observable event. If the TP model of rule (i), as represented in Figure 3.12, (resp. $\mathcal{B}$ model of the rule (ii)) had to perform an intern event (τ), wait of a period of time (χ) or execute an action to achieve an accept state ($\sharp$), then the resulting product model will evolve with the same transition to achieve a new state. This new state consists of the current state of the specification model $\mathcal{B}$ (resp. current state of TP model) and of the new state of the TP model (resp. new state of the specification model $\mathcal{B}$).

Note that it may occur that both the TP and $\mathcal{B}$ models had an internal events then each transition will be handled by combining the rule (i) and (ii) as shown in Figure 3.13.

In the third rule, both the specification model and the TP model evolve with the same event. Such case represents an event synchronization between $\mathcal{B}$ and TP. However, an event synchronization did not implies to have the same constraints.

(i) $\forall e \in \{\tau, \chi, \sharp\}$,

$$\frac{s_{TP} \xrightarrow{[g] \quad e \quad / \quad A}_{TP} s'_{TP}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g] \quad e \quad / \quad A}_{Prod} (s_{\mathcal{B}}, s'_{TP})}$$

(ii) $\forall e \in \{\tau, \chi, \sharp\}$,

$$\frac{s_{\mathcal{B}} \xrightarrow{[g] \quad e \quad / \quad A}_{\mathcal{B}} s'_{\mathcal{B}}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g] \quad \tau \quad / \quad A}_{Prod} (s'_{\mathcal{B}}, s_{TP})}$$

(iii) $\forall e \in Ev^? \cup Ev^! \cup Ex \cup \{\surd\}$,

$$\frac{\begin{array}{c} s_{\mathcal{B}} \xrightarrow{[g_{\mathcal{B}}] \quad e \quad / \quad A_{\mathcal{B}}}_{\mathcal{B}} s'_{\mathcal{B}}, \\ s_{TP} \xrightarrow{[g_{TP}] \quad e \quad / \quad A_{TP}}_{TP} s'_{TP} \end{array}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g_{\mathcal{B}} \wedge g_{TP}] \quad e \quad / \quad A_{TP}; A_{\mathcal{B}}}_{Prod} (s'_{\mathcal{B}}, s'_{TP})}$$

(iv) $\forall e \in Ev^? \cup Ev^! \cup Ex \cup \{\surd\}$,

$$\frac{\begin{array}{l} s_{\mathcal{B}} \xrightarrow{[g_{\mathcal{B}}] \quad e \quad / \quad A_{\mathcal{B}}}_{\mathcal{B}} s'_{\mathcal{B}}, \\ s_{TP} \xrightarrow{*}_{TP} s'_{TP}, \\ \nexists\, s_{TP} \xrightarrow{[g_{TP}] \quad e \quad / \quad A_{TP}}_{TP} s'_{TP} \end{array}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g_{\mathcal{B}}] \quad e \quad / \quad A_{\mathcal{B}}}_{Prod} (s'_{\mathcal{B}}, s'_{TP})}$$

Figure 3.11: Rules of the product computation between $\mathcal{B}$ and TP

Indeed, guard for the product transition will consider the $\mathcal{B}$ guard and TP guard (Figure 3.14). As for the actions, the actions of the product model will execute the TP actions then the $\mathcal{B}$ actions.

Finally, in the last rule (iv), we handle the * semantic. Remember that the * symbol is used in the TP model to represent abstraction of any or the rest of transitions in the specification model. In (iv) rule, the transition of the product model is the same as the one of the $\mathcal{B}$, while the source state from which we compute the product transition, does not satisfy any of the previous rules. In other words, if this state had no output transition with a non observable event, or no possible synchronization event then the transition of the product model is the same as the $\mathcal{B}$ model (Figure 3.15).

To enforce the acceptance states semantics, the product is cleaned up by pruning states (and related transitions) that are not co-reachable from acceptance states, *i.e.* any s such that $\nexists s' = (s'_{\mathcal{B}}, s'_{TP}) \in S_{Prod} \;.\; s \rightarrow *s' \wedge s' \in Accept$.
Notice that a cleaning of the WS-STS product is performed to keep only the paths that pass through an accept state, it is for this reason that we do not need reject

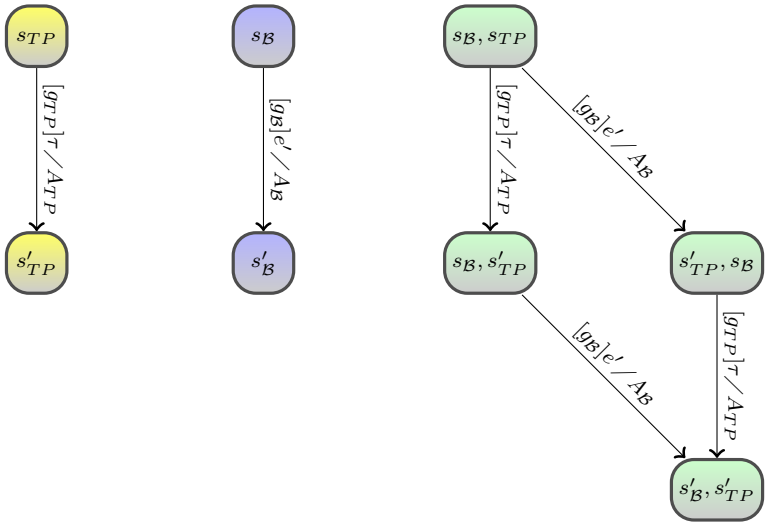

Figure 3.12: Product of the rule (i) where: $e' \notin \{\tau, \chi, \sharp\}$

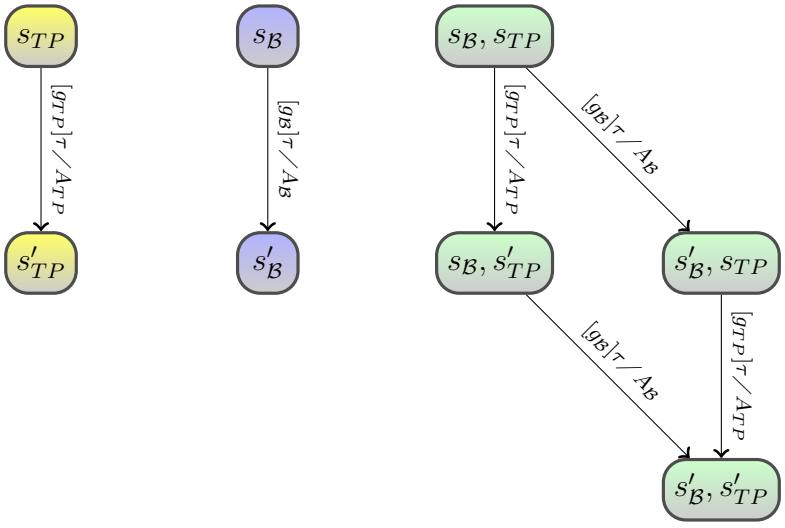

Figure 3.13: Product of the rule $(i$ bis)

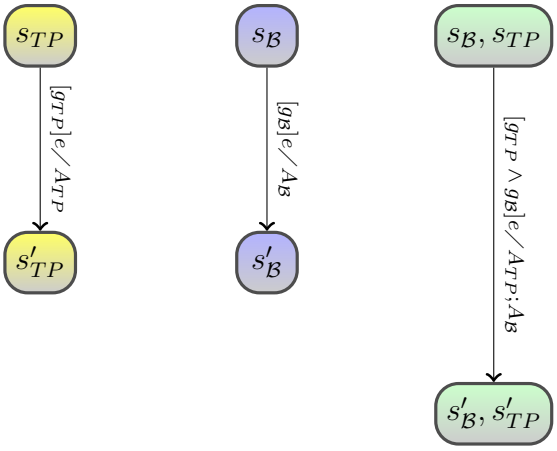

Figure 3.14: Product of the rule (iii) where: $e \in Ev^{?} \cup Ev^{!} \cup Ex \cup \{\surd\}$

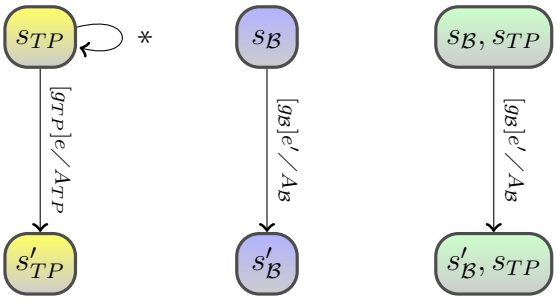

Figure 3.15: Product of the rule (iv) where: $e \neq e'$

states.

Application on the xLoan case study: The STS product of xLoan orchestration is shown in Figure 3.16.

3.4 Deriving Symbolic Test Cases

In this section, we present how symbolic test cases (STC) are generated from the WS-STS model. The first step consists in computing the Symbolic Execution Tree (SET) using the Symbolic Execution (SE) approach and an SMT solver. The SET represents the flattening view of the model, it means that each path of the tree represents a behavioral scenario depicted by the model. The computed paths denote possible test cases. However, with a repetitive behavior, as a loop in the WS-STS model, a huge SET size can be produced. The second step, consists in using some

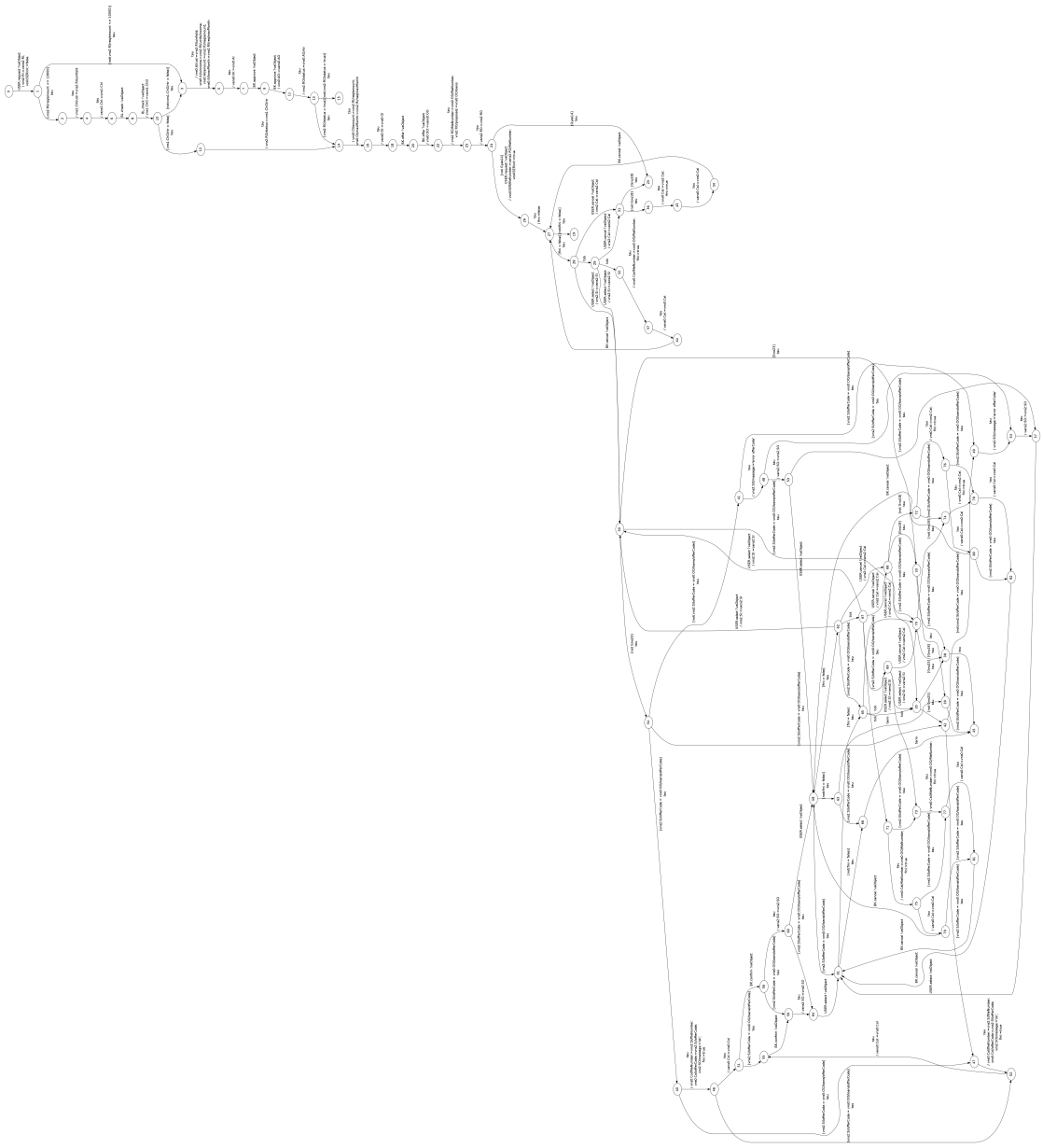

Figure 3.16: The product of the xLoan orchestration with the TP

criteria to circumscribe the generation of the SET. Note that TP could be used to limit the generation of STC, unfortunately sometimes it would not be enough. The final step consists in instantiating those test case and interacting with the orchestrator service. In the following we present in detail, how these steps are accomplished.

3.4.1 Symbolic Execution and Symbolic Execution Tree

Symbolic execution [84] (SE) has been originally proposed to overcome the state explosion problem when verifying programs with variables. SE represents values of the variables using symbolic values instead of concrete data [82].
Consequently, SE is able to deal with constraints over symbolic values, and output values are expressed as a function over the symbolic input values. More recently these techniques have been applied to the verification of interacting/reactive systems, including testing [82, 57, 65]. Using SE approach we present how we generate the Symbolic Execution Tree (SET) from a WS-STS model.

Symbolic Execution Tree (SET). The SE of a program is represented by a *symbolic execution tree* (SET). This SET represents the possible execution described by the WS-STS model. SET consists of nodes connected by edges, where: $\mathcal{N}_{\text{SET}}$ is a set of nodes.
Each node corresponds to a tuple $\eta_i = (s, \pi, \sigma)$ made up of the program counter s, the symbolic values of program variables σ, and a path condition, π.

Let $\mathcal{V}_{\text{symb}}$ be a set of (symbolic) variables (representing symbolic values), disjoint from the program variables, $\mathcal{V}$ ($\mathcal{V} \cap \mathcal{V}_{\text{symb}} = \emptyset$). σ is a map that associate to variable, a symbolic variable : $\mathcal{V} \rightarrow \mathcal{V}_{\text{symb}}$. A path from the SET is a potential symbolic test case with its associate path condition (PC). A PC is a Boolean formula with variables in $\mathcal{V}_{\text{symb}}$. The PC accumulates constraints that the symbolic variables must fulfill in order to follow a given path in the program.

Since we apply SE to an WS-STS, the program counter is an WS-STS state, and $\mathcal{V}$ corresponds to the WS-STS variables (either simple, message type, anonymous, or correlation variables from BPEL). The edges of the SET, $\mathcal{E}_{\text{SET}}$, are elements of $\mathcal{N}_{\text{SET}} \times Ev_{\text{symb}} \times \mathcal{N}_{\text{SET}}$, may be non deterministic. The Ev_{symb} corresponds to the WS-STS events (Ev) with symbolic variables in place of variables.

SET edge computation. The SET is computed in a Breadth-First Search (BFS) fashion as follows:
The root node is $(s_0, true, \sigma_0)$ where s_0 is the initial state of the WS-STS specification model, or the WS-STS product model when a TP was specified. σ_0 is the mapping of a fresh symbolic variable for each variable of the WS-STS ($\sigma_0 : \forall v \in \mathcal{V}, v \mapsto newVar$) and $\pi_0 = true$.
Each transition of the WS-STS $s \xrightarrow{[g] \quad e \quad / \ A} s'$ then corresponds to an edge $(s, \pi, \sigma) \xrightarrow{e'} (s', \pi', \sigma')$, where $\eta = (s, \pi, \sigma)$ and $\eta' = (s', \pi', \sigma')$. The new path condition π' consists of the previous path condition π and the symbolic computation of the guard g, the event e, and the action(s) A from the WS-STS transition. σ' however, is considered as an updating of σ with new mapping(s) between a variable

and a symbolic variable. Those steps are formally described as follows:

1. **guard:** $\pi^G = \pi \wedge g\sigma$, the computation of the new path condition π' begins with the constraint associate to the guard π^G. The latter is a conjunction between π, the previous path condition, and the transition guard in which all the variable are substituted by symbolic variable according to σ. If there is no guard $\pi^G = \pi$,

2. **event:** e', $\sigma^E = \begin{cases} pl.o?v_s, \sigma[v \to v_s] & \text{if } e = pl.o?v \\ e, \sigma & \text{otherwise} \end{cases}$
   If the processing transition of the WS-STS indicates an input event, then a new symbolic variable is created for the incoming input variable, with $v_s = new(\mathcal{V}_{\text{symb}}, \sigma)$, and reported in on the σ^E mapping. Consequently, the edge e' of the SET will be labeled much like the event e of the WS-STS transition, but using the corresponding symbolic variables from σ^E. Otherwise, if the processing transition is not an input event, then e' will be labeled using symbolic variables from σ.
   If e is a service partner, other than the USER, invocation return ($e = pl.o?v_{out} \wedge pl \neq \mathsf{USER}$), we set a constraint $\pi^E = \pi(o)[\sigma^E(v_{in})/in, v_s/out]$, where $e = pl.o!v_{in}$ is the label of the (unique) transition before the one we are dealing with, to take into account the operation specification ($\pi(o)$). Else, $\pi^E = \pi^G$.

3. **actions** ($A = \{x_i/path_i := t_i\}_{i,i\in\{1,\dots,n\}}$)**:**
   $\pi_i^A = \pi_{i-1}^A \wedge (v_{s_{x_i}}/path_i = t_i[\sigma^E(v_j)/v_j]_{v_j \in vars(t_i)})$
   with $\Delta = \{x \in \mathcal{V} \mid (x/path_i := t_i) \in A\}$, $\{v_{s_x}\}_{x\in\Delta} = new^{\#\Delta}(\mathcal{V}_{\text{symb}}, \sigma^E)$, $\sigma' = \sigma^E\{[v_{s_x}/x]\}_{x\in\Delta}$, $\pi_0^A = \pi^E$, and $\pi' = \pi_n^A$.

where $vars$ denotes the variables in a term, $new^n(\mathcal{V}_{\text{symb}}, \sigma)$ denotes the creation of n new (fresh) symbolic variables wrt. σ, $t[y/x]$ denotes the substitution of x by y in t, and $\sigma[x \to x_s]$ denotes σ where the mapping for x is overloaded by the one from x to x_s. Δ is the set of variables that are modified by the assignments. For each of these, we have a new symbolic variable. Note that we suppose without lost of generality that in practice one assign with parallel instructions is executed sequentially.

We denote $may(\eta)$, $\eta \in \mathcal{N}_{\text{SET}}$, the set $\{pl.o!v \mid \exists \eta \xrightarrow{L} *\eta' \xrightarrow{[g] \quad pl.o!v \quad / \ A} \eta"\}$ with L a sequence of labels such that the corresponding word (keeping only the event in labels), contains only non observable events or communication events with partners ($\{\tau, \chi, \surd\} \cup \{pl.o * v \mid * \in \{?, !\} \wedge pl \neq USER\}$). This set will be used later on for the test verdict emission.

Example: Let us consider the simplified WS-STS presented in Figure 3.17. The computation of its associate SET (Figure 3.18) is provided in the following: The set of variables is $\mathcal{V} = \{x_a, x, y_a, y\}$.

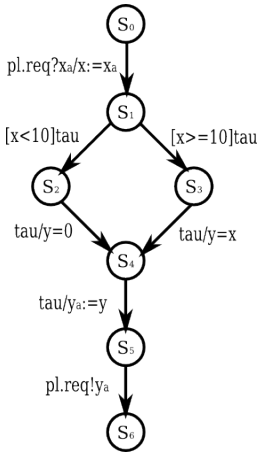

Figure 3.17: WS-STS model

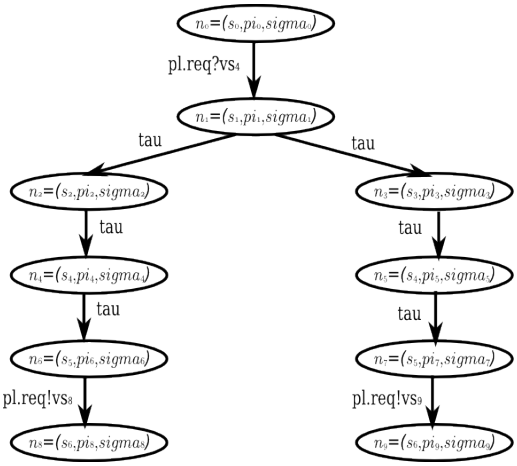

Figure 3.18: SET generation

η_0=(s_0, π_0, σ_0) where: $\pi_0 = true$ and $\sigma_0 = \{x_a \to vs_0, x \to vs_1, y_a \to vs_2, y \to vs_3\}$
η_1=(s_1, π_1, σ_1) where: $\pi_1 = \pi_0 \wedge vs_4 = 7 \wedge vs_5 = vs_4$ and $\sigma_1 = \{x_a \to vs_4, x \to vs_5, y_a \to vs_2, y \to vs_3\}$
η_2=(s_2, π_2, σ_2) where: $\pi_2 = \pi_1 \wedge vs_5 < 10$ and $\sigma_2 = \sigma_1$
η_3=(s_3, π_3, σ_3) where: $\pi_3 = \pi_1 \wedge vs_5 \geq 10$ and $\sigma_3 = \sigma_1$
η_4=(s_4, π_4, σ_4) where: $\pi_4 = \pi_2 \wedge vs_6 = 0$ and $\sigma_4 = \{x_a \to vs_4, x \to vs_5, y_a \to vs_2, y \to vs_6\}$
η_5=(s_4, π_5, σ_5) where: $\pi_5 = \pi_3 \wedge vs_7 = vs_5$ and $\sigma_5 = \{x_a \to vs_4, x \to vs_5, y_a \to vs_2, y \to vs_7\}$
η_6=(s_5, π_6, σ_6) where: $\pi_6 = \pi_4 \wedge vs_8 = vs_6$ and $\sigma_6 = \{x_a \to vs_4, x \to vs_5, y_a \to vs_8, y \to vs_7\}$
η_7=(s_5, π_7, σ_7) where: $\pi_7 = \pi_5 \wedge vs_9 = vs_7$ and $\sigma_7 = \{x_a \to vs_4, x \to vs_5, y_a \to vs_9, y \to vs_7\}$
η_8=(s_6, π_8, σ_8) where: $\pi_8 = \pi_6$ and $\sigma_8 = \sigma_6$
η_9=(s_6, π_9, σ_9) where: $\pi_9 = \pi_7$ and $\sigma_9 = \sigma_7$

In order to reduce the size of an SET we present two possible methods to limit the SET paths generation. Those methods are the *pruning of infeasible paths* and the *pruning of redundant paths.*

Pruning infeasible paths. Edges with inconsistent path conditions may be cut off while computing the SET. For this, we check when computing a new node η if $\pi(\eta)$ is satisfiable (there exists a valuation of variables in π such that π is true), if not, we cut the edge off. This is known to be an undecidable problem in general. Therefore, if the constraint solver does not yield a solution (or a contradiction) in a

given amount of time, we cut the edge off and we issue a warning specifying that the test process is to be incomplete. We use the Z3 SMT solver to reduce the number of infeasible paths. More details are provided in the next Chapter 4.

Pruning redundant paths. WS-STS may contain loops that would cause SET unboundedness. To solve this issue out, we propose two techniques.

We take into account a *path length* criterion while computing the SET. Given a constant k, we stop the SET computation at some node whenever this node is at k edges from the SET root, this technique is inspired by the k bounded model checking [34]. The user can re-start the SET computation process with $k+1$ if he wants more test cases.

A complementary approach is to use the *inclusion criterion* as proposed by [65]. Let us explain this principle helped with the Figure 3.19.

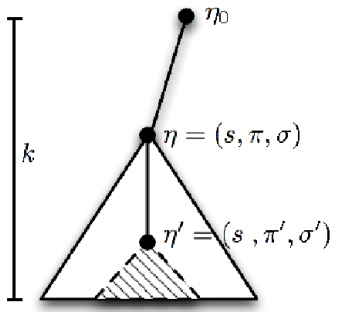

Figure 3.19: The inclusion criterion

Consider $\eta = (s, \pi, \sigma)$, a reachable node in the SET. Solving the associate path condition π means that it exists at least a value for each symbolic variable satisfying the constraint with regards to the σ mapping ($\mathcal{V} \rightarrow \mathcal{V}_{\text{symb}}$). Such constraint allows several interpretations (combinations of possible values). $\mathcal{M}_\eta^{\mathcal{V}}$ represents the set of all the possible interpretations for the variables $\mathcal{V}$ of the node η.
Consider now, an other node $\eta' = (s, \pi', \sigma')$ which has the same state as η. We say that η is included in η' ($\eta \subseteq \eta'$), when $\mathcal{M}_\eta^{\mathcal{V}} \subseteq \mathcal{M}_{\eta'}^{\mathcal{V}}$. That means that the set of interpretations of the variables of $\mathcal{V}$ belonging to η' is bigger than the one belonging to η, so $\pi\sigma \Longrightarrow \pi'\sigma'$. In such case if both nodes are on the same branch than we stop the generation of the SET at the node η, else we stop the computation of the η sub-tree and we create a reference from η to η'. We keep track of the prefix which is related to the test of a different part and with the reference we have a link to the rest of the path already computed.

Symbolic test case extraction. Symbolic test cases correspond to the SET paths. However, it may be relevant to test only paths leading to orchestration termination even if it is not mandatory for Web services since different instances are run for each test case (excepted if the same correlation data are used). Due to our k path length criterion in the SET computation, it follows that symbolic test cases have a length $n \leq k$. Notice that we can increase the value of k if we did not find errors during the execution of tests.

Application on the xLoan case study: The Figure 3.20 presents the SET of the xLoan example. The black path in the tree (also given on the top right part) represents one of the test cases that we may pick and execute against the implementation service. The associated path condition π_{305}, in the end of the path (node 305), is described in the top left part.

3.4.2 Online Testing Algorithm

In this part we present the (online) realization of symbolic test cases into concrete test cases with the use of a constraint solving tool.
First, let us note that since Web services are reactive systems, test case realization has to be performed step by step, by interacting with the Service Under Test (SUT). This is to avoid emitting erroneous verdicts.
Take a path $pl.o?x.pl.o!y$, with $\sigma = \{x \rightarrow v_{s_0}, y \rightarrow v_{s_1}\}$ and $\pi = v_{s_0} > 2 \wedge v_{s_1} > v_{s_0}$. Realization all-at-once would yield a realized path $p?v_{s_0}, p!v_{s_1}$ with, *e.g.* $\{v_{s_0} \rightarrow 3, v_{s_1} \rightarrow 4\}$. Suppose now we send message p with value 3 to the SUT and that it replies with value 5. We would emit a *Fail* verdict ($5 \neq 4$), while indeed 5 would be a correct reply ($5 > 3$).

The online realization process is summarized in Figure 3.21. The tester (or the oracle) is implemented as a reactive process that mirrors the observable behavior of the test case. We begin with the PC in the last state of the SET path corresponding to the test case. Whenever the tester has to send a message to the SUT (USER.o? in the test case), PC is solved to get correct data values to send. Whenever the tester has to receive a message from the SUT (USER.o! in the test case) a timeout is run. If it ends before we receive the message there is an *Inconclusive* verdict. If we receive the message from the SUT, data is extracted from it and the PC is updated with a correspondence between the reception variable(s) and the data, and PC is then checked to see if the data is correct or not. In both cases, we rely on a constraint solving tool. Before using the Z3 solver, we used the UMLtoCSP [38] tool to solve constraints. However, this solver required a very fine tuning of the variables' domains. Moreover, a more complex model transformation between this tool and ours was needed, which is no more the case with the Z3 SMT solver.

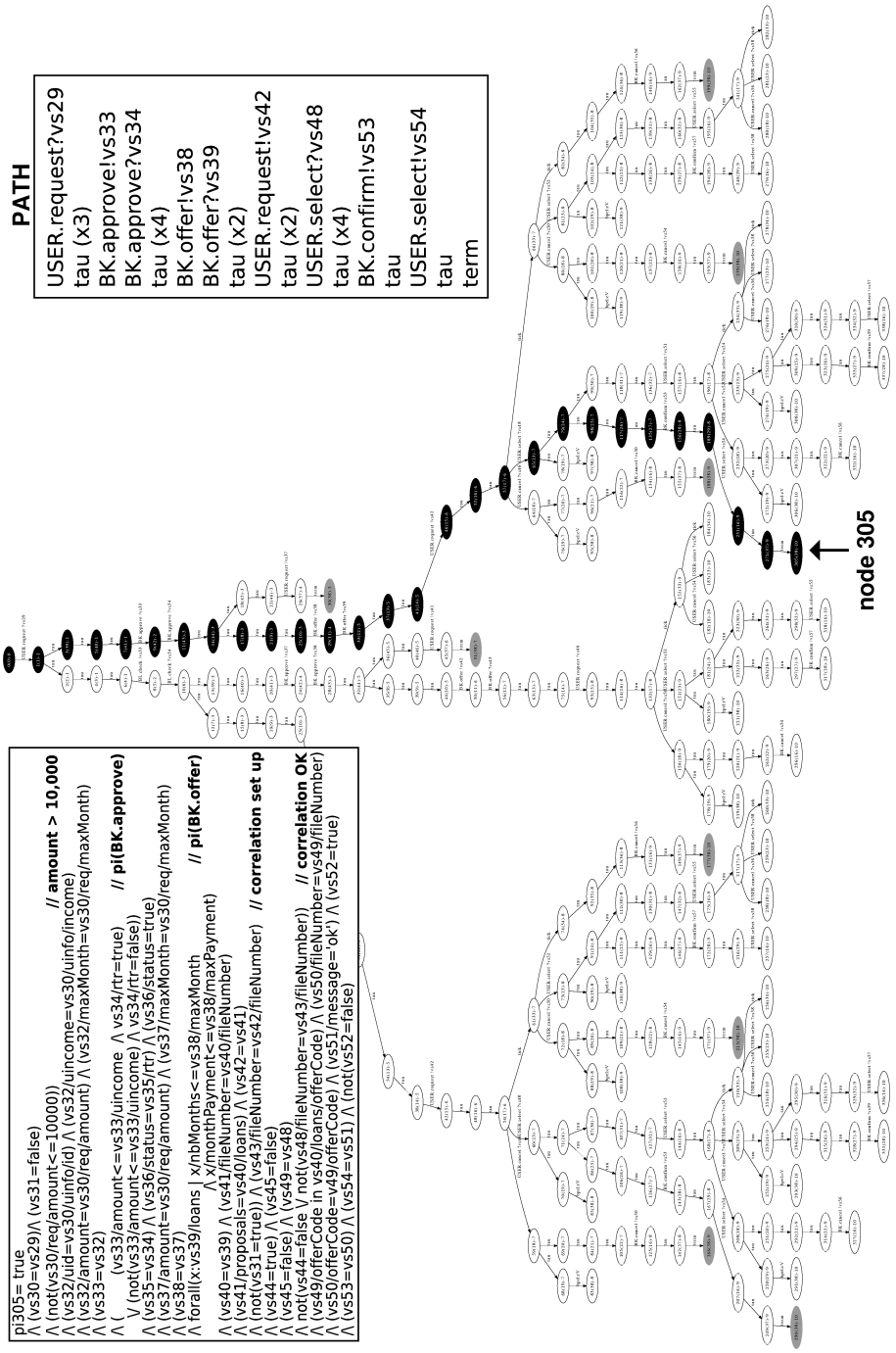

Figure 3.20: Selection of a test case from the SET of xLoan example

Online testing is presented in Algorithm 1. Its input is the SET with a distin-

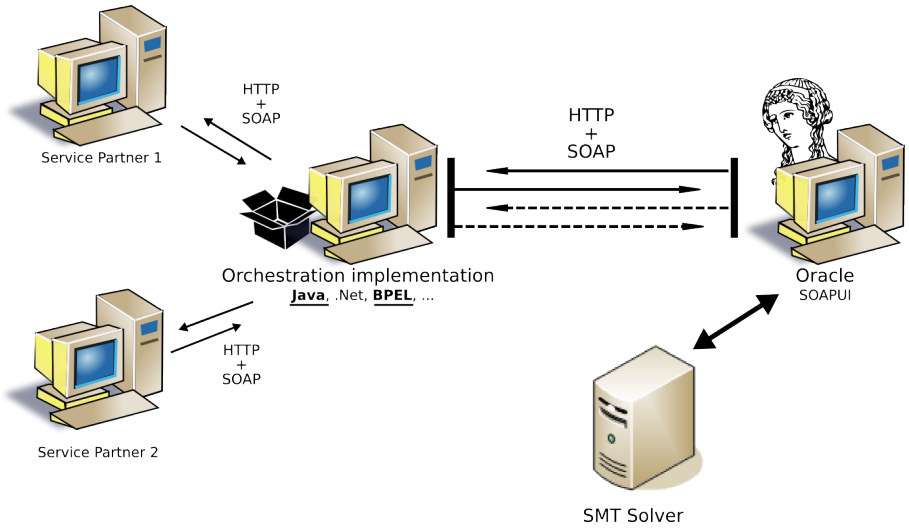

Figure 3.21: Testing Architecture of a Service Orchestration

guished symbolic path we want to test. The algorithm then animates the path by interacting, over messages for the USER partnerlink, with the SUT. Accordingly, input (resp. output) events in the path correspond to messages sent (resp. received) by the tester. Generation of data in sent messages and checking of data in received messages is supported using constraint solving over a Path Condition (PC). Initially PC corresponds to the path condition (π) in the last node of the path we test. The treatment of the path labeled edges (l_i) is then as follows.

- **Input events.** The tester has to generate a message to be sent to the SUT. For this, PC is solved (always succeeds, or the edge would have been cut off in the SET computation). We use the instantiation of the event variable, x_s, to send the message, and to update the PC. If the sent message yields an exception, we return a *Fail* verdict, else we pass to the next edge.

- **Output events.** The treatment of output events corresponds to message reception in the tester. Whenever an emission by the SUT is foreseen, a timer, TAC, is set up. Then three cases may occur. (i). If the timer elapses, we return a *Fail* result. (ii). If we receive the expected message before this, we update the PC with this new information and try to solve it. If it succeeds we continue to the next edge. If it fails we return a *Fail* verdict. If we do not get a result in a given amount of time we return an *Inconclusive* verdict (not in the Algorithm for simplicity). (iii). If we receive an unexpected event, we check in the SET if it is due to the specification non-determinism. If not, we return a *Fail* verdict. If it is the case, we return an *Inconclusive* verdict and the test path needs to be replayed in order to exhibit the behavior that this

Algorithm 1: Online Testing Algorithm

```
Data: SET + a distinguished path p, path p = n_1 l_1 n_2 l_2 ... l_{k-1} n_k ;
begin
    π = π_k; i := 1; rtr := Pass ;
    while i < k and rtr = Pass do
        switch l_i do
            case USER.e?x_s
                val := (SOLVE(π)[x_s]);
                try { send (e(val)); π := π ∧ x_s = val;}
                catch (e ∈ Ex) { rtr := Fail; }
            case USER.e!x_s
                start TAC;
                try {receive (e(val)); π = π ∧ (x_s = val);
                     if ¬SOLVE(π) then rtr := Fail; }
                catch (timeout_TAC) {rtr := Fail;}
                catch (receive e') { if e' ∈ may(η_i) then rtr := Inconclusive;
                                     else rtr := Fail;}
            case χ
                wait(1 unit of time);
            otherwise
                skip;
        i := i + 1;
    return rtr;
end
```

test path characterizes (for this we assume SUT fairness). Fault output events are supported in the same way

- **Time passing (χ)** corresponds to the passing of one unit of time. Accordingly, the tester waits for this time before going to the next event in the path. The unit of time is computed from the specification (one hour in our example). Other events are skipped.

3.5 Conclusion

In this chapter we have presented our symbolic approach for applying conformance testing on an orchestration of Web services.

Our testing approach begins with modeling an orchestration service from an ABPEL specification into a WS-STS model. This is achieved using our transformation

rules. We also support the possibility of generating specific test cases using Test Purposes (TP).
Indeed, we defined rules to compute the product between the specification and the test purpose models and thus, the generated product model describes the behavior of the composite service and handles the defined test purpose.

We used the Symbolic Execution (SE) on the model (or the product model), in order to avoid the problem of state space explosion when dealing with concrete data. The SE generates a Symbolic Execution Tree (SET) that represents the WS-STS execution semantics. Each path of the SET had its path condition (PC) that must be satisfied to execute the corresponding behavior. However, using the SE we generate symbolic test cases, that we need to realize, in order to execute the concrete test cases against the implementation service. For this aim, we presented an (online) algorithm that describes: (i) the interactions between the tester and the Z3 SMT solver, to instantiate the test cases and (ii) the interactions between the tester and the service under test in order to execute the test cases. Finally, a verdict on the conformance of the composite service with regard to its specification is emitted.

CHAPTER 4

Implementation and Tools support

In this chapter we start by presenting the test context also called *testing architecture*. Then, The main part of the chapter is devoted to the presentation of the tool chain that we have developed to support our approach.
The tool chain, as represented in Figure 4.11, includes: (i) The `BPEL2STS` tool that performs the transformation of a specification written in ABPEL into a Symbolic Transition System (WS-STS or STS for short) model. (ii) Whenever a Test Purpose (TP) is given, the `STSProd` tool is used to compute the product between the specification model and the test purpose model. Thus, the test purpose will be handled in the generation of test cases. (iii) The `STS2SET` tool is used for the generation of the Symbolic Execution Tree (SET) from which we will retrieve the test cases. Finally, we describe the testing process of a the Service Under Test (SUT) and the use of the Z3 SMT solver in order to generate the adequate input data needed for the interaction with the service orchestrator.

4.1 Testing Architecture

In the testing process, the context and the conditions in which test cases are executed against the implementation play an important role. With this information, also called test architecture and a test context, a tester (oracle) is able to identify errors and to know in which conditions these errors appear.

Since we are focusing on testing a service orchestration, we may distinguish two

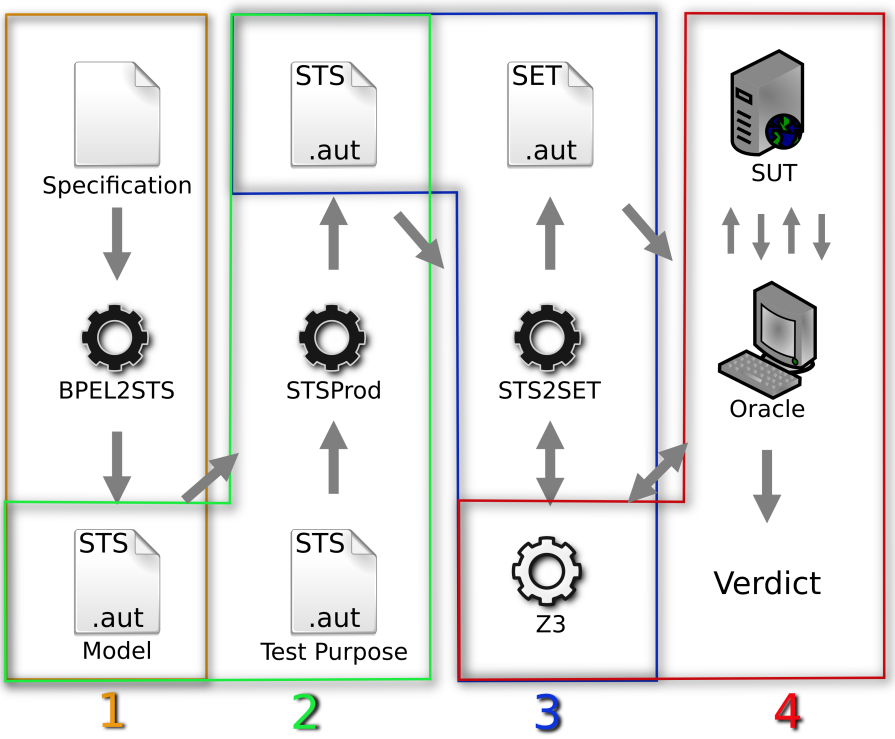

Figure 4.1: Overview of the tools chain

kinds of testing architecture: the *in-the-large* architecture and the *restricted* one.

As presented in the previous chapters, an orchestration of services consists of a service orchestrator and other partner services. Among those services, we have a specific one, the USER service which is the one that contributes in initiating the conversation protocol with the service orchestrator. Note that a composite service may have many USER services, for simplicity sake we specify only one.
Using an in-the-large testing architecture, a tester will simulate all the partner services as shown in Figure 4.2. Such architecture is used when the tester obtains the WSDL addresses of the partner services and then simulates them using dedicated tools like soapUI.
By contrast, a restricted testing architecture focuses only on the USER service as shown in Figure 4.3. In this kind of architecture, a testing hypothesis states that the other partner services, are correct.

In our work we use a restricted testing architecture. Since we are interested in checking if an implementation is consistent with its functional specification through an observable behavior, we do not need to simulate the other service partners. It would be interesting to use the in-the-large testing architecture for other sort of testing, such the robustness testing, where the tester can simulate erroneous responses

from the services partners and observe the behavior of the service orchestrator. This is left as a perspective.

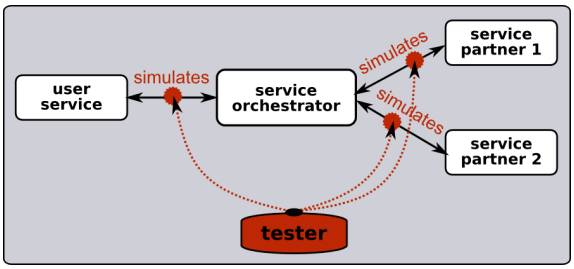

Figure 4.2: In-the-large testing architecture

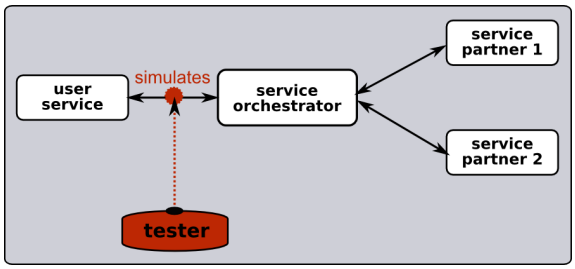

Figure 4.3: Restricted testing architecture

4.2 Conformance Testing of a Service Orchestrator

The Algorithm 2 represents the overall algorithm for the testing of service orchestration.

Algorithm 2: **The complete testing process of a BPEL orchestration**

Input: $Spec_{\mathcal{B}}$: $ABPEL$, TP:WS-STS, k:$\mathbb{N}$, $crit$: $SET_{inclusion}$, imp: $WSDL_{@}$
Output: verdict:$\{pass, fail, inconc\}$
1 **Variables**: $\mathcal{B}$: WS-STS, M_T: WS-STS, $\mathcal{T}$:SET
2 **Initialization**: verdict:= $pass$, i:=0
3 $\mathcal{B}$:=`BPELtoSTS`($Spec_{\mathcal{B}}$)
4 M_T:=`STSProd`($\mathcal{B}$,TP)
5 $\mathcal{T}$:=`STS2SET`(M_T, k)
6 STC:=`getPaths`($\mathcal{T}, crit$)
7 nbTest:= `size`(STC)
8 **while** $verdict \neq fail$ **and** $i < nbTest$ **do**
9 verdict := `verdictResponse`(verdict, `onlineTest`(STC[i], imp))
10 i++
11 **end**
12 **return** verdict

This algorithm takes as inputs, an ABPEL specification ($Spec_{\mathcal{B}}$), a test purpose expressed as a WS-STS model (TP), an integer number (k) representing the maximum bound for the SET length, with the inclusion criterion ($crit$) and the WSDL address of the implementation under test. By default the verdict is initialized to a pass one.

The first step, consists in the generation of the WS-STS specification model ($\mathcal{B}$) from the ABPEL specification using the `BPEL2STS` tool. In order to handle the specification of test purpose, a product is computed between the WS-STS model specification and the WS-STS of the test purpose, yielding a test model M_T. The next step is the computation of the SET. For that, the symbolic execution method is applied to unfold the test model behavior up to depth k using the `STS2SET` tool. Symbolic Test Cases (STC) are generated from the SET paths, according to the specified inclusion criterion. Then each STC is realized and executed stepwise against the implementation until a fail verdict is emitted or there are no more remaining STC. For each STC execution a verdict is emitted and the global verdict is updated. The global verdict is refreshed by taking into consideration the previous global verdict and the last STC verdict.

Algorithm 3 explains how the new global verdict is computed. In this algorithm the previous global verdict is represented by the $verdict_{prev}$ variable, the last STC emitted verdict by $verdict_{last}$ and the new global verdict by the $response$ variable. In case, the emitted verdict is a fail one, then the returned response will be also a fail. A second possibility may occur when the emitted verdict is an inconclusive, in that case the response will be inconclusive too. In principal, if such case occurs the STC is re-executed against the implementation. The last case is the emission of pass verdict.

Algorithm 3: **verdictResponse**

Input: verdict$_{prev}$: $\{pass, fail, inconc\}$, verdict$_{last}$: $\{pass, fail, inconc\}$
Output: response:$\{pass, fail, inconc\}$

```
1  if verdict_prev==fail or verdict_last==fail then
2  |  response:= fail
3  else
4  |  if verdict_prev==inconc or verdict_last==inconc then
5  |  |  response:= inconc
6  |  else
7  |  |  response:= pass
8  |  end
9  end
10 return response
```

4.2.1 ABPEL to WS-STS transformation

To rigorously describe the behavior of an orchestration the `BPEL2STS` tool is used to generate the corresponding formal model. From an ABPEL specification, the

`BPEL2STS` tool generates the associated WS-STS model. This tool is an implementation of the transformation rules (presented in the Chapter 3, section 3.3.2). It had been developed within the Project ANR *Pervasive Service Composition*[1] (PERSO).

4.2.2 Test Purpose Support

One may want to guide the generation of test cases in order to focus on particular functional aspects to be tested. The test purposes (TP), represent the specification of a test scenario. In our approach a TP is described as a WS-STS model. To incorporate a test purpose to the WS-STS specification model, a product of models must be computed.

In this part we present how the theoretical rules, defined in Chapter3, of computing the product between the specification model and the test purpose model, are executed.

Algorithm 4: WS-STS Computation

Input: $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$: WS-STS,
$TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$: WS-STS
Output: $M_T = (\mathcal{D}, \mathcal{V}, S, s_0, T)$: WS-STS

1 $s_0 = \{(s_{0_{\mathcal{B}}}, s_{0_{TP}})\}$; $\mathcal{D} = \mathcal{D}_{TP}$; $\mathcal{V} = \mathcal{V}_{TP}$; $T = \emptyset$; $E = s_0$; $E = S$
2 **while** $E \neq \emptyset$ **do**
3 take $s \in E$
4 **forall** $i \in \{1, 2, 3, 4\}$ **do**
5 (S_{new}, T_{new}) = `Product-Rule` i ($\mathcal{B}, TP, s$)
6 $T = T \cup T_{new}$
7 $E = E \cup S_{new}$
8 $S = S \cup S_{new}$
9 **end**
10 $E = E - \{s\}$
11 **end**
12 M_T= `coreach` (M_T)
13 **return** (M_T)

In the Algorithm 4, the symbolic product is implemented according to our previous four product computation rules. The algorithm takes the specification ($\mathcal{B}$) model and the TP model as input parameters and returns the computed product model name the M_T.
The computation of new states of M_T is achieved using a temporary set of states E that contains the M_T states to be processed. For each state of E the computation

[1]PERSO Project: *http : //www.lri.fr/ poizat/ANR_PERSO/*

Algorithm 5: **Product-Rule 1**

Input: $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$: WS-STS,
$TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$: WS-STS,
$s = (s_{\mathcal{B}}, s_{TP}) \in S_{\mathcal{B}} \times S_{TP}$: State
Output: S_{new}: State set, T_{new}: Transition set

$S_{new} = T_{new} = \emptyset$
forall $s_{TP} \xrightarrow{[g] \quad e \quad / \ A} s'_{TP} \in T_{TP}$ **do**
  if $e \in \{\tau, \chi, \sharp\}$ **then**
    $s_{new} = (s_{\mathcal{B}}, s'_{TP})$
    $S_{new} = S_{new} \cup \{s_{new}\}$
    $T_{new} = T_{new} \cup s \xrightarrow{[g] \quad e \quad / \ A} s_{new}$
  end
end
return (S_{new}, T_{new})

Algorithm 6: **Product-Rule 2**

Input: $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$: WS-STS,
$TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$: WS-STS,
$s = (s_{\mathcal{B}}, s_{TP}) \in S_{\mathcal{B}} \times S_{TP}$: State
Output: S_{new}: State set, T_{new}: Transition set

$S_{new} = T_{new} = \emptyset$
forall $s_{\mathcal{B}} \xrightarrow{[g] \quad e \quad / \ A} s'_{\mathcal{B}} \in T_{\mathcal{B}}$ **do**
  if $e \in \{\tau, \chi, \sharp\}$ **then**
    $s_{new} = (s'_{\mathcal{B}}, s_{TP})$
    $S_{new} = S_{new} \cup \{s_{new}\}$
    $T_{new} = T_{new} \cup s \xrightarrow{[g] \quad e \quad / \ A} s_{new}$
  end
end
return (S_{new}, T_{new})

Algorithm 7: **Product-Rule 3**

Input: $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$: WS-STS,
$TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$: WS-STS,
$s = (s_{\mathcal{B}}, s_{TP}) \in S_{\mathcal{B}} \times S_{TP}$: State
Output: S_{new}: State set, T_{new}: Transition set

1 $S_{new} = T_{new} = \emptyset$
2 **forall** $s_{\mathcal{B}} \xrightarrow{[g_{\mathcal{B}}] \quad e_{\mathcal{B}} \quad / \ A_{\mathcal{B}}} s'_{\mathcal{B}} \in T_{\mathcal{B}}$ **do**
3 **forall** $s_{TP} \xrightarrow{[g_{TP}] \quad e_{TP} \quad / \ A_{TP}} s'_{TP} \in T_{TP}$ **do**
4 **if** $e_{\mathcal{B}} = e_{TP}$ **then**
5 $s_{new} = (s'_{\mathcal{B}}, s'_{TP})$
6 $S_{new} = S_{new} \cup \{s_{new}\}$
7 $T_{new} = T_{new} \cup s \xrightarrow{[g_{\mathcal{B}} \wedge g_{TP}] \quad e_{\mathcal{B}} \quad / \ A_{\mathcal{B}}; A_{TP}} s_{new}$
8 **end**
9 **end**
10 **end**
11 **return** (S_{new}, T_{new})

of the following state(s) is performed according to the product rule 1, 2, 3 or 4 as presented in the Algorithm 5, 6, 7, or 8 respectively.

The Algorithm 5 represents the execution of the rule (i) in Section 3.3.4. This algorithm iterates over all the transitions of the TP and checks whether the transition event is an internal event ($e \in \{\tau, \chi, \sharp\}$). If it is the case then, a new state and transition are built. The new state will be composed of the state of $\mathcal{B}$ ($s_{\mathcal{B}}$), and the next state of TP model (s'_{TP}). However, the new created transition has the same labeling transition as the TP. The Algorithm 6 represents the execution of the rule (ii) in Section 3.3.4. It is identical to the Algorithm 5 except that its reasoning concerns a $\mathcal{B}$ transitions.

Algorithm 7 represents the synchronization of the transition $\mathcal{B}$ and TP on the same event ($e_{\mathcal{B}} = e_{TP}$). It results a new state for M_T, composed by $s'_{\mathcal{B}}$ and s'_{TP}, and a new M_T transition which is a combination of $T_{\mathcal{B}}$ and T_{TP}. This algorithm is associate to the rule (iii) in Section 3.3.4.

Finally, the Algorithm 8 process the case in which the event of T_{TP} is equal to * ($e_{TP} = *$). The first step is to check if there is a e_{TP} that could be synchronized with the $e_{\mathcal{B}}$. This verification is performed by using the boolean variable $found$. When there are no synchronization, the second step consists in creating a new state and transition for M_T based on the $T_{\mathcal{B}}$. The Algorithm 8 is associate to the rule (iv) in Section 3.3.4.

The Algorithm 9 aims to clean the M_T model by keeping only the states and

Algorithm 8: **Product-Rule 4**

Input: $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$: WS-STS,
$TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$: WS-STS,
$s = (s_{\mathcal{B}}, s_{TP}) \in S_{\mathcal{B}} \times S_{TP}$: State
Output: S_{new}: State set, T_{new}: Transition set

$S_{new} = T_{new} = \emptyset$
forall $s_{\mathcal{B}} \xrightarrow{[g_{\mathcal{B}}] \quad e_{\mathcal{B}} \quad / \ A_{\mathcal{B}}} s'_{\mathcal{B}} \in T_{\mathcal{B}}$ **do**
  found := false
  forall $s_{TP} \xrightarrow{[g_{TP}] \quad e_{TP} \quad / \ A_{TP}} s'_{TP} \in T_{TP}$ **do**
    if $e_{\mathcal{B}} = e_{TP}$ **then**
      found:= true
      break
    end
  end
  if *found = false* **then**
    forall $s_{TP} \xrightarrow{[g_{TP}] \quad e_{TP} \quad / \ A_{TP}} s_{TP} \in T_{TP}$ *and* $e_{TP} = *$ **do**
      $s_{new} = (s'_{\mathcal{B}}, s_{TP})$
      $S_{new} = S_{new} \cup \{s_{new}\}$
      $T_{new} = T_{new} \cup \{s \xrightarrow{[g_{\mathcal{B}}] \quad e_{\mathcal{B}} \quad / \ A_{\mathcal{B}}} s_{new}\}$
    end
  end
end
return (S_{new}, T_{new})

Algorithm 9: **coreach function**

Input: M_T: WS-STS
Output: M_T: WS-STS

foreach *path* $\in M_T$ **do**
  if *path contains a* $\sharp$ *transition label* **then**
    M':= copy the path
  end
  M_T:= model construction from M' paths
end
return M_T

transitions that leads to the acceptance states defined in the TP.

4.2.3 SET Computation

After generating the formal model M, we present through the Algorithm 10 its unfolding process. This unfolding represents the semantic of the formal model execution.
Note that the formal model is the test model M_T when a test purpose is specified ($M = M_T$). Otherwise it represents the specification model ($M = \mathcal{B}$).

An SET is characterized by a set of nodes ($\mathcal{N}_{\text{SET}}$) and a set of edges ($\mathcal{E}_{\text{SET}}$). Each node describes its associate WS-STS state (s), the path condition (π) that must be satisfied in order to reach this node and the σ function that maps each variable ($v \in \mathcal{V}$) with its corresponding symbolic variable ($vs \in \mathcal{V}_{\text{symb}}$). An edge is a link between two nodes. This link is label with a symbolic event.

To generate the SET, the first step of the Algorithm 10 consists in creating the initial node η_0. This node is made up of the s_0 state from the WS-STS, the initial condition which is *true* and the σ function which associates to each variable a new symbolic variable.
The algorithm describes the use of two temporary set of nodes: $nodeProcessing$ and $visitedNodes$. The $nodeProcessing$ set contains the unhandled nodes yet, such as the initial node η_0. A copy of the $nodeProcessing$ is saved in $visitedNodes$. The $nodeProcessing$ is then emptied in order to receive new unhandled nodes.

While, it still exist unhandled nodes of ($nodeProcessing$) and the maximum depth k is not reached yet, the algorithm continues the computation of the SET. Each node η of the $visitedNodes$ will be processed by extracting information about the out transitions from the state parameter of this node. For each out transition the Algorithm 11 is used to generate new node(s) and edge(s).

Once the path condition π' of the new generated node is computed, it is sent to the solver in order to check its satisfiability. If the solver can find at least one concrete value for each symbolic variables, then it replies with *sat*. This implies that the path condition is feasible and the unfolding of the WS-STS continues. However, if the solver replies with an *unsat* response, this means that the unfolding stops and then the algorithm processes the next unhandled node. If π' is satisfiable then the inclusion criterion is checked (line 24 of the algorithm), in order to detect repetitive behavior and thus decide stopping or not the generation of the SET path. Finally, the created node an edge are added to $\mathcal{N}_{\text{SET}}$ and $\mathcal{E}_{\text{SET}}$ respectively.
The creation of the new node (η') and edge is depicted in the Algorithm 11.

Algorithm 10: **SET generation**

Input: $M = (\mathcal{D}, \mathcal{V}, S, s_0, T)$: WS-STS, k: $\mathbb{N}$, *crit*: $SET_{inclusion}$
Output: $\mathcal{T} = (\mathcal{N}_{\text{SET}}, \mathcal{E}_{\text{SET}})$: SET
Variables: $vs_i \in \mathcal{V}_{\text{symb}}$, nodeProcessing: set of SET nodes, visitedNodes: set of SET nodes, edge $=(\mathcal{N}_{\text{SET}} \times Ev_{\text{symb}} \times \mathcal{N}_{\text{SET}}) \in \mathcal{E}_{\text{SET}}$, node $\in \mathcal{N}_{\text{SET}}$
Initialization: $i = 0$, $depth = 0$
while $v \in \mathcal{V}$ *and* $i \leqslant size(\mathcal{V})$ **do**
    $vs_i :=$ `createNewSymbVar`(v)
    $\sigma_0 := (v, vs_i)$
    $i++$
end
$\pi_0 :=$ true
$\eta_0 := (s_0, \pi_0, \sigma_0)$
$\mathcal{N}_{\text{SET}} := \{\eta_0\}$
nodeProcessing$:= (\{\eta_0\})$
while *nodeProcessing* $\neq \emptyset$ **do**
    visitedNodes := nodeProcessing
    depth++
    nodeProcessing$:=\emptyset$
    if *depth* $<$ k **then**
        foreach $\eta \in visitedNodes$ **do**
            foreach $t \in$ `outTransitionStateOf`(η) **do**
                $(\eta', edge) :=$ `createNewNodeAndEdge`(η, t)
                // where $\eta' :=$ (`targetStateOf`(t), $\pi_{\eta'}$, $\sigma_{\eta'}$)
                satisfiability := `checkSat`$(\pi_{\eta'})$
                if *satisfiability*$==$*sat* **then**
                    foreach $node \in \mathcal{N}_{\text{SET}}$ **do**
                        if `stateOf`(η') = `stateOf`$(node)$ *and* $\pi_{node} \subseteq \pi_{\eta'}$ **then**
                            stop the unfolding of this path
                        end
                    end
                    nodeProcessing$:=$ nodeProcessing $\cup\{\eta'\}$)
                    $\mathcal{N}_{\text{SET}} := \mathcal{N}_{\text{SET}} \cup \{\eta'\}$
                    $\mathcal{E}_{\text{SET}} := \mathcal{E}_{\text{SET}} \cup \{edge\}$
                end
            end
        end
    end
end
return $\mathcal{T}$

Algorithm 11: **Create new node and edge**

Input: $\eta{=}(\eta, \pi, \sigma) \in \mathcal{N}_{\text{SET}}$, $t{=}\ (s, geA, s') \in$ WS-STS transition
Output: $\eta'{=}(\eta', \pi', \sigma') \in \mathcal{N}_{\text{SET}}$, edge$=(\eta, e_{\text{symb}}, \eta') \in \mathcal{E}_{\text{SET}}$,

$\eta'{:=}\ (s',\ \pi_\eta,\ \sigma_\eta)$
if $g \neq \emptyset$ **then**
  $\pi_{\eta'}{:=}\ \pi_{\eta'} \wedge$ `getGuardWithSymbVar`$(g, \sigma_{\eta'})$
end
if *e is an input* **then**
  create new symbolic Variables for received and assigned variables
  update $\sigma_{\eta'}$ with the created symbolic variables
  e_{symb} := `getEventWithSymbVar`$(e,\ \sigma_{\eta'})$
else
  if *e is an output* **then**
    e_{symb} := `getEventWithSymbVar`$(e,\ \sigma_{\eta'})$
  else
    $e_{\text{symb}}{:=}\ e$
  end
end
if $A \neq \emptyset$ **then**
  foreach $a \in A$ **do**
    create new symbolic variables for the assigned variable
    update $\sigma_{\eta'}$ with the created symbolic variables
    $a_{\eta'}$:= `getActionWithSymbVar`$(A,\ \sigma_{\eta'})$
    $\pi_{\eta'} := \pi_{\eta'} \wedge \pi_{a_{\eta'}}$
  end
end
return $(\eta', edge)$

The state parameter of η' is the target state of the transition t. The computation of the constraint π' associated to η' represents a conjunction between the previous node constraint, the constraints of the guard, the event (if it is an input event) and the actions of the WS-STS transition. All of these constraints are expressed using symbolic variables from σ'. The latter is updated with new symbolic variables when a variable is received or for the assigned variables in the action part of the transition. The new created edge (e_{symb}) is label from the event of t, using symbolic variables as mapped in $sigma'$

4.2.4 Symbolic Test Cases

Each path in the SET represents a potential symbolic test case (STC). The path condition (PC) associated to a path acts as a restriction on the possible data value, thus the test scenario can be executed. This restriction is due to the interdependence of the PC variables and the satisfaction of its specified conditions.

4.2.5 Online realization with a constraint solving tool

We presented the interactions between the tester (also called the oracle) and the implementation service in the online testing of Web services (Chapter 3.Section 3.4.2) trough the Algorithm 1.
In this subsection we explain the realization part that allows us to obtain concrete data for the communication with the implementation. In other words, we present how the solving of predicates, used in the SET computation (Chapter 3.Section 3.4.1), and the online testing Algorithm 1 is achieved using the Z3 SMT solver. This is achieved in four steps:

1. Computation of a tree representing the structure of each variable
2. Creation of a new variable for each tree leaf
3. Creation of a new variable for each node that plays a role in the predicate to be solved
4. Dumping into Z3 format, launching Z3 and parsing results

The first algorithm (Algorithm 12) computes the tree structure of the variables. The tree is created according to the variable type using the $createTreeFromType$ function as describes in the Appendix C (Algorithm 16). If the variable has a simple type then a leaf is created. Otherwise, the tree structure is constructed according to the variable type structure.

Example: Let us assume a variable x, a variable y and a complex type named T_1. T_1 consists of a part a and another part b, both lead to an integer value. We provide the following annotation for this complex type: $T_1 : \{a : Int, b : Int\}$.
The variables x and y have the following types: $x :: Int$ and $y :: T_1$.

The corresponding tree for x will be reduced to an integer leaf and the one associated to y is a tree with an integer leaf through edge label a and an integer leaf through edge label b. We provide in Figure 4.4 a graphical view of the trees.

When the trees corresponding to variables with complex type are constructed, a new variable is assigned to each leaf of the tree, as presented in the Algorithm 13.

Algorithm 12: **Creation of the associate tree for each variable of $\mathcal{V}$**

Input: a set of Variables $\mathcal{V}$
Output: Creation of the trees for each variable of $\mathcal{V}$

```
1 foreach v ∈ V do
2 |   type_v = typeOf(v)
3 |   Tree t = createTreeFromType(type_v)
4 |   valueOf(v) = t
5 |   variableOf(t) = v
6 end
```

h

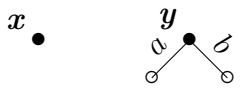

Figure 4.4: Creation of trees from variables types

These new variables will be used as pointers when the instantiation of variables of complex type will be performed. In fact, each path of the tree starting from the root to a leaf is represented with a unique variable as shown in Figure 4.5 for the previous example.

Algorithm 13: **creation of new leaf variables**

Input: a set of Variables $\mathcal{V}$
Output: a set of variables $\mathcal{V}$ within the new leaf variables

```
1 VA := ∅
2 foreach v ∈ V do
3 |   if typeOf(v) ≠ simpleType then
4 |   |   VA = VA ∪ createNewLeafVars(v.type, v.value)
5 |   end
6 end
7 return V ∪ VA
```

Once the previous step is achieved, in the next one we process the predicate. A predicate is a conjunction of clauses. Theses clauses may represent an equivalence between variables or/and paths in trees (an $XPath$ expression) as well as arithmetic and logic operations.
Let us consider the same previous variables x and y and the following predicate:

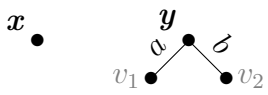

Figure 4.5: Creation of leaf variables for the trees

Algorithm 14: Creation of variables for clauses

Input: a set of Variables V, a constraint C
Output: a set of variables V within the new VC variables

```
1 foreach c ∈ clausesOf(C) do
2 |   V = V ∪ createNewVCVars(V, leftPartOf(c))
3 |   V = V ∪ createNewVCVars(V, rightPartOf(c))
4 end
5 return V
```

$x = 3 + 4 \wedge y/b = x$.

The clause $y/b = x$ expresses the equivalence between the value of the variable x and the value of the location pointed by the path y/b.
In case the $XPath$ expression leads to a leaf, then the path is replaced by its associated variable. If the $XPath$ expression y/b does not correspond to a leaf, but to a sub-tree then a new variable is created to represent this sub-tree.
To demonstrate this, let us consider the variable z (in addition to x and y) and a new complex type T_2 where: $T_2 : \{a : Int, b : T_1\}$ and $z :: T_2$.
Let us also suppose we have: $x = 3 + 4 \wedge y/b = x \wedge z/b = y$

In such a case, a new variable will be created to represent z/b sub-tree. The graphical view is given Figure 4.6

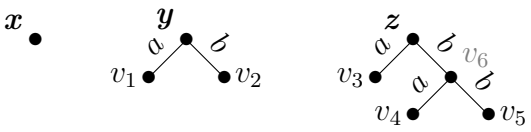

Figure 4.6: Creation of new variables form the the subtrees

The last step is to submit the predicate to the Z3 solver. For this, we replace the variables in it by variables created in previous steps, *e.g.* we solve: $x = 3 + 4 \wedge v_2 = x \wedge v_6 = y$ (given the structure of x, y, z, v_2, v_6)
However, the solver can not process the predicate as it is. A transformation of this

predicate into an understandable format by the solver must be done.
For this purpose we defined a function that generates an SMT input file for Z3 solver. This input file represents a problem to be submitted to Z3 and has to satisfy the following BNF rules:

Problem	::=	Prefix Theory Var-Declaration Constraint-Formula Suffix
Prefix	::=	" `( benchmark data` "
Theory	::=	Labels-Declaration String-Declaration Tree-Declaration
Var-Declaration	::=	" `:extrafuns (` " *(variableName variableType)*$^+$ " `)` "
Constraint-Formula	::=	"`:formula ( and` " *variable-Structure constraintClauses*"`)`"
Suffix	::=	" `)` "
Labels-Declaration	::=	" `:datatypes ((label` " *Labels** " `)` "
String-Declaration	::=	" `(String` ("*word*$^+$ | "`noString`)" `)`
Tree-Declaration	::=	"`(tree (nil) (IntLeaf (val Int))` `(RealLeaf (val Real))(StringLeaf (val String))` `(BoolLeaf (val Bool))` `(cons (firstChildLab label) (firstChild tree)` `(siblingChilds tree))))`"

A problem is divided into three main sub-parts: the *Theory*, the *Var-Declaration* and the *Constraint-Formula*. We use the `teletype font` to indicate the part that will appear as it is, in the Z3 input file.

The *Theory* is used to specify the string words allowed as a given database and also to specify the structure of a complex type variable. The second part, i.e. the *Var-Declaration*, allows to declare each variable and its type. The different types can be: *Int*, *Real*, *Bool* or a *tree* type. We also support String using a set of constructors.
Finally, the *Constraint-Formula* sub-part allows to represent at first the structure of a complex variable type as specified by the theory (this corresponds to the "*variable-Structure*" in the rules above). Secondly the predicate (or the constraint), presented as a conjunction of clauses, will be written into a particular form (this corresponds to the *constraintClauses* in the rules above).
To explain this last point, let us consider the following simple example: $x = 3 + 4$. The corresponding constraint with the concrete syntax of Z3 will be written as follows: $(= x\ (+\ 3\ 4))$.

Labels* represent the set of tree labels of complex type variables, there is no labels for simple type variables. The *word*$^+$ item represents the words given as input to the program. The *Tree-Declaration* defines a tree structure.

We give below an example of a tree given as an input to the solving function and how we encode it with the concrete syntax of Z3. We also give a linear representation

of the tree, with the label connected to its ancestor, its first child with its descendants and all the siblings of this child and their descendants. The *nil* constructor represents an empty tree.
To help the understanding of the constraints solving process, we present the following example. Let us consider the variable w with a complex type: $w :: T$, where: $T_1 : \{d : Int, e : Int\}$, $T_2 : \{f : Int, g : Int\}$ and $T : \{a : Int, b : T_1, e : T_2\}$.

The tree structure of w is graphically represented in the left part of the Figure 4.7. The associated description of w in the concrete syntax of Z3 as follows:

$$(= w\ (cons\ a\ (IntLeaf\ v_0)\ (cons\ b\ (cons\ d\ (IntLeaf\ v_1)\ (cons\ e\ (IntLeaf\ v_2)\ (nil)))(cons\ c\ (cons\ f\ (IntLeaf\ v_3)\ (cons\ g\ (IntLeaf\ v_4)\ (nil))))\ (nil))))$$

The graphical representation of w as an input format for Z3 is given in the right part of the Figure 4.7. The *nil* symbol, represented as $\perp$, is associated to some nodes to highlight the fact that those nodes do not have siblings, *e.g.* the root node w or the node c.

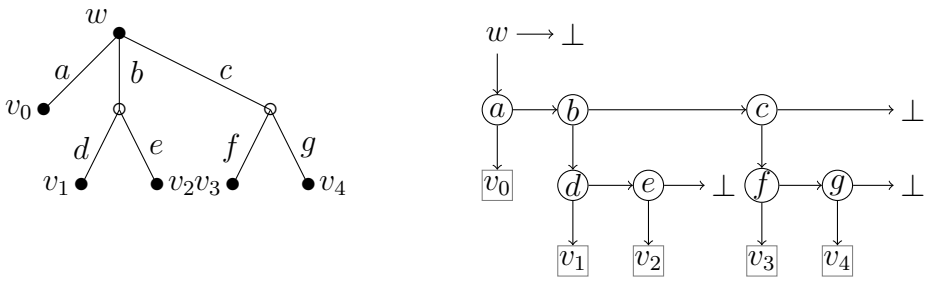

Figure 4.7: The structure of the w variable

The generated Z3 input file is submitted to the solver. The latter, returns an answer about the predicate satisfiability and/or an instantiation of variables. The predicate could be *satisfiable* (*sat*) or an *unsatisfiable* answer (*unsat*). A third answer could be given, which is the *unknown* response (*unknown*). The unknown answer is emitted when the solver cannot give a response in a given amount of time.
When an instantiation of the variable is provided, a solving function is in charge of retrieving those values then assigning them to the variables. If the variables have a tree representation then the leaves variables will receive the different values and thus complete the construction of trees. The instantiated variables will be used during the test case execution.

The function that translates the predicate into an input file for the Z3 SMT solver, can easily be modified to interact with different solving tools. All we need to

add are some methods for the dumping of the problem to be solved in the syntax accepted by the tool and also to add a method to retrieve the responses of this tool.

Here we present an example of how the interaction process with the Z3 solver. Let us assume the following inputs for the solving function. T_1 and T_2 are complex types: $T_1 : \{a : Int, b : Real\}$ and $T_2 : \{a : String, b : T_1\}$. The input variables x, y and z have the following types: $x :: Bool$, $y :: T_1$ and $z :: T_2$.
Further, the predicate is : $x = true \wedge y/a = 8.2 + 4.0 \wedge z/b = y$ and $\{hello, everyone\}$ are possible string words.

The graphical view of the variables and their trees is the same as in the Figure 4.6. The Z3 input file for this example is given in the Figure 4.8: the *datatypes* part provides the name of the used labels the allowed string words and the theory that describes a tree structure. The second part, the *extrafuns*, is used for variables declaration and the specification of their types. Finally, the *formula* part contains a description of the variables with a tree structure and the expressed predicate.

```
( benchmark data
:datatypes      ((label a b )
                 (String hello everyone )
                 (tree (nil) (IntLeaf (val Int)) (RealLeaf (val Real))
                 (StringLeaf (val String)) (BoolLeaf (val Bool))
                 (cons (firstChildLabel label) (firstChild tree) (SiblingChild tree))))
:extrafuns      ( (x Bool) (y tree) (z tree ) (v1 Int) (v2 Real) (v3 String)
                 (v4 Int) (v5 Real) (v6 tree))
:formula ( and  (= y ( cons a ( IntLeaf v1 ) ( cons b ( RealLeaf v2 ) nil )))
                 (= z ( cons a ( StringLeaf v3 )( cons b ( cons a (IntLeaf v4)
                 (cons b (RealLeaf v5) nil )) nil )))
                 (= v6 ( cons a ( IntLeaf v4 ) ( cons b ( RealLeaf v5 ) nil )))
                 (= x true )
                 (= v2 ( + 8.2 4.0 ) )
                 (= v6 y ))
)
```

Figure 4.8: The Z3 input file for the example.

As response to the input file, the Z3 solver provides a satisfiability response preceded by an instance of the variables in case the predicate is sat. The response of the solver for our example is given in Figure 4.9.

In our example the predicate is satisfiable that means that the solver finds for each variable a corresponding value while respecting the conditions of the predicate. For the variables that were not used by the predicate such as the integer variable v_6

```
x   -> true
v2  -> 61/5
y   -> (cons a(IntLeaf 0)(cons b(RealLeaf 61/5)nil))
v1  -> 0
z   -> (cons a(StringLeaf hello)(cons b(cons a(IntLeaf 0)   } //Variable
       (cons b(RealLeaf 61/5)nil))nil))                     } //instantiation
v3  -> hello
v4  -> 0
v5  -> 61/5
v6  -> (cons a(IntLeaf 0)(cons b(RealLeaf 61/5)nil))

sat                 // Satisfiability response
```

Figure 4.9: The Z3 solver response.

or the real (rational) variable $v2$ in our example the solver assigns a default value depending on its type. At the end, we find (Figure 4.10) and returns: $\{\ x = true$, $y = \{a = 0, b = 61/5\}$ and $z = \{a = hello, b = \{a = 0, b = 61/5\}\}\}$.

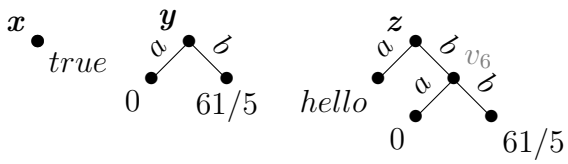

Figure 4.10: Variables realization

4.3 Conclusion

After the theoretical description of our approach in Chapter 3, within this chapter we have presented some technical implementation detail and tool support associated to our framework. We described the functionalities of the used tool chain using algorithms all along the different steps of the approach.

We reuse the Figure 4.11 to summarize the different steps of our approach that are supported by the tools. In step 1: from a specification represented as a BPEL file we use the `BPELtoSTS` (4220 lines in Java) tool to generate an WS-STS model. Followed by step 2: the produced model and a test purpose WS-STS (in the same format) are

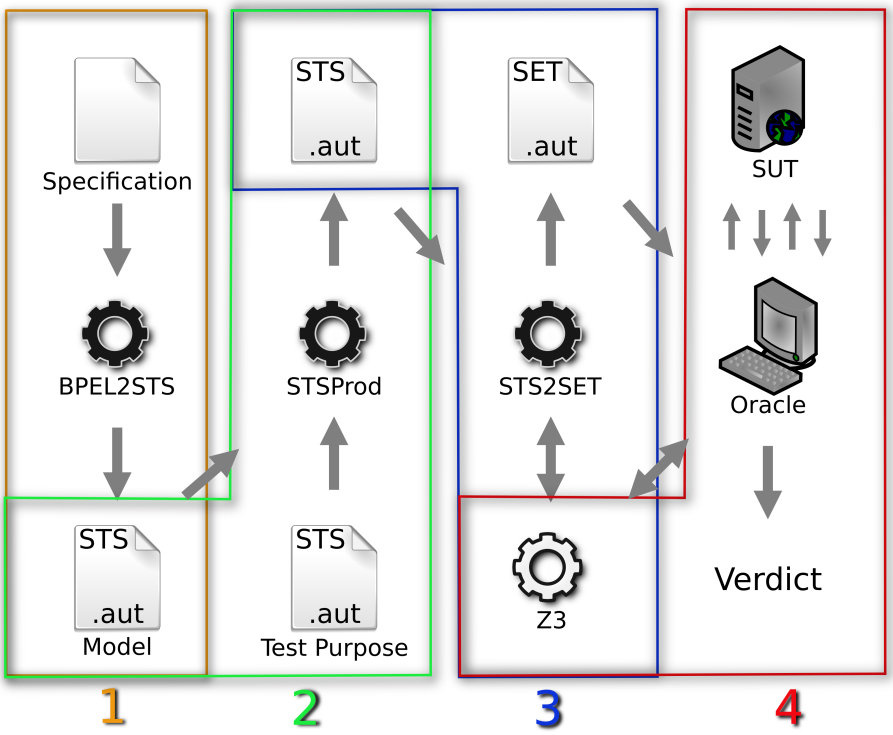

Figure 4.11: Overview of the tools chain

provided as inputs files to our `STSprod` tool (436 lines in Java) that computes the product. In step 3: the `STS2SET` (2098 lines in Python) tool takes as input the STS product and uses the Z3 solver to compute the SET. We have implemented only the path length criterion. Currently, we are working on implementing the `STS2SET` tool as a java plugin for Eclipse. Finally in step 4: using the SET path we execute the test cases as described in the test cases realization. This step is done manually. The test cases are exercised against an implementation, the service under test (SUT), which is not isomorphic to the specification.
In order to provide graphical representation of the STSs and the SET, we used intermediate formats following the Aldebaran format (.aut) which supports a graphical representation with the CADP tool [62].

CHAPTER 5

Conclusion

The Service Oriented Architecture (SOA) paradigm is changing the style of developing software applications. It allows more flexibility, especially with the use of loosely coupled software modules. A widely used implementation of this paradigm are Web services. The reliability of these systems is an imperative property that must be considered as a standard. However, such a guaranty implies tedious and sometimes expensive work. For these reasons, many research works explored the possibility of automating the testing and verification activities.

In this thesis we have focused on testing the conformance of service orchestrations with a black box approach. In the present chapter we present the achievements in our work and discuss future work to perform and toolkit to improve.

5.1 Contributions

In chapter 2, we have presented our symbolic approach for the generation of symbolic test cases, with the intention of applying conformance testing on a centralized composite service.
Given an ABPEL specification of a service orchestration, we generate its associate WS-STS model using our transformation rules. Then we compute the symbolic execution tree (SET) from the WS-STS using symbolic execution. The use of a symbolic model together with symbolic execution enables us to avoid state explosion issues in presence of data handled by Web services. Each path of the SET is associated to a path condition thus, constituting a (symbolic) test case. The realization of these symbolic test cases is achived thanks to the Z3 SMT solver. After that we execute

test cases by interacting step by step with the orchestration implementation. At the end, we emit a verdict about the conformance of the implementation regarding to its specification.
In our approach, we also offer the possibility to guide the test cases generation by supporting test purposes (TP). The specific behavior to be tested, depicted by the TP, is incorporated to the specification model using a WS-STS product. Then, the remaining testing process corresponding to the generation of the SET, the realization of the test cases then their execution against the implementation stays unchanged.

The chapter 3 was a presentation of the tool chain that supports our approach. The transformation of the ABPEL specification into a WS-STS model is done using the `BPEL2STS` prototype. This prototype was developed within the Project ANR *Pervasive Service Composition*[1] (PERSO).
When a specific test scenario is defined then the `STS-Product` prototype computes the product between the WS-STS specification and the WS-STS of the TP, thus providing a new STS model which describes the specification while taking into account the test purpose. The `STS2SET` tool is in charge of computing the SET tree from which test cases are deduced. Another tool is used for the realization of the test cases by interacting with the Z3 SMT solver.
For the moment, the last step that consists in executing the test cases is performed manually by (i) using the soapUI tool, to interact with the implementation, and (ii) the Z3 solver to check the satisfiability of the implementation response. This part is currently being implemented. Finally a verdict is emitted on the conformance of the implementation with regard to its specification. We applied our symbolic approach on two medium size cases studies.

5.2 Perspectives

Besides the contribution presented above, some perspectives are still to be explored. We can split those perspectives as research vs. tool perspectives.

Among the research perspectives, there is the support of additional activities of BPEL, such as compensation and termination handlers. This may impact the WS-STS model, and accordingly the transformation rules and the online testing algorithm.
It would also be interesting, following the domain specification language paradigm, to define a specific language for test purposes instead of providing them as STS models. Another perspective that should be investigated is to consider the UML4SOA profile modeling language or the SENSORIA Reference Modelling Language (SRML) to specify service compositions.

[1] $http://www.lri.fr/\ poizat/ANR_PERSO/index.html$

Since we have applied our approach on a restricted testing architecture, another perspective is to apply it on an in-the-large testing architecture, where the tester simulates the user(s) but also the service partners. This latter perspective would also enable us to apply our approach on a choreography of services in which the business process is no longer managed by a unique service.

A first tool perspective is the automation of the online test execution process where the tester interacts with the service implementation. This is under process. A second tool improvement would be the integration of a solver for String constraints such as the HAMPI [83] solver, which supports constraints over fixed-size String variables, without requiring to given String constraints.

APPENDIX A

x-Loan Case Study

In this section we present our xLoan case study. It is a modified/extended version of the Loan approval composition exposed within the WS-BPEL standard [117], which usually serves for demonstration purposes in articles on BPEL verification. Our extensions are targeted at demonstrating our support for BPEL important features: complex data types, complex service conversations including message correlation, loops and alarms. Hence, more complex and realistic data types are used, to model user information, loan requests and loan proposals.

A.1 Specification

The specified sub-services respectively deal with loan approval (**BankService**) and black listing (**BlackListingService**), with users not being blacklisted asking for low loans ($\leq 10,000$) getting loan proposals without requiring further approval. As-is, these services resemble the ones proposed in [117]. Yet, once a loan is accepted, proposals may be sent to the requester. Further communication then takes place, letting the requester select one proposal or cancel, which is then transmitted to **BankService**. If the selected offer code is not correct the requester is issued an error message and may try again (select or cancel). Timeouts are also modelled, and the bank is informed about cancelling if the requester does not reply in a given amount of time (2 hours). The figure A.1 represent an overview of the xLoan conversation protocol.

There is no official graphical notation neither for orchestration architectures (WSDL interfaces and partner links), nor for the imported data types (XML Schema

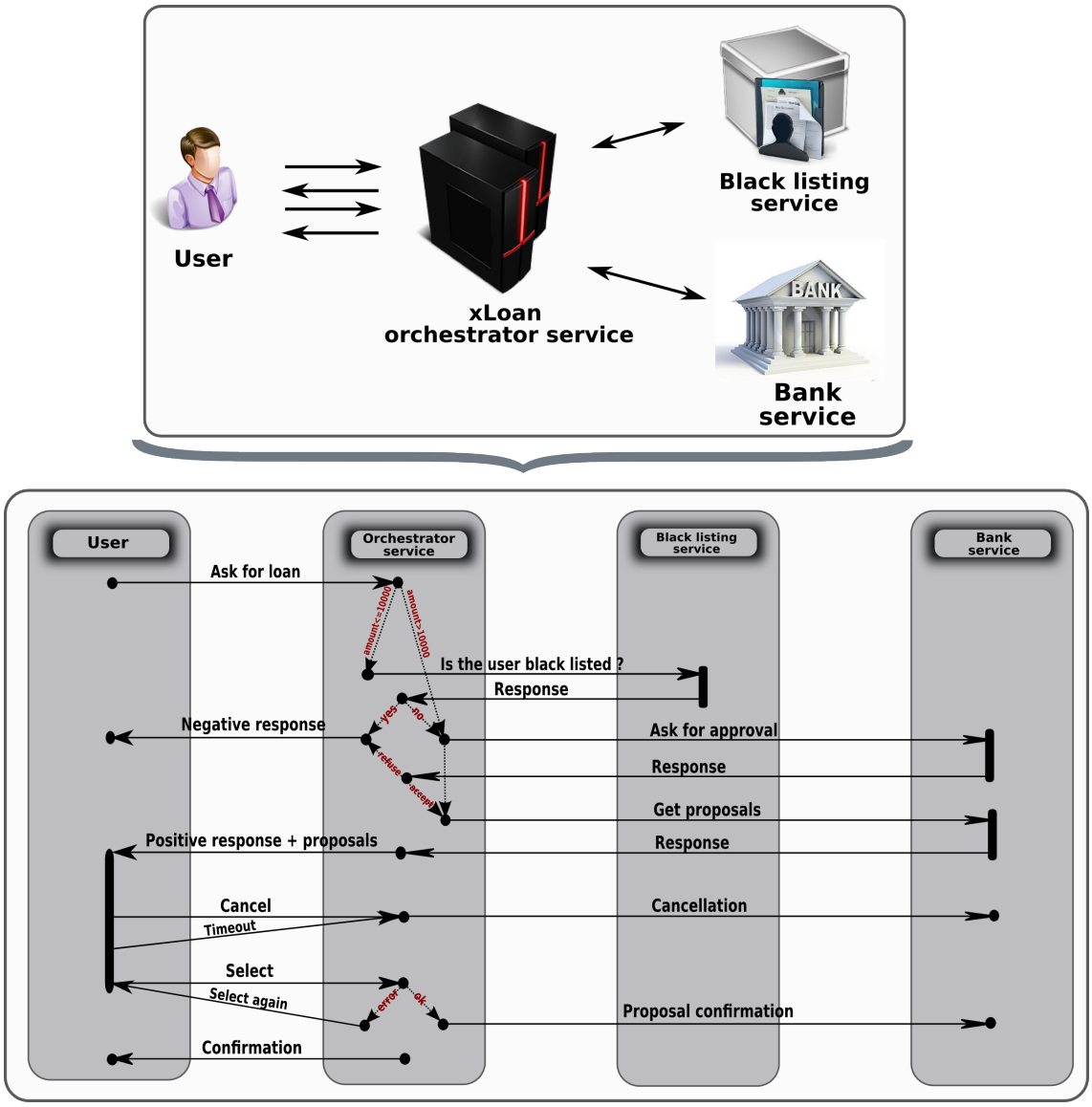

Figure A.1: **xLoan** Example – Business process

files) or the service conversation (BPEL <**process**> definition). For the former ones (Fig. A.2) we use the UML notation that we extend with specific stereotypes in order to represent message types, correlations and properties. Moreover, XML namespaces are represented with packages. Additionally, there is currently an important research effort on relating the Business Process Modelling Notation (BPMN) with Abstract BPEL or BPEL code. Therefore, concerning the graphical presentation of service conversations, we take inspiration from BPMN, while adding our own annotations supporting relation with BPEL. Communication activities are represented with the

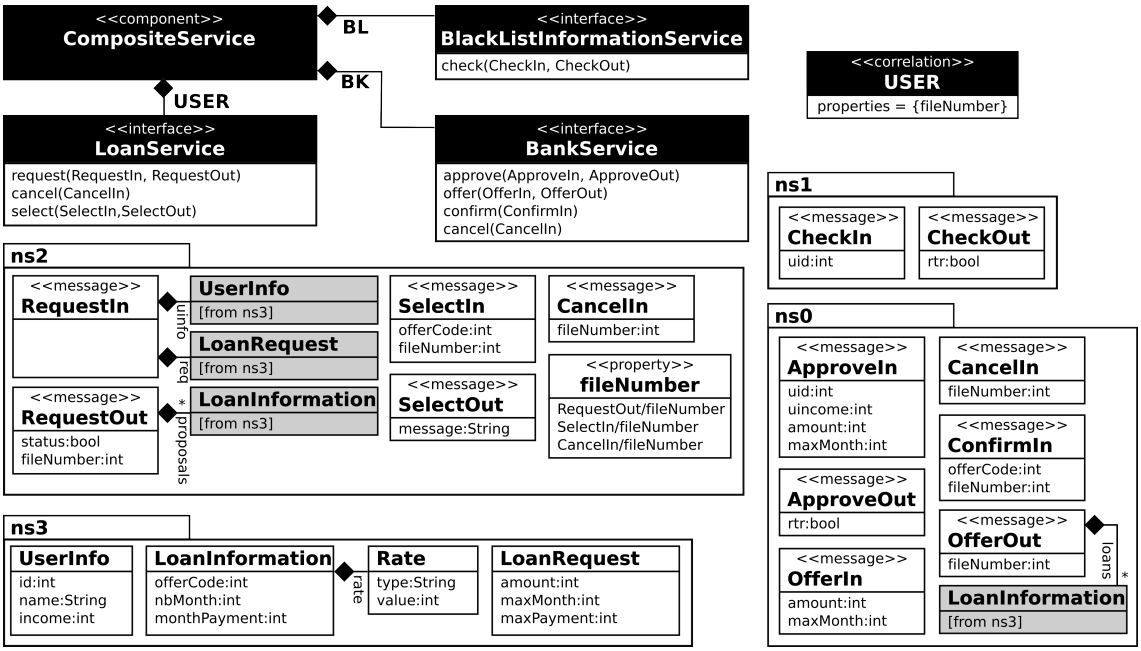

Figure A.2: **xLoan** Example – Data and Service Architecture

concerned partnerlink (**USER** for the user of the orchestration, **BK** or **BL** for the two sub-services), operation, input/output variables, and, when it applies, information about message correlation.

Figure A.3 presents the orchestration specification. The overall process is presented in Figure A.3, upper part, while its lower part concerns the (potentially looping) subprocess, **GL&S** (*Get Loan and Select*), for loan proposal selection.

A.2 WS-STS model

The STS obtained from the xLoan example presented is presented in Figure A.4 where **tau** (resp. **tick**, **term**) denote τ (resp. χ, $\surd$). The zoom corresponds to the while part. One may notice states 16 (while condition test), 17/33 (pick), 34 (onAlarm timeout), and 18/23 (correlation testing).

A.3 Symbolic Execution Tree

The SET computed from the Figure A.4 STS is presented in Figure A.5 and in Figure A.6. There are 10 leaves corresponding to termination (in gray). The zoom presents the path (in black) we use for demonstration in the next Section. Its final node is number 305, and its path condition, π_{305} is also given in the Figure.

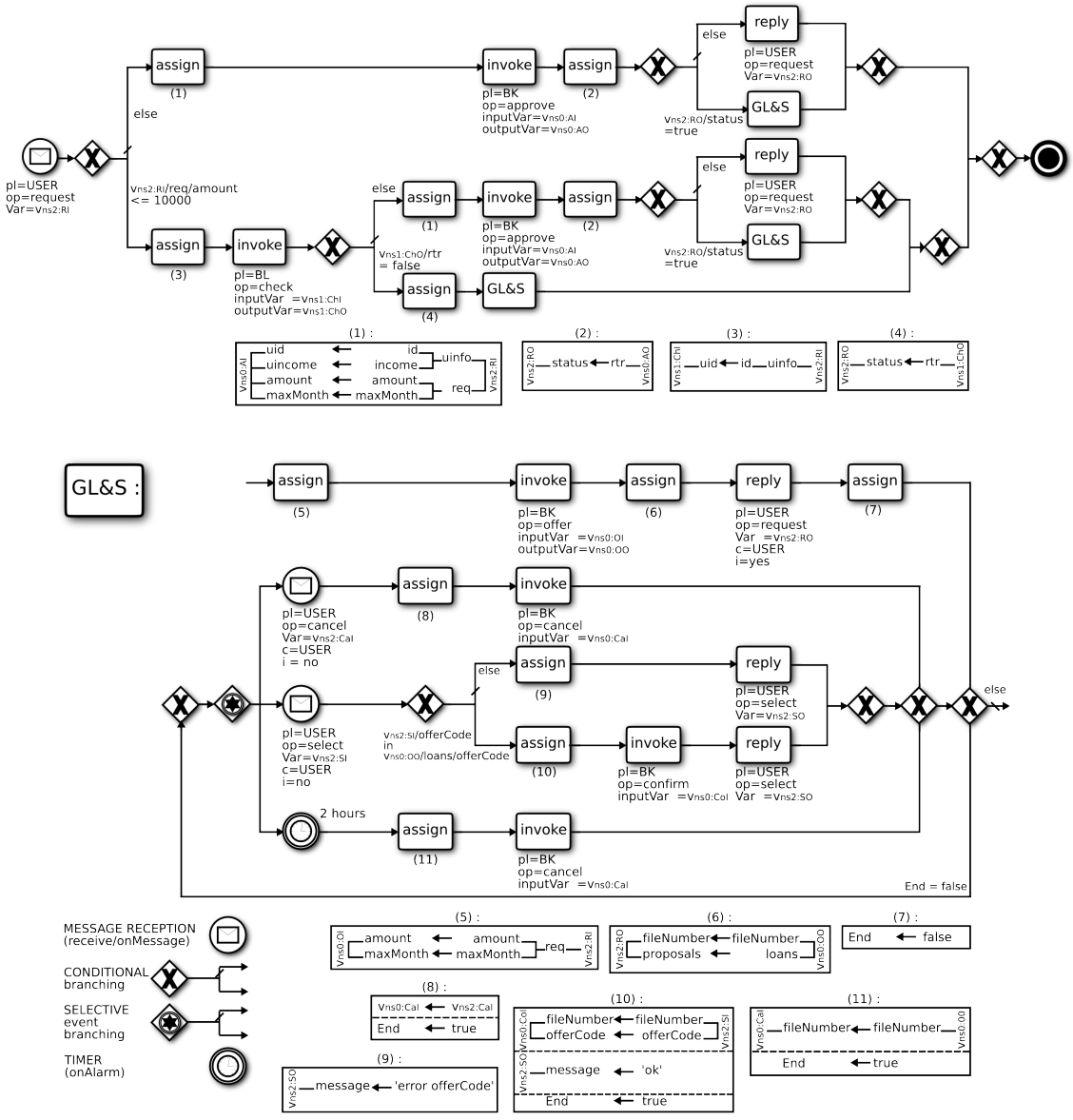

Figure A.3: **xLoan** Example – Orchestration Specification

A.4 Symbolic Test Cases

In this part we present our online testing approach using the UML2CSP tool [38]. Our approach is automated by means of prototypes written in the Python language that serve as a proof of concept. As far as constraint solving is concerned, we chose the UML2CSP tool [38], which supports OCL constraint solving over UML class diagrams,

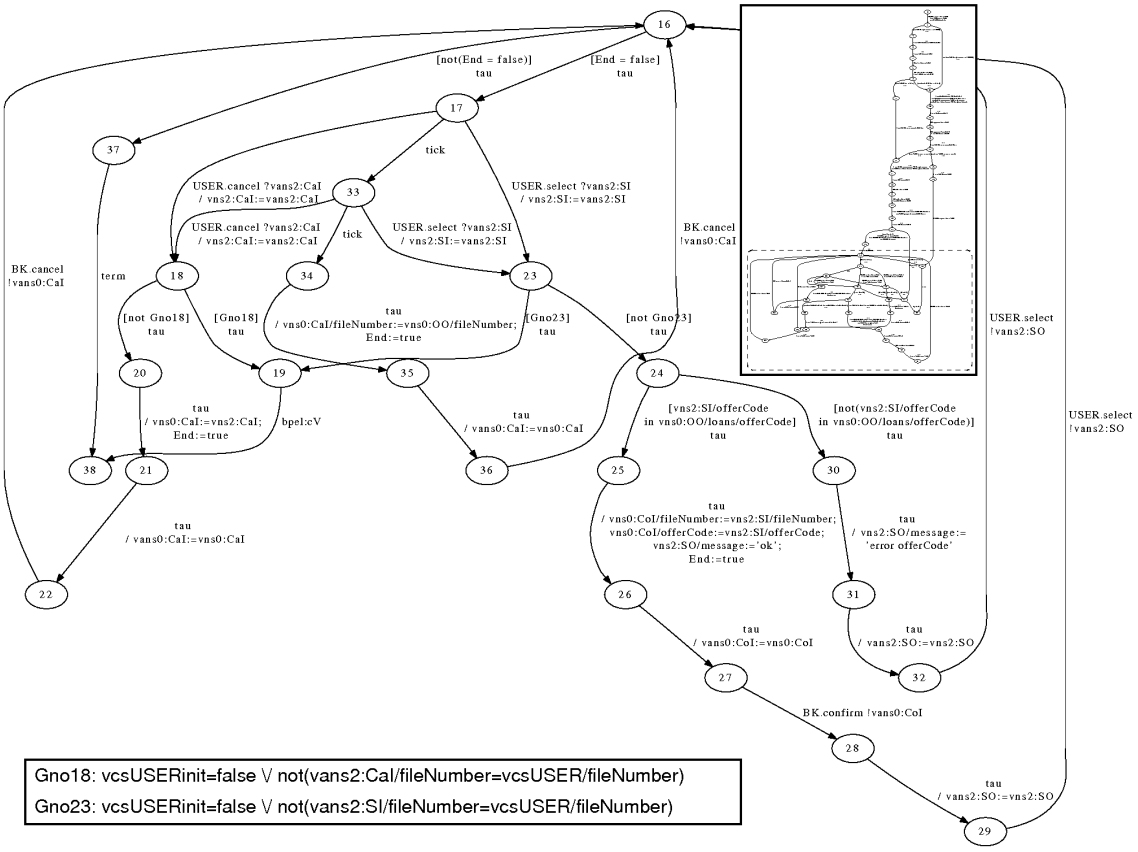

Figure A.4: **xLoan** Example – Symbolic Transition System

that correspond to the constraints we have on XML schema data. Additionally, UML2CSP is able to generate witness object diagrams when the constraints are satisfiable. More precisely, in order to reuse this tool, we proceed as follows:

- we translate XML schema definitions into an UML class diagram, see Figure A.2. Additionally, domain limits are set up in UML2CSP according to uniformity hypotheses, *e.g.*, here we have set **maxMonth:{12,24,36}** and **maxPayment:[1000..100000]**). This step is done only once.

- to check if some π is satisfiable *before* sending a message, an additional root class (**Root**) is created wrt. the UML diagram, with as many attributes as symbolic variables in π. The π constraint is translated in OCL. If π is satisfiable, UML2CSP generates an object diagram. From it we get data for the variable of interest to be sent (step $val := (SOLVE(\pi)[x_s])$ in the online Algorithm).

- to check if some π is satisfiable *after* receiving a message, we perform as before,

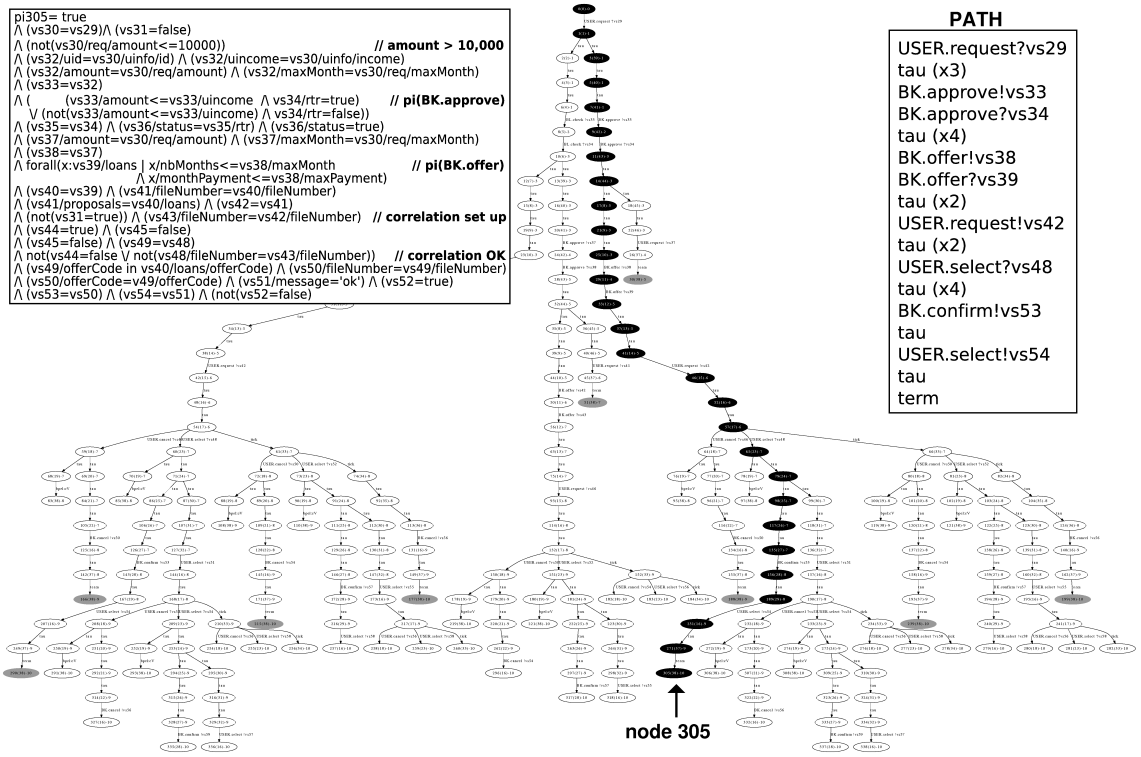

Figure A.5: **xLoan** Example – Symbolic Execution Tree (k=10, τs not counted)

but adding an OCL constraint enforcing that the symbolic variable of the reception is equal to the data effectively received (steps $\pi = \pi \wedge (x_s = val)$ and $\neg SOLVE(\pi)$ in the online Algorithm).

- cutting infeasible paths in the SET computation is a sub-case of satisfaction before message sending (the generated object diagram is discarded).

- strings are treated as integers which represent an index in an enumerated type. This corresponds to a set of specific string constants for the test.

For the time being, interaction with UML2CSP is manual. The automation of this step is under process as part of an Eclipse plug-in we are developing.

Experiments have been applied on an implementation of **xLoan** which is not isomorphic to its specification as, *e.g.*, the BPEL code relies on additional boolean variables rather than on the workflow structure to decide if the subprocess for loan proposal selection or cancelling is applicable. The implementation size is 246 lines long (186 XML tags). In order to demonstrate our online algorithm, we take one of the 10 complete paths we have in the SET (see Fig. A.5). Due to lack of room we focus on the first interaction steps of the path (loan request, loan reply). The first call

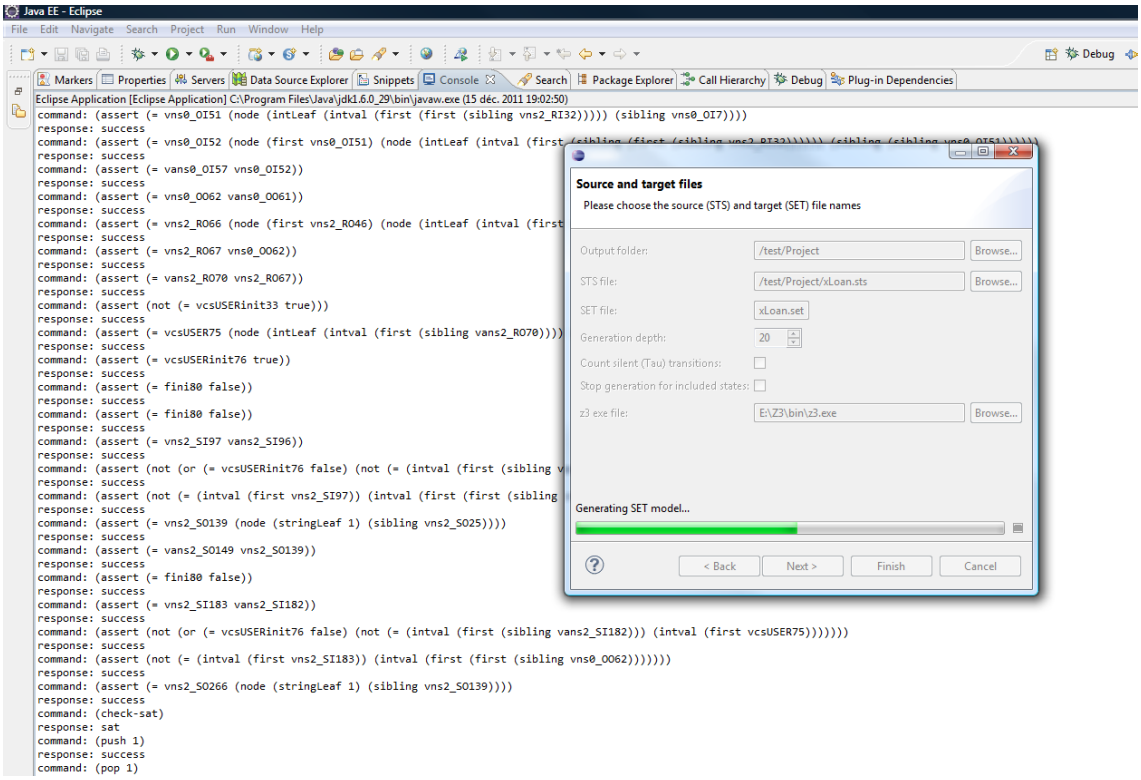

Figure A.6: **xLoan** Example – Symbolic Execution Tree - Overview of the STS2SET tool

to UML2CSP with the end path condition, π_{305}, enables one to retrieve values for the **RequestIn** message (id, name, income, amount, maxMonth, maxPayment), *e.g.*, the part of the path condition relative to the requested amount generates the `Context Root inv PC : not(self.vs30.req.amount<=10000)` OCL constraint, and value 10001 for the request amount. We then generate the message data in Figure A.7, left, and send it using SoapUI[1]. The corresponding received message is in Figure A.7, right. We translate this data as an OCL constraint and solving it with UML2CSP we are able to show that it is a correct output. We may then proceed generating data for a correct offer selection (**offerCode=1**).

We also used the solver SMT Z3 with the xLoan examples as shown in Figure A.8.

[1] `http://www.soapui.org/`

```
<soapenv:Envelope xsi:...="http:... >
<soapenv:Body>
<ns2:RequestIn>
  <ns3:uInfo>
    <id>1</id>
    <name>Simpson</name>
    <income>10002</income>
  </ns3:uInfo>
  <ns3:req>
    <amount>10001</amount>
    <maxMonth>12</maxMonth>
    <maxPayment>1000</maxPayment>
  </ns3:req>
</ns2:RequestIn>
</soapenv:Body>
</soapenv:Envelope>
```

```
<soapenv:Envelope xsi:...="http:... >
<soapenv:Body>
<ns2:RequestOut>
  <status>true</status>
  <fileNumber>1</fileNumber>
  <ns3:proposals>
    <offerCode>1</offerCode>
    <nbMonths>12</nbMonths>
    <monthPayment>918</monthPayment>
    <ns3:rate>
      <type>fixed</type>
      <value>10</value>
    </ns3:rate>
  </ns3:proposals>
</ns2:RequestOut>
</soapenv:Body>
</soapenv:Envelope>
```

Figure A.7: **xLoan** Example – a Sent and a Received Message (parts of)

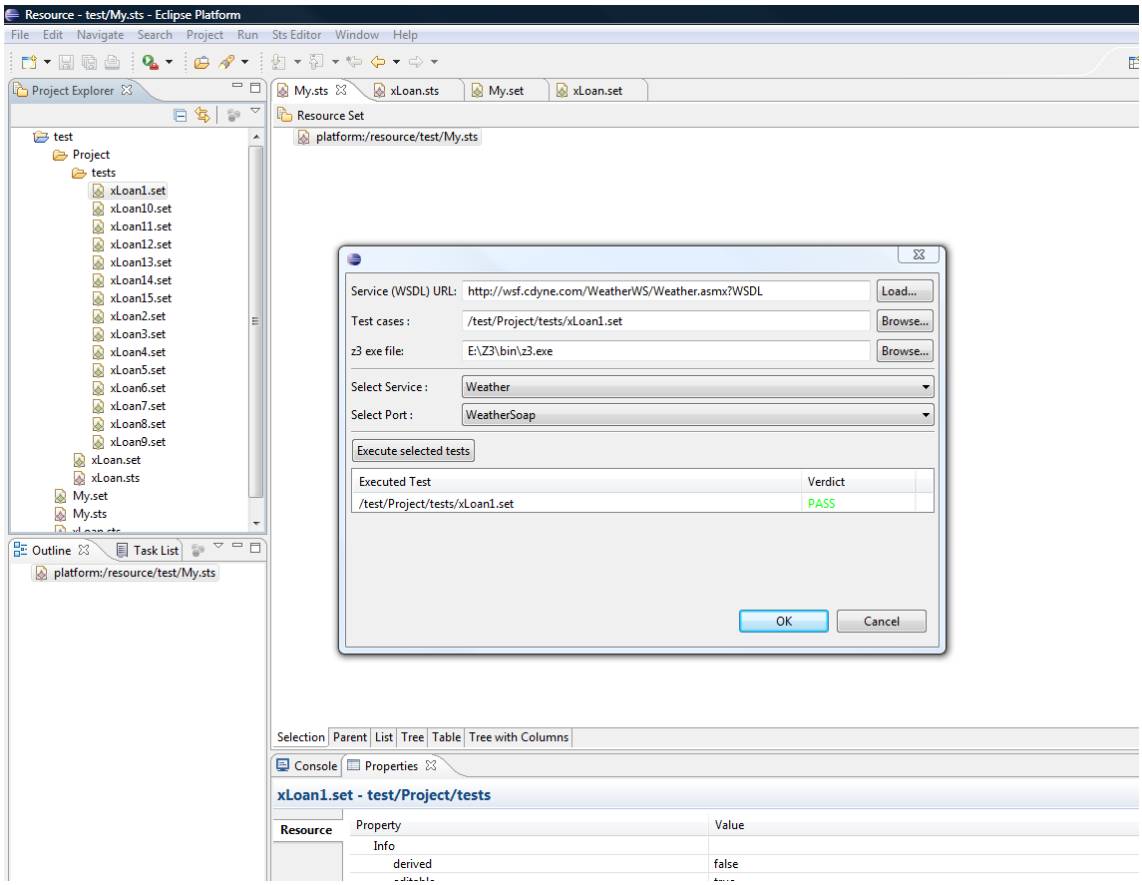

Figure A.8: **xLoan** Example – Execution of a test case

APPENDIX B

E-Conference Case Study

In this section we present our e-Conference case study. This medium-size orchestrated service provides functionalities to researchers going to a conference such as information about the conference, flight booking and fees refunding. Parts of the orchestration, *e.g.*, the e-governance rules, correspond to reality. Other parts, *e.g.*, the sub-service used to book plane tickets, represent a simplified yet realistic version of it.

B.1 Specification

e-Conference is based on three sub-services: ConferenceService, FlightService, and e-govService as presented in the Figure B.1. Its specification is as follows. A user starts the process by providing the conference name and edition, together with personal information (ordersSetup operation). Conference information is first retrieved using ConferenceService then e-govService is invoked to check if any travel alerts exist in the conference country. If there is one, the user is informed and the process stops. If not, an orders id is asked to e-govService, a plane ticket is bought using FlightService and all gathered information (orders id, conference and plane) is sent back to the user who may then go to the conference. Upon return, the user may choose to be refunded from either a fees or package basis (returnSetup operation). In both cases, the user will end by validating the mission (validate operation) and e-Conference will reply with refunding information. If fees basis has been chosen, the user will be able to send information on fees (using addFee operation several times).

Figure B.2 exhibits our extension of the UML notation corresponding to the e-Conference orchestration. Within this diagram we highlight the stereotypes for

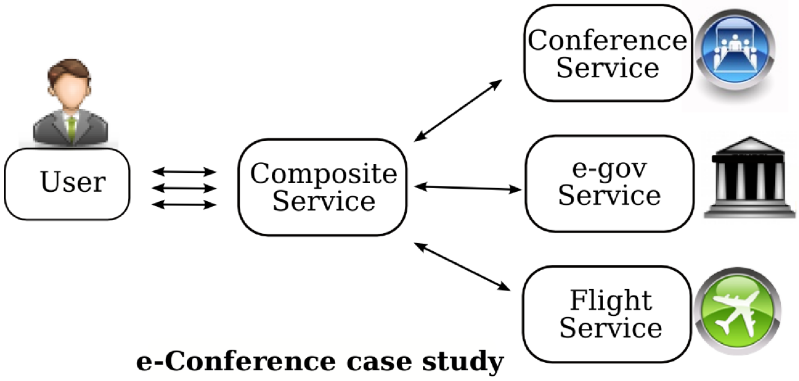

Figure B.1: e-Conference example

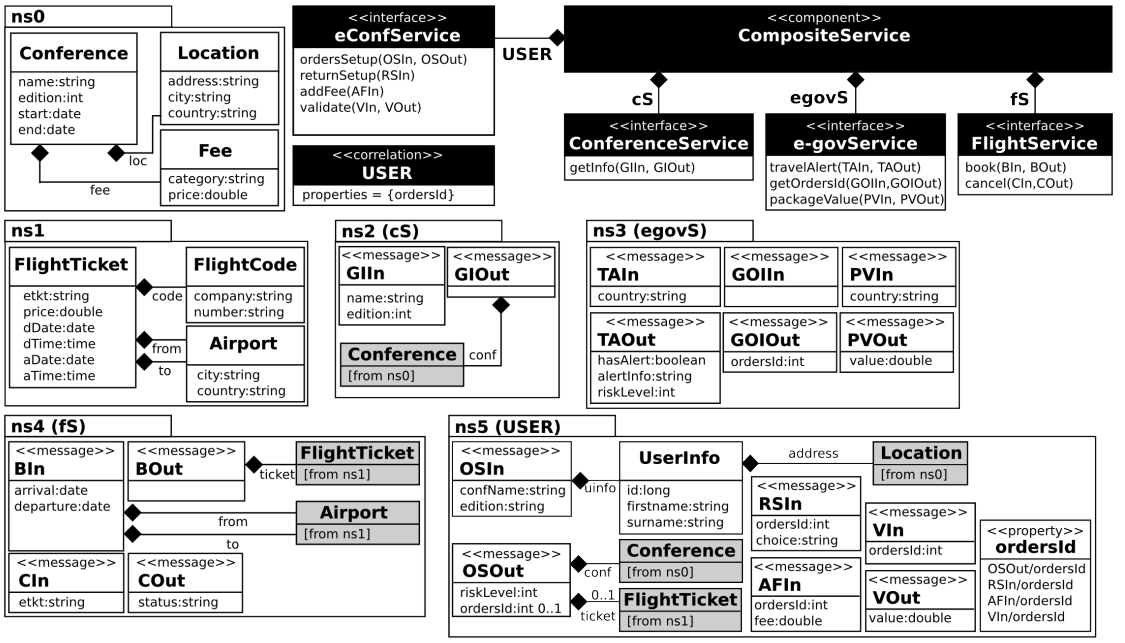

Figure B.2: e-Conference Example – Data and Service Architecture (UML Extended)

message types, correlations and properties to represent the orchestration architecture and the imported data (XML schema files structured in namespaces ns_x). Concerning orchestration specification shown in Figure B.3, we take inspiration from BPMN, while adding our own annotations supporting relation with BPEL. Communication activities are represented with the concerned partnerlink (USER for the user, cS, egovS, and fS for the sub-services), operation, input/output variables, and, when it applies, information about message correlation.

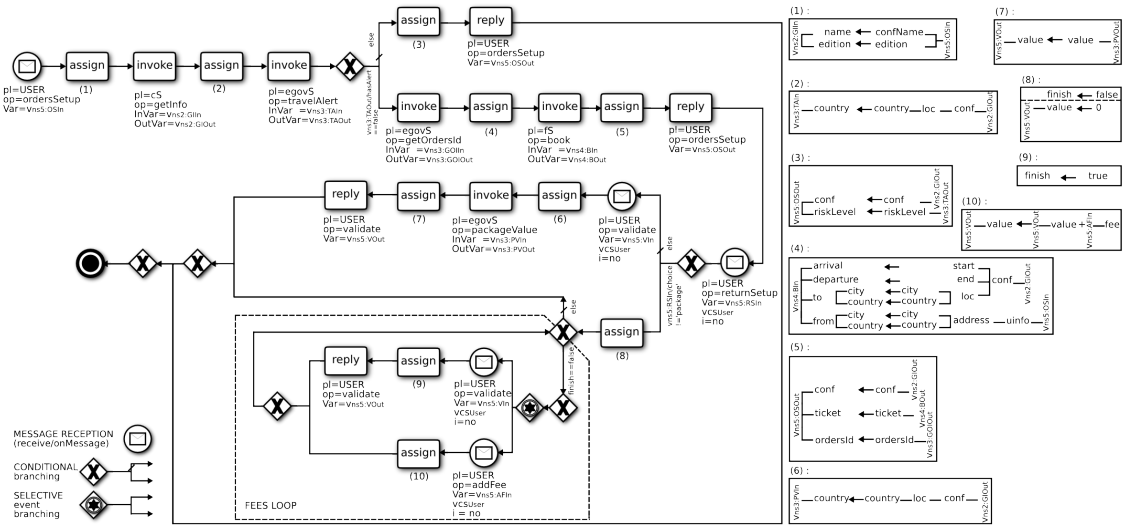

Figure B.3: e-Conference Example – Orchestration Specification (Inspired and extended from BPMN)

B.2 WS-STS model

From the e-Conference specification (see Figure B.3), we obtain the STS in Figure B.4 (49 states, 57 transitions) where tau (resp. term) denote τ (resp. $\surd$). The zoom (grey states) corresponds to fees loop. One may notice states 39 (while condition test) and 40 (pick). In states 41/45 it is checked if incoming messages (validate or addFee) come from the same user than the previous ones in the conversation (ordersSetup and returnSetup). When it is not the case (correlation failure) an exception is raised (in state 26). Variables names, in our example are prefixed with namespaces (*e.g.*, vns5:VOut is the variable storing values of type ns5:VOut) to help the reader.

B.2.1 Test purpose

Let us assume one wants to focus on testing refunding on a fees basis. Possible TPs are given in Figure B.5. In the first TP, one may note the use of the addFee transition, leading to an acceptance state (Labelled with #). This specifies addFee must be part of the generated test cases. Several transitions in the specification could be done before and after the addFee one, therefore, the two TP states are equipped with a * loop. The * loop is interpreted as being the execution of any other communication messages (including calls to sub-services), except those explicitly expressed. The first TP requires that there is at least one addFee in the test cases. One may want to take also into account the case where there are none. This can be

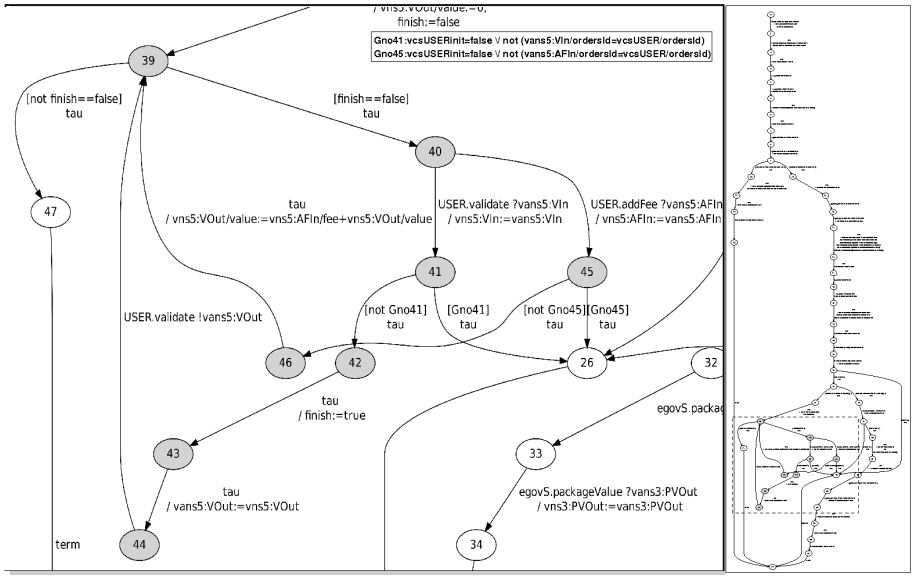

Figure B.4: e-Conference Example – Orchestration model (STS)

specified as in the second TP. There, one relies on the returnSetup transitions that carry user requests relative to the return mode (package or fees basis). In order to specify it is the later which is required, a guarded transition is used (choice should be different from 'package'). Note that the guard is put on a τ transition after the reception in order to be consistent with (symbolic) execution semantics: the guard can only be evaluated once the variable has been received. The last TP, used in the sequel, is more realistic and demonstrates TPs expressiveness with four requirements: (i) return is on a fees basis, (ii) each fee is greater or equal to 10, (iii) the total amount for the mission is less than 1000, and (iv) there are at most N fees added. The later requires to use a TP additional variable, n, to count addFee iterations.

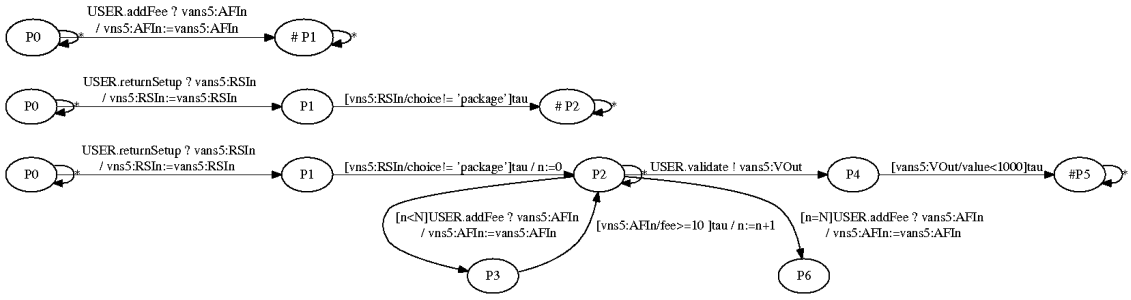

Figure B.5: e-Conference Example - Test Purposes (STS)

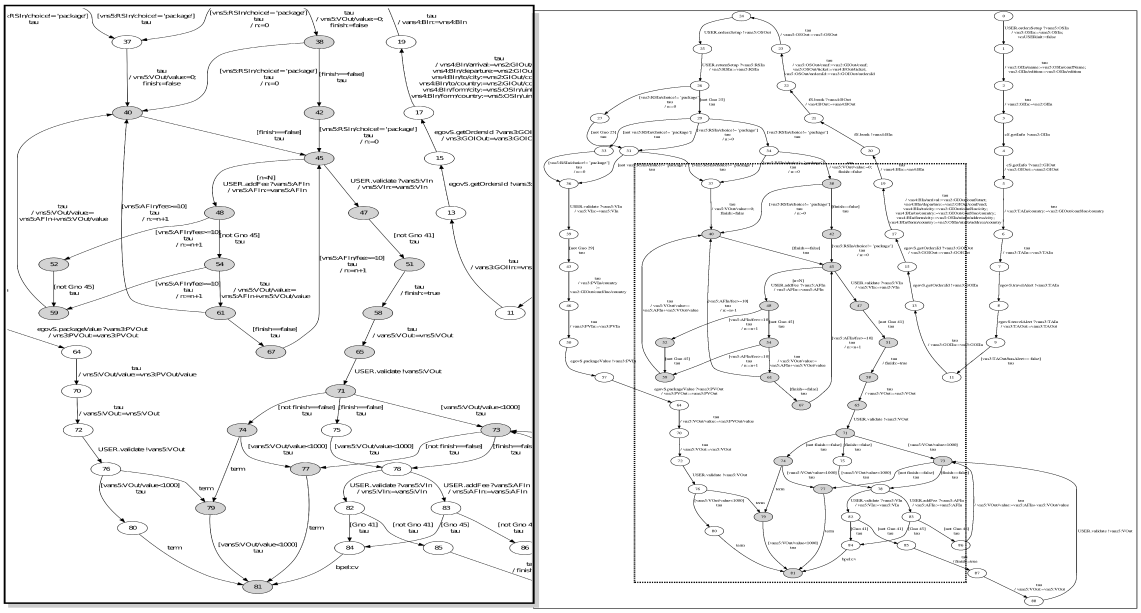

Figure B.6: e-Conference Example - Product (STS)

B.2.2 The product model

The product of the orchestration (Figure B.4) with the third TP (Figure B.5) is given in Figure B.6. It has 68 states and 85 transitions (89 states and 119 transitions before pruning). Its set of symbolic traces is a subset of TP one (hence also of the orchestration one). One may note for example that receiving addFee is possible only if done less than N times (see guard [n<N] in the transition outsourcing from state 45), and that the condition on fee values is also taken into account (states 48/54/61/67).

B.3 Symbolic Execution Tree

Among the 85 paths in the SET ($k = 15$) of the product STS (see Figure B.6), there are 7 complete paths corresponding to the coverage of the last test purpose described in Figure B.5. A path example is:

```
USER.ordersSetup?vs36 tau(x2) cS.getInfo!vs40
cS.getInfo?vs41 tau(x2) egovS.travelAlert!vs44
egovS.travelAlert?vs45 tau(x2) egovS.getOrdersId!vs47
egovS.getOrdersId?vs48 tau(x2) fS.book!vs51
fS.book?vs52 tau(x2) USER.ordersSetup!vs55
USER.returnSetup?vs58 tau(x5) USER.addFee?vs69
```

```
tau(x4) USER.validate?vs74 tau(x3) USER.valivate!vs77
tau(x2) term.
```

B.4 Symbolic Test Cases

For the example e-Conference, we have defined five test purposes that allow to cover the different functional behaviour of the service. One is related to a demand of a user with a country having a high alert. Two others are related to an order with a country with no alert and with either a fees or package refund. The two last are related to orders with fees refund and a variation on the number of addfee. For each test purpose, we have produced several test cases, and only one can be executed with an uniformity hypothesis on the different values of the variables. To evaluate the quality of our tests, we need to apply mutations on the implementations. We will in our future work consider mutants in order to check if our test cases are able to kill the mutants and hence evaluate the quality of our generated tests.

In this work, we focus on generating test cases and solving constraints in order to first compute the SET then use the instantiate data returned by the solver to interact with the implementation. The interaction with the implementation can be done using the soapUI tool.

Appendix

In this appendix, we provide the algorithms for the generation of trees for the variable and the interaction with the Z3 SMT solver using pseudo-code.

C.1 The steps:

The Z3problem allows :

1. Building trees for each given variable according to its type

2. Processing the constraint (giving as an input) to generate an input file for the Z3 solver

3. Sending, then receiving the answer of the solver about the satisfiability of the constraint and its variables

C.1.1 Inputs:

The Symbolic Web Service Testing (SWST) tool takes as inputs: a set of Variables V and a Constraint C

The first parameter (V) is represented as an Array that contains the initial variables (for the moment). The second parameter, (C) is represented as a conjunction of clauses in which $XPath$ expressions are used.

C.1.2 Outputs:

As a result the SWST tool provides an answer about the satisfiability of the constraint according to the variables types.
This result is represented as a couple $(Response, M)$. Where the different responses could be:

- $Unsat$, means that the Z3 solver could not find an instantiation of the variables that satisfies the constraint
- $Unknown$, means that due to a lack of information
- Sat, means that the Z3 solver found at least one instantiation for the variables in the way that satisfies the constraint

The M parameter contains the instantiation of the variables when the response is Sat. Otherwise M is empty.

C.2 Description of the functions

C.2.1 The main functions

The Sequencing of our tool is : Creation of the trees for each variable, the new leaf variables, then handling the creation of new variable clauses VC. For sake of simplicity we only consider the basic type integer.

C.2.2 Defining the functions

Algorithm 15: **Creation of the trees for each variable**

Input: a set of Variables V
Output: Creation of the trees for each variable of V

1 **foreach** $v \in V$ **do**
2 $\quad type_v = v.type$
3 $\quad$ Tree $t = createTreeFromType(type_v)$
4 $\quad v.value = t$
5 $\quad t.variable = v$
6 **end**

Algorithm 16: **createTreeFromType**

Input: Type t
Output: Tree

```
1  if t.name == "IntType" then
2  |  IntLeaf l = new IntLeaf()
3  |  return l
4  end
5  else
6  |  Node n = new Node()
7  |  foreach st ∈ t.subTypes do
8  |  |  SubTree sn = new SubTree()
9  |  |  sn.field = st.field
10 |  |  sn.subTree = createTreeFromType(st.subType)
11 |  |  sn.parent = n
12 |  |  sn.subTree.parent = sn
13 |  |  n.children = n.children ∪ {sn}
14 |  end
15 |  return n
16 end
```

Algorithm 17: **the new leaf variables**

Input: a set of Variables V
Output: a set of variables V within the new leaf variables

```
1  VA = ∅
2  foreach v ∈ V do
3  |  if v.type.name ≠ "IntType" then
4  |  |  VA = VA ∪ createNewLeafVars(v.type, v.value)
5  |  end
6  end
7  return V ∪ VA
```

Algorithm 18: **createNewLeafVars**

Input: Type t, Tree val
Output: a set of variables VA

```
1  VA = ∅
2  if t.name = "IntType" then
3  |  Variable vnew = new Variable()
4  |  vnew.name = "v" + num
5  |  num++
6  |  vnew.type = t
7  |  vnew.value = val
8  |  val.variable = vnew
9  |  t.toVar = t.toVar ∪ {vnew}
10 |  VA = VA ∪ {vnew}
11 end
12 else
13 |  foreach st ∈ t.subTypes do
14 |  |  field = st.field
15 |  |  nvtype = st.subType
16 |  |  nvval = val.getSubTree(field, val)
17 |  |  VA = VA∪ createNewLeafVars(nvtype, nvval)
18 |  end
19 end
20 return VA
```

Algorithm 19: **the new VC variables**

Input: a set of Variables V, a constraint C
Output: a set of variables V within the new VC variables

```
1  foreach c ∈ C.clauses do
2  |  V = V ∪ createNewVCVars(V, c.leftPart)
3  |  V = V ∪ createNewVCVars(V, c.rightPart)
4  end
5  return V
```

Algorithm 20: **createNewVCVars**

Input: set of variables V, ALE e
Output: a set of variables : V

```
1  if e is an IntConstant then
2      return V
3  end
4  else
5      if e is an Operation then
6          V = createNewVCVars(V, e.leftPart)
7          V = createNewVCVars(V, e.rightPart)
8          return V
9      end
10     else
11         if e.path ≠ 0 then
12             if ¬hasVariableForPath(e.vaiableRoot.value, e.paht) then
13                 Variable vnew = new Variable()
14                 vnew.name = "v" + num
15                 num++
16                 vnew.type =
                   computeVarTypeFromVarRoot(e.variableRoot.type, e.path)
17                 vnew.type.toVar = vnew
18                 vnew.value =
                   computeVarValueFromVarRoot(e.variable.value, e.paht)
19                 vnew.value.variable = vnew
20                 e.varaiable = vnew
21                 return V ∪ {vnew}
22             end
23             else
24                 e.variable =
                   findVariableForPath(e.variableRoot.value, e.paht)
25                 return V
26             end
27         end
28         else
29             e.variable = e.variableRoot
30             return V
31         end
32     end
33 end
```

Algorithm 21: **hasVariableForPath**

Input: Tree n, a list of paths p
Output: Boolean

```
1 if p = [] then
2 |   return (n.variable ≠ null)
3 end
4 else
5 |   Node n' = new Node()
6 |   n' = n.getSubTree(p[0], n)
7 |   return hasVariableForPath(n', p − p[0])
8 end
```

Algorithm 22: **computeVarValueFromVarRoot**

Input: Tree n, a list of paths p
Output: Tree

```
1 if p = [] then
2 |   return n
3 end
4 else
5 |   Node n' = new Node()
6 |   n' = n.getSubTree(p[0], n)
7 |   return computeVarValueFromVarRoot(n', p − p[0])
8 end
```

Algorithm 23: **computeVarTypeFromVarRoot**

Input: Type t, a list of pahts p
Output: Type

```
1 if p = [] then
2 |   return t
3 end
4 else
5 |   ComplexType t' = new ComplexType()
6 |   t' = t.getSubType(p[0], t)
7 |   return computeVarTypeFromVarRoot(t', p − p[0])
8 end
```

Algorithm 24: **findVariableForPath**

Input: Tree n, a list of paths p
Output: Variable

```
1 if p = [] then
2 |   return (n.variable)
3 end
4 else
5 |   Node n' = new Node()
6 |   n' = n.getSubTree(p[0], n)
7 |   return findVariableForPath(n', p − p[0])
8 end
```

Algorithm 25: **getSubType**

Input: String f, ComplexType ct
Output: Type

```
1 foreach st ∈ t.subTypes do
2 |   if st.field == f then
3 |   |   return st.subType
4 |   end
5 end
```

Algorithm 26: **getSubTree**

Input: String f, Node n
Output: Tree

```
1 foreach sn ∈ n.children do
2 |   if sn.field == f then
3 |   |   return sn.subTree
4 |   end
5 end
```

APPENDIX

D.1 WSDL of the xLoan Service Orchestrator

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="LoanService" targetNamespace="http://j2ee.netbeans.org/
       wsdl/LoanService"
3      xmlns="http://schemas.xmlsoap.org/wsdl/"
4      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://j2ee.
           netbeans.org/wsdl/LoanService" xmlns:ns="http://xml.netbeans.org/
           schema/LS" xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype
           " xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:vprop="
           http://docs.oasis-open.org/wsbpel/2.0/varprop" xmlns:ns0="http://j2ee
           .netbeans.org/wsdl/BankService" xmlns:ns5="http://xml.netbeans.org/
           schema/BK" xmlns:ns1="http://xml.netbeans.org/schema/RefSchemaLS">
6      <import location="../WEB-INF/wsdl/BankService/BankService.wsdl"
           namespace="http://j2ee.netbeans.org/wsdl/BankService"/>
7      <types>
8          <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/
               LoanService">
9              <xsd:import namespace="http://xml.netbeans.org/schema/LS"
                   schemaLocation="LS.xsd"/>
10             <xsd:import schemaLocation="../WEB-INF/wsdl/BankService/BK.xsd"
                   namespace="http://xml.netbeans.org/schema/BK"/>
11             <xsd:import schemaLocation="../source/RefSchemaLS.xsd" namespace
                   ="http://xml.netbeans.org/schema/RefSchemaLS"/>
12         </xsd:schema>
13     </types>
14     <message name="requestIn">
15         <part name="part1" element="ns:MT-requestIn"/>
16     </message>
17     <message name="requestOut">
18         <part name="part1" element="ns:MT-requestOut"/>
19     </message>
```

```
20      <message name="cancelIn">
21          <part name="part1" element="ns:MT-cancelIn"/>
22      </message>
23      <message name="selectIn">
24          <part name="part1" element="ns:MT-selectIn"/>
25      </message>
26      <message name="selectOut">
27          <part name="part1" element="ns:MT-selectOut"/>
28      </message>
29      <portType name="LoanServicePortType">
30          <operation name="request">
31              <input name="input1" message="tns:requestIn"/>
32              <output name="output1" message="tns:requestOut"/>
33          </operation>
34          <operation name="cancel">
35              <input name="input2" message="tns:cancelIn"/>
36          </operation>
37          <operation name="select">
38              <input name="input3" message="tns:selectIn"/>
39              <output name="output2" message="tns:selectOut"/>
40          </operation>
41      </portType>
42      <binding name="LoanServiceBinding" type="tns:LoanServicePortType">
43          <soap:binding style="document" transport="http://schemas.xmlsoap.org
                /soap/http"/>
44          <operation name="request">
45              <soap:operation/>
46              <input name="input1">
47                  <soap:body use="literal"/>
48              </input>
49              <output name="output1">
50                  <soap:body use="literal"/>
51              </output>
52          </operation>
53          <operation name="select">
54              <soap:operation/>
55              <input name="input3">
56                  <soap:body/>
57              </input>
58              <output name="output2">
59                  <soap:body/>
60              </output>
61          </operation>
62          <operation name="cancel">
63              <input name="input2">
64                  <soap:body/>
65              </input>
66          </operation>
67      </binding>
68      <service name="LoanServiceService">
69          <port name="LoanServicePort" binding="tns:LoanServiceBinding">
70              <soap:address location="http://localhost:${HttpDefaultPort}/
                    LoanServiceService/LoanServicePort"/>
71          </port>
72      </service>
73      <plnk:partnerLinkType name="LoanService">
74          <!-- A partner link type is automatically generated when a new port
                type is added. Partner link types are used by BPEL processes.
```

```
75              In a BPEL process, a partner link represents the interaction
                    between the BPEL process and a partner service. Each
                    partner link is associated with a partner link type.
76              A partner link type characterizes the conversational
                    relationship between two services. The partner link type
                    can have one or two roles.-->
77          <plnk:role name="LoanServicePortTypeRole" portType="
                tns:LoanServicePortType"/>
78      </plnk:partnerLinkType>
79          <!-- Partie pour la corrlation set  -->
80      <vprop:property name="LS_PROP" type="xsd:long"/>
81          <!-- Partie des property Alias  -->
82      <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
            ns0:offerOut" part="part1">
83          <vprop:query>ns5:fileNumber</vprop:query>
84      </vprop:propertyAlias>
85      <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
            tns:requestOut" part="part1">
86          <vprop:query>ns:fileNumber</vprop:query>
87      </vprop:propertyAlias>
88          <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
                tns:selectIn" part="part1">
89          <vprop:query>ns:fileNumber</vprop:query>
90      </vprop:propertyAlias>
91          <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
                tns:cancelIn" part="part1">
92      <vprop:query>ns:fileNumber</vprop:query>
93      </vprop:propertyAlias>
94
95  </definitions>
```

D.2 WSDL of the Black List Service

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <definitions name="BlackList" targetNamespace="http://j2ee.netbeans.org/wsdl
        /BlackList"
3       xmlns="http://schemas.xmlsoap.org/wsdl/"
4       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5       xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://j2ee.
            netbeans.org/wsdl/BlackList" xmlns:ns="http://xml.netbeans.org/schema
            /BL" xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
            xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
6       <types>
7           <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/BlackList
                ">
8               <xsd:import namespace="http://xml.netbeans.org/schema/BL"
                    schemaLocation="BL.xsd"/>
9           </xsd:schema>
10      </types>
11      <message name="checkIn">
12          <part name="part1" element="ns:MT-checkIn"/>
13      </message>
14      <message name="checkOut">
15          <part name="part1" element="ns:MT-checkOut"/>
16      </message>
17      <portType name="BlackListPortType">
```

```
18          <operation name="check">
19              <input name="input1" message="tns:checkIn"/>
20              <output name="output1" message="tns:checkOut"/>
21          </operation>
22      </portType>
23      <binding name="BlackListBinding" type="tns:BlackListPortType">
24          <soap:binding style="document" transport="http://schemas.xmlsoap.org
                /soap/http"/>
25          <operation name="check">
26              <soap:operation/>
27              <input name="input1">
28                  <soap:body use="literal"/>
29              </input>
30              <output name="output1">
31                  <soap:body use="literal"/>
32              </output>
33          </operation>
34      </binding>
35      <service name="BlackListService">
36          <port name="BlackListPort" binding="tns:BlackListBinding">
37              <soap:address location="http://localhost:${HttpDefaultPort}/
                    BlackListService/BlackListPort"/>
38          </port>
39      </service>
40      <plnk:partnerLinkType name="BlackList">
41          <!-- A partner link type is automatically generated when a new port
                type is added. Partner link types are used by BPEL processes.
42  In a BPEL process, a partner link represents the interaction between the
        BPEL process and a partner service. Each partner link is associated with
        a partner link type.
43  A partner link type characterizes the conversational relationship between
        two services. The partner link type can have one or two roles.-->
44          <plnk:role name="BlackListPortTypeRole" portType="
                tns:BlackListPortType"/>
45      </plnk:partnerLinkType>
46  </definitions>
```

D.3 WSDL of the Bank Service

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <definitions name="BankService" targetNamespace="http://j2ee.netbeans.org/
        wsdl/BankService"
3       xmlns="http://schemas.xmlsoap.org/wsdl/"
4       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5       xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://j2ee.
            netbeans.org/wsdl/BankService" xmlns:ns="http://xml.netbeans.org/
            schema/BK" xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype
            " xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:ns0="http:
            //j2ee.netbeans.org/wsdl/LoanService" xmlns:ns6="http://j2ee.netbeans
            .org/wsdl/LoanService">
6       <types>
7           <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/
                BankService">
8               <xsd:import namespace="http://xml.netbeans.org/schema/BK"
                    schemaLocation="BK.xsd"/>
9           </xsd:schema>
```

```
10      </types>
11      <message name="approveIn">
12          <part name="part1" element="ns:MT-approveIn"/>
13      </message>
14      <message name="approveOut">
15          <part name="part1" element="ns:MT-approveOut"/>
16      </message>
17      <message name="offerIn">
18          <part name="part1" element="ns:MT-offerIn"/>
19      </message>
20      <message name="offerOut">
21          <part name="part1" element="ns:MT-offerOut"/>
22      </message>
23      <message name="confirmIn">
24          <part name="part1" element="ns:MT-confirmIn"/>
25      </message>
26      <message name="cancelIn">
27          <part name="part1" element="ns:MT-cancelIn"/>
28      </message>
29      <portType name="BankServicePortType">
30          <operation name="approve">
31              <input name="input1" message="tns:approveIn"/>
32              <output name="output1" message="tns:approveOut"/>
33          </operation>
34          <operation name="offer">
35              <input name="input2" message="tns:offerIn"/>
36              <output name="output2" message="tns:offerOut"/>
37          </operation>
38          <operation name="confirm">
39              <input name="input3" message="tns:confirmIn"/>
40          </operation>
41          <operation name="cancel">
42              <input name="input4" message="tns:cancelIn"/>
43          </operation>
44      </portType>
45      <binding name="BankServiceBinding" type="tns:BankServicePortType">
46          <soap:binding style="document" transport="http://schemas.xmlsoap.org
                /soap/http"/>
47         <!-- <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
               style="document"/>-->
48          <operation name="approve">
49              <soap:operation/>
50              <input name="input1">
51                  <soap:body use="literal"/>
52              </input>
53              <output name="output1">
54                  <soap:body use="literal"/>
55              </output>
56          </operation>
57          <operation name="offer">
58              <soap:operation/>
59              <input name="input2">
60                  <soap:body/>
61              </input>
62              <output name="output2">
63                  <soap:body/>
64              </output>
65          </operation>
66          <operation name="confirm">
```

```
67              <input name="input3">
68                  <soap:body/>
69              </input>
70          </operation>
71          <operation name="cancel">
72              <input name="input4">
73                  <soap:body/>
74              </input>
75          </operation>
76      </binding>
77      <service name="BankServiceService">
78          <port name="BankServicePort" binding="tns:BankServiceBinding">
79              <soap:address location="http://localhost:${HttpDefaultPort}/
                    BankServiceService/BankServicePort"/>
80          </port>
81      </service>
82      <plnk:partnerLinkType name="BankService">
83          <!-- A partner link type is automatically generated when a new port
                type is added. Partner link types are used by BPEL processes.
84  In a BPEL process, a partner link represents the interaction between the
        BPEL process and a partner service. Each partner link is associated with
        a partner link type.
85  A partner link type characterizes the conversational relationship between
        two services. The partner link type can have one or two roles.-->
86          <plnk:role name="BankServicePortTypeRole" portType="
                tns:BankServicePortType"/>
87      </plnk:partnerLinkType>
88  <!--
89  <vprop:property name="LS_PROP" type="xsd:long"/>
90
91     <vprop:propertyAlias propertyName="tns:LS_PROP" element="ns:MT-offerOut">
92          <vprop:query>ns:fileNumber</vprop:query>
93      </vprop:propertyAlias>
94
95      <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
            tns:offerOut" part="part1">
96          <vprop:query>ns:fileNumber</vprop:query>
97      </vprop:propertyAlias>
98      <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
            tns:confirmIn" part="part1">
99          <vprop:query>ns:fileNumber</vprop:query>
100     </vprop:propertyAlias>
101     <vprop:propertyAlias propertyName="tns:LS_PROP" messageType="
            tns:cancelIn" part="part1">
102     <vprop:query>ns:fileNumber</vprop:query>
103     </vprop:propertyAlias>
104 -->
105 </definitions>
```

Bibliography

[1] Oracle application testing suite. URL http://www.oracle.com/technetwork/oem/app-test/index.html. 44

[2] Soap version 1.2. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, . URL http://www.w3.org/TR/2007/REC-soap12-part0-20070427/. 18

[3] Soapsonar, . URL http://www.crosschecknet.com/products/soapsonar.php. 44

[4] Parasoft soatest, . URL http://www.parasoft.com/jsp/fr/products/soatest.jsp. 44

[5] Testing and test control notation version 3 (ttcn-3). URL http://www.ttcn-3.org/. 45

[6] Universal description discovery and integration (uddi). URL http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm. 20

[7] Web services description language (wsdl). http://www.w3.org/TR/wsdl. URL http://www.w3.org/TR/wsdl. 18

[8] Business process model and notation (bpmn), . URL http://www.omg.org/spec/BPMN/2.0/. 38

[9] bpmn2bpel, . URL http://code.google.com/p/bpmn2bpel/. 38

[10] Construction and analysis of distributed processes (cadp). URL http://www.inrialpes.fr/vasy/cadp/. 41

[11] Service component architecture. URL http://osoa.org/display/Main/Service+Component+Architecture+Home. 38

[12] Software engineering for service-oriented overlay computers (sensoria). URL http://www.sensoria-ist.eu/. 38

[13] soapui tool. URL http://www.soapui.org/. 45

[14] Sensoria reference modeling language. URL http://www.cs.le.ac.uk/srml/. 16, 38

[15] Web services choreography description language version 1.0. URL http://www.w3.org/TR/ws-cdl-10/. 16, 21

[16] *JOpera: an agile environment for Web service composition with visual unit testing and refactoring*, 2005. doi: 10.1109/VLHCC.2005.48. URL http://dx.doi.org/10.1109/VLHCC.2005.48. 50

[17] Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, 2004. 30

[18] Joao Abreu, Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In *Proceedings of International Conference on Formal Methods for Networked and Distributed Systems Special focus on Service Oriented Computing*, volume 4574 of *Lecture Notes in Computer Science*, pages 358–373. Springer Verlag, 2007. doi: 10.1007/978-3-540-73196-2_23. URL http://www.pst.informatik.uni-muenchen.de:8080/Sensoria/DOWNLOAD/SRML_FORTE.pdf. 38

[19] Christian Attiogbé, Pascal Poizat, and Gwen Salaün. A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes. *IEEE Transactions on Software Engineering*, 33(3):157–170, 2007. 43

[20] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. Wsdl-based automatic test case generation for web services testing. In *Proceedings of the IEEE International Workshop*, pages 215–220, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2438-9. 45, 50

[21] Xiaoying Bai, Yongbo Wang, Guilan Dai, Wei-Tek Tsai, and Yinong Chen. A framework for contract-based collaborative verification and validation of web services. In *Proceedings of the 10th international conference on Component-based software engineering*, CBSE'07, pages 258–273, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73550-2. URL http://portal.acm.org/citation.cfm?id=1770657.1770679. 53

[22] Xiaoying Bai, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. Ontology-based test modeling and partition testing of web services. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 465–472, sept. 2008. doi: 10.1109/ICWS.2008.111. 54

[23] C. Bartolini, A. Bertolino, Francesca Lonetti, Eda Marchetti, and Andrea Polini. Taxi public page. URL http://labsewiki.isti.cnr.it/labse/tools/taxi/public/main. 45

[24] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Towards Automated WSDL-Based Testing of Web Services. In *Proc. of ICSOC*, volume 5364 of *LNCS*, 2008. 45, 50

[25] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Ioannis Parissis. Data flow-based validation of web services compositions: Perspectives and examples. In Rogério de Lemos, Felicita Di Giandomenico, Cristina Gacek, Henry Muccini, and Marlon Vieira, editors, *Architecting Dependable Systems V*, volume 5135 of *Lecture Notes in Computer Science*, pages 298–325. Springer Berlin / Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-85571-2_13. 54

[26] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Whitening soa testing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 161–170, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: http://doi.acm.org/10.1145/1595696.1595721. URL http://doi.acm.org/10.1145/1595696.1595721. 53

[27] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. *Software Testing, Verification, and Validation, 2008 International Conference on*, pages 326–335, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/ICST.2009.28. 50

[28] Gilles Bernot. Testing against formal specifications: a theoretical view. In *Proceedings of the international joint conference on theory and practice of software development on Advances in distributed computing (ADC) and colloquium on combining paradigms for software development (CCPSD): Vol. 2*, pages 99–119, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-53981-6. URL http://portal.acm.org/citation.cfm?id=112287.112303. 34

[29] Antonia Bertolino and Andrea Polini. The audition framework for testingweb services interoperability. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '05, pages 134–142, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2431-1. 54

[30] Antonia Bertolino, Lars Frantzen, and Andrea Polini. Audition of web services for testing conformance to open specified protocols. In *Architecting Systems*

with Trustworthy Components, number 3938 in LNCS. Springer-Verlag, 2006. 45, 54

[31] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Taxi–a tool for xml-based testing. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 53–54, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2892-9. 45

[32] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Automatic test data generation for xml schema-based partition testing. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2971-2. 45

[33] Aysu Betin-Can and Tevfik Bultan. Verifiable web services with hierarchical interfaces. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2409-5. doi: http://dx.doi.org/10.1109/ICWS.2005.128. URL http://dx.doi.org/10.1109/ICWS.2005.128. 51

[34] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003. 85

[35] Raquel Blanco, José García-Fanjul, and Javier Tuya. A first approach to test case generation for bpel compositions of web services using scatter search. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 131–140, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3671-2. doi: 10.1109/ICSTW.2009.24. URL http://portal.acm.org/citation.cfm?id=1547559.1548245. 53

[36] Ed Brinksma. Specification modules in lotos. In *FORTE*, pages 101–115, 1989. 41

[37] Michael J. Butler, Carla Ferreira, and Muan Yong Ng. Precise modelling of compensating business transactions and its application to bpel. *J. UCS*, 11(5): 712–743, 2005. 42

[38] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In *Proc. of ASE*, 2007. 86, 118

[39] Tien-Dung Cao, Patrick Felix, Richard Castanet, and Ismail Berrada. Testing web services composition using the tgse tool. In *Proceedings of the 2009 Congress on Services - I*, pages 187–194, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3708-5. doi: 10.1109/SERVICES-I.2009.42. URL http://portal.acm.org/citation.cfm?id=1590963.1591549. 48, 54

[40] W. K. Chan, S. C. Cheung, and Karl. R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 470–476, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2472-9. doi: http://dx.doi.org/10.1109/QSIC.2005.67. URL http://dx.doi.org/10.1109/QSIC.2005.67. 50

[41] Kevin M. Conroy, Mark Grechanik, Matthew Hellige, Edy S. Liongosari, and Qing Xie. Automatic test generation from gui-based applications for testing web services. In *In ICSM*, 2007. 52

[42] Guilan Dai, Xiaoying Bai, Yongbo Wang, and Fengjun Dai. Contract-based testing for web services. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 517–526, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2870-8. doi: http://dx.doi.org/10.1109/COMPSAC.2007.100. URL http://dx.doi.org/10.1109/COMPSAC.2007.100. 54

[43] Guilan Dai, Xiaoying Bai, and Chongchong Zhao. A framework for model checking web service compositions based on bpel4ws. In *Proceedings of the IEEE International Conference on e-Business Engineering*, ICEBE '07, pages 165–172, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3003-6. doi: http://dx.doi.org/10.1109/ICEBE.2007.11. URL http://dx.doi.org/10.1109/ICEBE.2007.11. 51

[44] Lourival F. Junior de Almeida and Silvia R. Vergilio. Exploring perturbation based testing for web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 717–726, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2669-1. doi: 10.1109/ICWS.2006.60. URL http://portal.acm.org/citation.cfm?id=1172963.1173112. 51

[45] Wen-Li Dong and Hang YU. Web service testing method based on fault-coverage. In *Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops*, EDOCW '06, pages 43–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2743-4. 53

[46] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing bpel-based web service composition using high-level petri nets. In *Proceedings of the 10th IEEE*

International Enterprise Distributed Object Computing Conference, pages 441–444, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2558-X. 46, 52

[47] André Takeshi Endo, Adenilso da Silva Simão, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. Web services composition testing: A strategy based on structural testing of parallel programs. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3383-4. doi: 10.1109/TAIC-PART.2008.9. URL http://portal.acm.org/citation.cfm?id=1475701.1476366. 53

[48] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, April 2004. 16

[49] J.P. Escobedo, C. Gaston, P. Le Gall, and A. Cavalli. Testing web service orchestrators in context: A symbolic approach. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 257–267, sept. 2010. doi: 10.1109/SEFM.2010.30. 48

[50] José Pablo Escobedo Del Cid. *Symbolic test case generation for testing orchestrators in context*. These, Institut National des Télécommunications; Université Evry Val d'Essonne, November 2010. 47, 48, 54

[51] Andrea Ferrara. Web services: a process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 242–251, 2004. ISBN 1-58113-871-7. URL http://doi.acm.org/10.1145/1035167.1035202. 41

[52] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. *Proceedings of Web Services and Formal Methods (WSFM'07)*, 4184:193–213, 2006. doi: http://dx.doi.org/10.1007/11841197_13. URL http://rap.dsi.unifi.it/sensoriasite/41840193.pdf. 38

[53] Howard Foster, Sebastián Urrutia, Jeff Magee, and Jeff Kramer. Web service compositions: From xml syntax to service models. In *IDEAlliance XML Conference*, 2005. URL http://publicaciones.dc.uba.ar/Publications/2005/FUMK05. 41, 52

[54] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In *ICSE '06: Proceedings of the 28th international conference on Software*

engineering, pages 771–774, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. 41, 52

[55] L. Frantzen, J. Tretmans, and R. d. Vries. Towards Model-Based Testing of Web Services. In A. Polini, editor, *International Workshop on Web Services - Modeling and Testing – WS-MaTe 2006*, pages 67–82, Palermo, Italy, 2006. URL http://www.cs.ru.nl/~lf/publications/FTdV06.pdf. 46

[56] L. Frantzen, M.N. Huerta, Z.G. Kiss, and T. Wallet. On-The-Fly Model-Based Testing of Web Services with Jambition. In *Proc. of WS-FM*, volume 5387 of *LNCS*, 2009. 46, 54

[57] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A Symbolic Framework for Model-Based Testing. In *Proc. of FATES/RV*, volume 4262 of *LNCS*, 2006. 82

[58] Chen Fu, Barbara Ryder, Ana Milanova, and David Wonnacott. Testing of java web services for robustness. In *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA*, pages 23–34. ACM Press, 2004. 53

[59] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. pages 621–630. ACM Press, 2004. 42, 51

[60] Xiang Fu, Tevfik Bultan, and Jianwen Su. Wsat: A tool for formal analysis of web services. In *the Proc. of 16th Int. Conf. on Computer Aided Verification (CAV*, pages 510–514. Springer, 2004. 42, 51

[61] Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *CAV*, pages 158–163, 2007. 41

[62] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In Parosh Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2011*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-19834-2. 109

[63] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *Proc. of WS-MaTe*, 2006. 42, 51

[64] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In *Proc. of CAV '01*, 2001. 74

[65] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic Execution Techniques for Test Purpose Definition. In *Proc. of TESTCOM*, volume 3964 of *LNCS*, 2006. 82, 85

[66] M.-C. Gaudel. Testing can be formal, too. In *TAPSOFT'95, International Joint Conference, Theory And Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96, Aarhus, Denmark, 1995. Springer Verlag. 34

[67] Marie-Claude Gaudel. Formal methods and testing: Hypotheses, and correctness approximations. In John Fitzgerald, Ian Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 597–597. Springer Berlin / Heidelberg, 2005. URL http://dx.doi.org/10.1007/11526841_2. 34

[68] Zhang Guangquan, Rong Mei, and Zhang Jun. A business process of web services testing method based on uml2.0 activity diagram. In *Intelligent Information Technology Application, Workshop on*, pages 59–65, dec. 2007. doi: 10.1109/IITA.2007.83. 53

[69] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference - Volume 17*, ADC '03, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. ISBN 0-909-92595-X. URL http://portal.acm.org/citation.cfm?id=820085.820121. 40

[70] Samer Hanna and Malcolm Munro. Fault-based web services testing. In *Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 471–476, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3099-4. 50

[71] Reiko Heckel and Marc Lohmann. Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, 82:2003, 2004. 50

[72] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *Proceedings of the International Conference on Business Process Management (BPM2005), volume 3649 of Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2005. 40

[73] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5. 41

[74] Shan-Shan Hou, Lu Zhang, Qian Lan, Hong Mei, and Jia-Su Sun. Generating effective test sequences for bpel testing. In *Proceedings of the 2009*

Ninth International Conference on Quality Software, QSIC '09, pages 331–340, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3828-0. doi: http://dx.doi.org/10.1109/QSIC.2009.50. URL http://dx.doi.org/10.1109/QSIC.2009.50. 53

[75] http. Hypertext transfer protocol (http). http://www.w3.org/TR/2009/WD-HTTP-in-RDF10-20091029/. URL http://www.w3.org/TR/2009/WD-HTTP-in-RDF10-20091029/. http://www.w3.org/Protocols/. 18

[76] Hai Huang, Wei-Tek Tsai, Raymond Paul, and Yinong Chen. Automated model checking and testing for composite web services. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, pages 300–307, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2356-0. doi: http://dx.doi.org/10.1109/ISORC.2005.16. URL http://dx.doi.org/10.1109/ISORC.2005.16. 51

[77] T. Jéron, V. Rusu, B. Jeannet, and E. Zinovieva. Symbolic Test Selection based on Approximate Analysis. In *Proc. of TACAS*, volume 3440 of *LNCS*, 2005. 74

[78] Hui Kang, Xiuli Yang, and Sinmiao Yuan. Modeling and verification of web services composition based on cpn. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, NPC '07, pages 613–617, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2943-7. URL http://portal.acm.org/citation.cfm?id=1306873.1306989. 40, 52

[79] Kathrin Kaschner and Niels Lohmann. Automatic Test Case Generation for Interacting Services. In *Proc. of ICSOC 2008 Workshops*, volume 5472 of *Lecture Notes in Computer Science*, 2009. 46, 54

[80] Raman Kazhamiakin, Paritosh Pandya, and Marco Pistore. Representation, verification, and computation of timed properties in web services composition. pages 497–504, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/ICWS.2006.112. 41, 52

[81] Raman Kazhamiakin, Paritosh K. Pandya, and Marco Pistore. Timed modelling and analysis in web service compositions. In *ARES*, pages 840–846, 2006. 41, 52

[82] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. of TACAS*, volume 2619 of *LNCS*, 2003. 82

[83] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints, 2009. 113

[84] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976. 56, 82

[85] David Kitchin, Evan Powell, and Jayadev Misra. Simulation using orchestration (extended abstract). In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 2–15. Springer, 2008. ISBN 978-3-540-79979-5. doi: 10.1007/978-3-540-79980-1_2. 16

[86] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a language for extended goals. In *Eighteenth national conference on Artificial intelligence*, pages 447–454, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. ISBN 0-262-51129-0. URL http://portal.acm.org/citation.cfm?id=777092.777163. 42

[87] Mounir Lallali. *Modélisation et Test Fonctionnel de l'Orchestration de Service Web*. PhD thesis, Université d'Evry-Val d'Essone. Institut national des télécommunications, 2009. 8, 42, 54, 66

[88] Mounir Lallali, Fatiha Zaïdi, Ana Cavalli, and Iwon Hwang. Automatic Timed Test Case Generation for Web Services Composition. In *Proc. of ECOWS*, 2008. 48, 54

[89] J.C. LAPRIE. *Guide de la sûreté de fonctionnement*. Cepadues, 1996. 30

[90] Shufang Lee, Xiaoying Bai, and Yinong Chen. Automatic mutation testing and simulation on owl-s specified web services. In *Proceedings of the 41st Annual Simulation Symposium (anss-41 2008)*, pages 149–156, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3143-4. 51

[91] C. Lenz, J. Chimiak-Opoka, and R Breu. Model driven testing of soa–based software. In *Proceedings of the SEMSOA Workshop 2007 on Software Engineering Methods for Service-Oriented Architecture, Hannover*, 2007. 53

[92] Li Li, Wu Chou, and Weiping Guo. Control flow analysis and coverage driven testing for web services. In *Proceedings of the 2008 IEEE International Conference on Web Services*, pages 473–480, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3310-0. doi: 10.1109/ICWS.2008.104. URL http://portal.acm.org/citation.cfm?id=1474549.1474764. 52

[93] Yingmin Li. *"Modeling BPEL Web Services for Diagnosis: towards self-healing Web Services"*. PhD thesis, UNIVERSITE PARIS SUD 11, December 2010. 54

[94] Yingmin Li, Tarek Melliti, and Philippe Dague. A colored petri nets model for diagnosing data faults of bpel services. *The 20th International Workshop on Principles of Diagnosis (DX'09)*, 2009. 40, 54

[95] Zhong Jie Li, Jun Zhu, Liang-Jie Zhang, and Naomi M. Mitsumori. Towards a practical and effective method for web services test case generation. In *AST*, pages 106–114, 2009. 50

[96] Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhang. Bpel4ws unit testing: Framework and implementation. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 103–110, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2409-5. 45, 52

[97] Feng Lin, Michael Ruth, and Shengru Tu. Applying safe regression test selection techniques to java web services. In *Proceedings of the International Conference on Next Generation Web Services Practices*, pages 133–142, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2664-0. 53

[98] Chien-Hung Liu, Shu-Ling Chen, and Xue-Yuan Li. A ws-bpel based structural testing approach for web service compositions. In *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 135–141, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3499-2. 46

[99] Niels Lohmann. *Correctness of services and their composition*. PhD thesis, Universität Rostock / Technische Universiteit Eindhoven, 2010. 21

[100] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting bpel processes. In *Proceedings of the 4th International Conference on Business Process Management (BPM2006), volume 4102 of Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2006. 52

[101] Niels Lohmann, H.M.W. Verbeek, and Remco Dijkman. Petri net transformations for business processes - a survey. *LNCS ToPNoC*, II(5460):46–63, March 2009. Special Issue on Concurrency in Process-Aware Information Systems. 40

[102] Nik Looker, Binka Gwynne, Jie Xu, and Malcolm Munro. Determining the dependability of service-oriented architectures, 2007. 51

[103] Formal Systems (Europe) Ltd. *FDR (Failures-Divergence Refinement) is a model-checking tool*. URL http://www.fsel.com/. 41

[104] Chunyan Ma, Chenglie Du, Tao Zhang, Fei Hu, and Xiaobin Cai. Wsdl-based automated test data generation for web service. *Computer Science*

and Software Engineering, International Conference on, 2:731–737, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/CSSE.2008.790. 50

[105] Chunyan Ma, Junsheng Wu, Tao Zhang, Yunpeng Zhang, and Xiaobin Cai. Testing bpel with stream x-machine. In *Proceedings of the 2008 International Symposium on Information Science and Engieering - Volume 01*, pages 578–582, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3494-7. doi: 10.1109/ISISE.2008.201. URL http://portal.acm.org/citation.cfm?id=1493610.1493664. 53

[106] Bendick Mahleko and Andreas Wombacher. Indexing business processes based on annotated finite state automata. In *Proceedings of the IEEE International Conference on Web Services*, pages 303–311, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2669-1. doi: 10.1109/ICWS.2006.74. URL http://portal.acm.org/citation.cfm?id=1172963.1173065. 41

[107] Evan Martin, Suranjana Basu, and Tao Xie. Automated testing and response analysis ofweb services. *Web Services, IEEE International Conference on*, 0: 647–654, 2007. doi: http://doi.ieeecomputersociety.org/10.1109/ICWS.2007.49. 51

[108] Radu Mateescu and Sylvain Rampacek. Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In *Advances in Enterprise Engineering I*, volume 10 of *Lecture Notes in Business Information Processing*, pages 179–193. Springer, 2008. 41, 52, 63, 67

[109] Philip Mayer and Daniel Lübke. Bpelunit - the open source unit testing framework for bpel. URL http://www.bpelunit.net/. 45, 50

[110] Philip Mayer and Daniel Lübke. Towards a bpel unit testing framework. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, TAV-WEB '06, pages 33–42, New York, NY, USA, 2006. ACM. ISBN 1-59593-458-8. 45, 50

[111] Philip Mayer, Andreas Schroeder, and Nora Koch. A model-driven approach to service orchestration. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, pages 533–536, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3283-7-02. doi: 10.1109/SCC.2008.91. URL http://portal.acm.org/citation.cfm?id=1443230.1444290. 37

[112] Hong Mei and Lu Zhang. A framework for testing web services and its supporting tool. In *Proceedings of the IEEE International Workshop*, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2438-9. doi: 10.1109/SOSE.2005.1. URL http://portal.acm.org/citation.cfm?id=1105001.1105566. 50

[113] Lijun Mei, W.K. Chan, and T.H. Tse. Data flow testing of service-oriented workflow applications. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 371–380, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. 53

[114] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353. 41

[115] Gerardo Morales, Stéphane Maag, Ana R. Cavalli, Wissam Mallouli, Edgardo Montes de Oca, and Bachar Wehbi. Timed extended invariants for the passive testing of web services. In *ICWS*, pages 592–599, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4128-0. 48, 54

[116] Mark Harman Mustafa Bozkurt and Youssef Hassoun. Testing web services: A survey. Technical Report TR-10-01, Department of Computer Science, King's College London, January 2010. 43

[117] Oasis. *Web Services Business Process Execution Language (WSBPEL) Version 2.0*, April 2007. URL http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html. 16, 21, 25, 27, 28, 57, 59, 65, 71, 115

[118] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29:1–10, September 2004. ISSN 0163-5948. 50

[119] C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center Report, 2006. 38

[120] Chun Ouyang, Eric Verbeek, Wil M.P. van der Aalst, Stephan W. Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Wofbpel: A tool for automated analysis of bpel processes. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Third International Conference on Service Oriented Computing (ICSOC 2005)*, pages 484–489, Amsterdam, The Netherlands, 2005. Springer. URL http://eprints.qut.edu.au/2918/. 52

[121] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67:162–198, July 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.03.002. URL http://portal.acm.org/citation.cfm?id=1274201.1274469. 40

[122] Amit M. Paradkar, Avik Sinha, Clay Williams, Robert D. Johnson, Susan Outterson, Charles Shriver, and Carol Liang. Automated functional conformance

test generation for semantic web services. *Web Services, IEEE International Conference on*, 0:110–117, 2007. doi: http://doi.ieeecomputersociety.org/10.1109/ICWS.2007.48. 53

[123] Y. Pencole, Y. Cordier, M. -o. Grastien, Council Canada, M. o, Yuhong Yan, Marie odile Cordier, Yannick Pencolé, and Alban Grastien. Monitoring web service networks in a model-based approach. In *In 3rd IEEE European Conference on Web Services (ECOWS05*, pages 14–16. IEEE Computer Society. 42, 54

[124] Liam Peyton, Bernard Stepien, and Pierre Seguin. Integration testing of composite applications. In *Hawaii International Conference on System Sciences*, 2008. doi: 10.1109/HICSS.2008.212. 50

[125] Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test synthesis from uml models of distributed software. *IEEE Trans. Software Eng.*, 33(4):252–269, 2007. 74

[126] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 293–301, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2409-5. URL http://dx.doi.org/10.1109/ICWS.2005.27. 41

[127] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. *Current Trends in Concurrency, volume 224 of Lecture Notes in Computer Science, pages 510-584. Springer-Verlag*, 1986. 34

[128] Pascal Poizat and Jean-Claude Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. *Journal of Universal Computer Science*, 12(12):1741–1782, 2006. 56, 64

[129] Michael P.Papazoglou and Jean jacques Dubray. A survey of web service technologies. Technical Report 586, University of Trento, July 2004. URL http://eprints.biblio.unitn.it/archive/00000586/. 18, 20

[130] Pemadeep Ramsokul and Arcot Sowmya. Aseha: A framework for modelling and verification ofweb services protocols. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 196–205, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2678-0. doi: 10.1109/SEFM.2006.8. URL http://portal.acm.org/citation.cfm?id=1158333.1158365. 51

[131] Pemadeep Ramsokul and Arcot Sowmya. A sniffer based approach to ws protocols conformance checking. In *Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 58–65, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2638-1. 51

[132] Valentín Valero Ruiz, M.E. Cambronero, G. Díaz, and J.J. Pardo. Transforming web services choreographies with priorities and time constraints into prioritized-time petri nets. *FLACOS 2007*, 2007. 40

[133] Michael Ruth and Shengru Tu. A safe regression test selection technique for web services. In *Proceedings of the Second International Conference on Internet and Web Applications and Services*, pages 47–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2844-9. doi: 10.1109/ICIW.2007.8. URL http://portal.acm.org/citation.cfm?id=1260202.1260609. 53

[134] Gwen Salaün, Andrea Ferrara, and Antonella Chirichiello. Negotiation among web services using lotos/cadp. In *ECOWS*, pages 198–212, 2004. 41, 52

[135] S. Salva and A. Rollet. Automatic web service testing from wsdl descriptions. In *8th International Conference on Innovative Internet Community Systems I2CS 2008*, volume 2011 of *Lecture Notes in Informatics (LNI)*, Schoelcher, Martinique, 06 2008. Gesellschaft für Informatik (GI). 45, 53

[136] Holger Schlingloff, Axel Martens, and Karsten Schmidt. Modeling and model checking web services. *Electronic Notes in Theoretical Computer Science*, 126:3–26, 2005. ISSN 1571-0661. doi: DOI:10.1016/j.entcs.2004.11.011. URL http://www.sciencedirect.com/science/article/B75H1-4FKXPY9-F/2/611b9d32436c15382a1c01b62ee85090. Proceedings of the 2nd International Workshop on Logic and Communication in Multi-Agent Systems (2004). 51

[137] Karsten Schmidt. Lola: a low level analyser. In *Proceedings of the 21st international conference on Application and theory of petri nets*, ICATPN'00, pages 465–474, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3-540-67693-7. URL http://portal.acm.org/citation.cfm?id=1754589.1754619. 40

[138] Reda Siblini and Nashat Mansour. Testing web services. In *AICCSA'05*, pages –1–1, 2005. 51

[139] Avik Sinha and Amit Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, TAV-WEB '06, pages 17–22, New York, NY, USA, 2006. ACM. ISBN 1-59593-458-8. 54

[140] Colin Smythe. Initial investigations into interoperability testing of web services from their specification using the unified modelling language. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 95–119, Palermo, Sicily, ITALY, June 9th 2006. 54

[141] Harry M. Sneed and Shihong Huang. Wsdltest - a tool for testing web services. In *WSE'06*, pages 14–21, 2006. 50

[142] Christian Stahl. A Petri Net Semantics for BPEL. Informatik-Berichte 188, Humboldt-Universität zu Berlin, July 2005. 40

[143] Abbas Tarhini, Hacène Fouchal, and Nashat Mansour. A simple approach for testing web service based applications. In *IICS*, pages 134–146, 2005. 54

[144] Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Formal Methods for Service Composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10, 2007. 41

[145] G.J. Tretmans. Test generation with inputs, outputs and repetitive quiescence, 1996. URL http://doc.utwente.nl/65463/. 46

[146] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, pages 46–65, 1999. 34

[147] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008. 34, 62

[148] Stefan Troschütz. *Web Service Test Framework with TTCN-3*. PhD thesis, The University of Göttingen - Germany, Germany, 2007. 45

[149] W. T. Tsai, Ray Paul, Weiwei Song, and Zhibin Cao. Coyote: An xml-based framework for web services testing. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering*, HASE '02, pages 173–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1769-2. URL http://portal.acm.org/citation.cfm?id=795685.797694. 50

[150] W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao. Verification of web services using an enhanced uddi server. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:131, 2003. ISSN 1530-1443. doi: http://doi.ieeecomputersociety.org/10.1109/WORDS.2003.1218075. 50

[151] Wei-Tek Tsai, Yinong Chen, Raymond Paul, Hai Huang, Xinyu Zhou, and Xiao Wei. Adaptive testing, oracle generation, and test case ranking for web services. In *Proceedings of the 29th Annual International Computer*

Software and Applications Conference - Volume 01, COMPSAC '05, pages 101–106, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2413-3. doi: http://dx.doi.org/10.1109/COMPSAC.2005.40. URL http://dx.doi.org/10.1109/COMPSAC.2005.40. 52

[152] Wei-Tek Tsai, Xiao Wei, Yinong Chen, Ray Paul, and Bingnan Xiao. Swiss cheese test case generation for web services testing. *IEICE - Trans. Inf. Syst.*, E88-D:2691–2698, December 2005. ISSN 0916-8532. 51

[153] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003. ISSN 0098-5589. 74

[154] Franck van Breugel and Maria Koshkina. Models and verification of bpel, 2006. URL www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf. 40

[155] Marco Vieira, Nuno Laranjeiro, and Henrique Madeira. Benchmarking the robustness of web services. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 322–329, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3054-0. doi: 10.1109/PRDC.2007.24. URL http://portal.acm.org/citation.cfm?id=1345534.1345824. 51

[156] W3C. XML Path Language (XPath) Version 1.0. Technical report, W3C, 1999. 65

[157] *XML Schema*. W3C recommendation. URL http://www.w3.org/XML/Schema. 18

[158] Rui Wang and Ning Huang. Requirement model-based mutation testing for web service. In *Next Generation Web Services Practices, 2008. NWESP '08. 4th International Conference on*, pages 71–76, oct. 2008. doi: 10.1109/NWeSP.2008.20. 51

[159] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient analysis of bpel 2.0 processes using p-calculus. In *APSCC*, pages 266–274, 2007. 41, 52

[160] Stephen A. White. *Using BPMN to Model a BPEL Process*. 38

[161] Andreas Wombacher, Peter Fankhauser, and Erich Neuhold. Transforming bpel into annotated deterministic finite state automata for service discovery. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, pages 316–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2167-3. doi: http://dx.doi.org/10.1109/ICWS.2004.117. URL http://dx.doi.org/10.1109/ICWS.2004.117. 41

[162] Chunxiang Xu, Hanpin Wang, and Wanling Qu. Modeling and verifying bpel using synchronized net. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 2358–2362, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-753-7. doi: http://doi.acm.org/10.1145/1363686.1364248. URL http://doi.acm.org/10.1145/1363686.1364248. 52

[163] Wuzhi Xu, Jeff Offutt, and Juan Luo. Testing web services by xml perturbation. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 257–266, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2482-6. doi: 10.1109/ISSRE.2005.44. URL http://portal.acm.org/citation.cfm?id=1104997.1105254. 50

[164] Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang. Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 75–84, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2684-5. doi: 10.1109/ISSRE.2006.16. URL http://portal.acm.org/citation.cfm?id=1190616.1191224. 52

[165] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying web services composition based on hierarchical colored petri nets. In *Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, IHIS '05, pages 47–54, New York, NY, USA, 2005. ACM. ISBN 1-59593-184-8. doi: http://doi.acm.org/10.1145/1096967.1096977. URL http://doi.acm.org/10.1145/1096967.1096977. 52

[166] Yanping Yang, QingPing Tan, Yong Xiao, Feng Liu, and Jinshan Yu. Transform bpel workflow into hierarchical cp-nets to make tool support for verification. In *APWeb*, pages 275–284, 2006. 40, 52

[167] Hsu-Chun Yen. Introduction to petri net theory. In *Recent Advances in Formal Languages and Applications*, pages 343–373. 2006. 39

[168] W. L. Yeung. Mapping ws-cdl and bpel into csp for behavioural specification and verification of web services. In *Proceedings of the European Conference on Web Services*, pages 297–305. IEEE Computer Society, 2006. ISBN 0-7695-2737-X. doi: 10.1109/ECOWS.2006.26. 41, 52

[169] Xiaochuan Yi and Krys J. Kochut. A cp-nets-based design and verification framework for web services composition. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, pages 756–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2167-3. doi: http://dx.doi.org/10.1109/ICWS.2004.2. URL http://dx.doi.org/10.1109/ICWS.2004.2. 52

[170] Yuan Yuan, Zhongjie Li, and Wei Sun. A graph-search based approach to bpel4ws test generation. In *Proceedings of the International Conference on Software Engineering Advances*, pages 14–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2703-5. doi: 10.1109/ICSEA.2006.6. URL http://portal.acm.org/citation.cfm?id=1193212.1193784. 42, 46, 53

[171] Bensheng Yun, Junwei Yan, and Min Liu. Behavior-based web services matchmaking. *Network and Parallel Computing Workshops, IFIP International Conference on*, 0:483–487, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/NPC.2008.56. 41

[172] Jian Zhang. Specification analysis and test data generation by solving boolean combinations of numeric constraints. *Asia-Pacific Conference on Quality Software*, 0:267, 2000. doi: http://doi.ieeecomputersociety.org/10.1109/APAQ.2000.883800. 46

[173] Yongyan Zheng and Paul J. Krause. Automata semantics and analysis of bpel. 2007. URL http://epubs.surrey.ac.uk/publcomp3/3. 41, 51

[174] Yongyan Zheng, Jiong Zhou, and Paul Krause. A model checking based test case generation framework for web services. In *ITNG*, pages 715–722, 2007. 41, 51

[175] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a uml profile for software product lines. In *Proc. of Software Product-Family Engineering*, pages 129–139, 2003. 74

[176] Eléna Zinovieva-Leroux. *Méthodes symboliques pour la génération de tests de systèmes réactifs comportant des données*. PhD thesis, Université de Rennes 1, 2004. 34, 36