An Automatic Test Framework for BPEL-based Web Services

Yongyan Zheng
Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

A thesis submitted for the degree of
Doctor of Philosophy
May 2007
Abstract

Recent years have seen a rapid growth in the development of web services technology. BPEL (Business Process Execution Language) as a de-facto standard for web service orchestration has drawn particularly attention from researchers and industries. BPEL is a semi-formal flow language with complex features, so it is essential to apply automated validation tools in finding the interaction inconsistencies of BPEL processes. In addition to validating the model properties by verification, it is desirable to test the correctness with respect to the functional requirements. To test a model thoroughly, we need to cover different execution scenarios. As is well known, it is tedious, time-consuming, and error prone to design test cases manually, especially for complex modelling languages. Hence, it is desirable to apply existing model-based-testing techniques in the domain of web services.

This thesis proposes a web service automaton as the operational semantics for BPEL, and presents an automatic test framework to verify and test BPEL processes. From the testing point of view, we show the suitability of using web service automaton formalism for BPEL by modelling various BPEL features. Based on the web service automata, we provide a model checking based test framework to verify the general properties and generate test cases for BPEL processes. The framework supports both control-flow and data-flow testing of BPEL. Two levels of test cases can be generated to check the behavioural and interface conformance for web services. To our knowledge, none of the prior research studies the verification and testing for BPEL control and data flows in a unified approach.

The formal work in this thesis underpins the development of an automated test case generation and execution tool that has been integrated into the DBE Studio that was developed under the EU funded Digital Business Ecosystems Integrated Programme.
I would like to thank my supervisor Professor Paul Krause, for his support and guidance throughout this PhD research. I really appreciate for his always kindness and thoughtful mentoring. I would also like to thank Dr. Mike Shields, for spending valuable time with me to formalise this work. Thanks must also go to DBE project for funding this research.

I should give a big thank to all my friends and colleagues for their friendship. I would especially thank Jiong and Fanfan, who are the angels in my research life. I would be very much thanking to my mother, sisters, and aunt for their support, understanding, and love. Finally and most importantly, I wish to dedicate this thesis to my father.
## Contents

### Nomenclature

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Literature Review</td>
<td>10</td>
</tr>
<tr>
<td>3 Web Service Automata</td>
<td>26</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Background .................................................. 2
1.2 Motivation .................................................... 6
1.3 Aims and Objectives ........................................... 7
1.4 Methodology .................................................. 7
1.5 Contributions .................................................. 8
1.6 Thesis Structure .............................................. 9

### 2 Literature Review

2.1 Formal Semantics and Verification of BPEL .................. 10
  2.1.1 Petri Nets based approaches .............................. 11
  2.1.2 Process Algebras based approaches ....................... 13
  2.1.3 Automata based approaches ............................... 15
2.2 Analysis of Data Dependencies in Orchestration .......... 17
2.3 Testing of BPEL based web services ......................... 20
2.4 Multiple Inputs Automata .................................... 21
2.5 Summary ...................................................... 23

### 3 Web Service Automata

3.1 Static Semantics .............................................. 26
3.2 Dynamic Semantics ........................................... 30
3.3 Composition .................................................. 34
3.4 Compatibility and Anti-patterns .............................. 40
  3.4.1 Compatibility ............................................. 41
CONTENTS

5.2 Framework Overview .............................................. 95
5.3 Model Transformation ............................................. 97
  5.3.1 Model Simplification in WSA ................................. 97
  5.3.2 From BPEL to WSA ........................................... 100
  5.3.3 From WSA to Promela ....................................... 100
  5.3.4 From WSA to SMV ........................................... 102
5.4 BPEL Verification .................................................. 104
  5.4.1 Pre-checking Illegal Cross-boundary Links .................. 105
  5.4.2 Model Checking BPEL General Properties .................... 105
5.5 BPEL Test Case Generation ....................................... 109
  5.5.1 Test Coverage Criteria in Trap Properties .................. 110
  5.5.2 Symbolic Test Case Generation ............................... 112
  5.5.3 From Counterexamples to Test Cases ......................... 113
5.6 WSDL Test Case Generation ....................................... 113
5.7 Summary ............................................................ 113

6 Case Studies and Tool Support ................................. 115
  6.1 Case Studies ..................................................... 115
    6.1.1 Loan Approval ............................................ 116
    6.1.2 Purchase Order ........................................... 123
    6.1.3 Shipping Service .......................................... 126
  6.2 Tool Support .................................................... 128
  6.3 Summary ........................................................ 130

7 Conclusions and Future Work ..................................... 131
  7.1 Conclusions ...................................................... 131
  7.2 Discussion ....................................................... 135
  7.3 Future Work ..................................................... 136
## List of Figures

1.1 Web Service Stack .................................................. 3
1.2 UML diagrams and web service specifications .................. 4

2.1 An example of Petri nets for BPEL activities [46] .......... 11

3.1 Message overtaking .............................................. 32
3.2 Unspecified reception .............................................. 43
3.3 An example of anti-pattern 1 ...................................... 44
3.4 Non-local branching 1 .............................................. 44
3.5 An example of anti-patterns 2 and 3 ............................ 45
3.6 Non-local branching 2 .............................................. 46

4.1 An example of machines with parent and child relationships .. 49
4.2 The common layout of compound machines .................... 50
4.3 Multiple-input events and common machine structure .......... 51
4.4 linkWrapper machine ................................................ 53
4.5 Unreachable and deadlock activities ............................. 55
4.6 Modelling internal data exchanges ............................... 60
4.7 Internal data exchange algorithm ................................. 61
4.8 External data exchange model ..................................... 63
4.9 The machine of the receive and reply activities ............... 65
4.10 The machine of the assign, throw, terminate, and empty activities .. 66
4.11 The machines of synchronous and asynchronous invoke activities .... 67
4.12 Invoke activity with and without inline handlers ............... 68
4.13 The machine of the sequence activity ........................... 69
4.14 The machine of the flow activity ................................. 70
LIST OF FIGURES

4.15 The machine of the while activity ........................................ 71
4.16 The machine of the switch activity ........................................ 72
4.17 The machine of the pick activity .......................................... 73
4.18 The machine of the default terminationHandler ....................... 76
4.19 The machine of the scope activity with faultHandlers .......... 78
4.20 The machine of the scope activity without faultHandlers .... 79
4.21 The machine of the process activity ...................................... 81
4.22 The machine of the faultHandlers activity ............................ 83
4.23 The hierarchy of compensationHandler invocation ................. 84
4.24 The machine of the compensationHandler activity ................ 86
4.25 The compensateManager machine ....................................... 87
4.26 The machine of the eventHandlers activity ........................... 89
4.27 The machine of the MEH for eventHandlers activity ............. 90

5.1 The model checking approach ............................................... 93
5.2 Framework architecture .................................................... 95
5.3 An example of Promela model ............................................ 102
5.4 An example of SMV model .................................................. 104
5.5 Simultaneously enabled transitions ....................................... 108
5.6 An example of state coverage ............................................. 111
5.7 An example of predicate handling ....................................... 112

6.1 The business scenarios of the loan approval example ............ 117
6.2 The BPEL activity hierarchy of approval, assessor, and approver 117
6.3 The machine relationships of approval, assessor, and approver services 118
6.4 Global data exchange model ............................................... 118
6.5 The machines for the approval BPEL model ........................... 120
6.6 The machines for the assessor or approver BPEL model ......... 121
6.7 A single BPEL process as the SUT ................................... 121
6.8 Multiple BPEL processes as the SUT ................................ 122
6.9 The business scenarios of the ordering service .................... 124
6.10 The BPEL activity hierarchy and machine relationships of ordering service 125
6.11 The business scenarios of the shipping service ................. 126
6.12 The BPEL activity hierarchy and machine relationships of shipping service 127
LIST OF FIGURES

6.13 Case Study Summary ........................................ 128
6.14 A snapshot of the BPEL test wizard ......................... 129
Lists of Publications

Chapter 1

Introduction

In recent years, Service-Oriented Computing (SOC) has been actively researched. SOC provides a systematic and extensible framework for application-to-application interaction, built on top of existing web protocols and based on open XML standards. It defines a standardised mechanism to describe, locate, and communicate with online applications. In a SOC environment, each application becomes an accessible web service that is described using open standards. Due to the advantages of open standards, the SOC paradigm provides a flexible, re-usable, and loosely coupled model for distributed computing. SOC offers three main functions: communication protocols, service descriptions, and service discovery. In this work we look at the service descriptions, with a focus on the verification and testing of the behavioural aspect of web services. BPEL is the de-facto industry standard language to model the behaviour of web service compositions. BPEL is a semi-formal flow-based language with complex features, which may thus include fault behaviours. It becomes essential to verify a web service design before publishing it, and to test whether the published service conforms to the design model. The manual writing and verification of test cases for complex models is tedious, time-consuming and error prone. Hence, it is vital to automate this process. This thesis provides the theoretical background to a test-case generation and execution plug-in that has been developed for the DBESudio. The plug-in may be downloaded from the Source-Forge DBE Project.
1.1 Background

SOC is the emerging paradigm for the realisation of heterogeneous, distributed systems, obtained from the dynamic combination of remote applications owned and operated by distinct organisations. SOC characterises a collection of autonomous and self-contained web services. The essence of SOC lies in independent web services which communicate with each other exclusively through messages. No knowledge of the partner service is shared other than the message formats and the sequences of the messages that are expected. The agreed-on standards of service description, discovery, and communication explicitly allow that the partner service may be implemented with heterogeneous technology, with diverse applications written in different programming languages and running on different operating systems and hardware.

In the SOC architecture, there are three roles: service provider, service consumer, and service registry. Service providers can publish their services in a service registry. Service requesters or consumers can use the services that are retrieved via the service registry. A service registry provides facilities for service providers and consumers to find each other. A web service can play any or all of these roles.

A. Web Service Stack

To support the SOC architecture, web services must provide standards-based definitions of interoperability communication protocols, mechanisms for service description, discovery, and composition as well as quality of service (QoS) protocols. The web service stack is shown in Figure 1.1 below.

The initial specifications of web services consisting of SOAP, WSDL and UDDI, provides open XML-based mechanisms for service communication, service description, and service discovery.

- UDDI (Universal Description, Discovery and Integration)[3] provides a standard way for publishing and discovering information about web service.

- WSDL (Web service Definition Language) [17] defines a public interface of a web
1.1 Background

Figure 1.1: Web Service Stack

service, by describing the functions that can be provided by a web service. WSDL enables dynamic discovery and binding of compatible services which are used in conjunction with registry services.

- SOAP (Simple Object Access Protocol) [4] is a platform and language independent communication protocol that defines an XML-based format for web services to exchange information over HTTP by using remote procedure calls.

Web services describe their functionality using WSDL and they interact with each other by exchanging SOAP messages serialised in XML and sent over a transport protocol, usually HTTP. Moreover, UDDI is used to interconnect service providers and service consumers. This basic SOC model covers discovery, description, messaging, and transport layers of the web service stack in Figure 1.1.

To move beyond this basic model, mechanisms for service composition and QoS protocols are required. Several specifications have been proposed in these areas:


1.1 Background

- Quality of service layer: WS-Security ensures end-to-end message integrity, confidentiality, and authentication. WS-Reliable Messaging allows messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures. WS-Transaction provides a means to compose transactional qualities of service.

- Description layer: XML Schema describes the data formats used for constructing the messages addressed to and received from web services. WS-Policy extends WSDL to all the encoding and attachment of QoS information to services, in the form of reusable service policies.

In order to give an intuitive view of the main web service standard languages, we use UML diagrams to link SBVR, WS-CDL, BPEL, and WSDL. In Figure 1.2, SBVR may be seen to play a similar role to UML use case diagrams in capturing the business requirement; WS-CDL is similar to UML collaboration or sequence diagrams in modelling the global web service interactions; BPEL has a similar role to UML statecharts in modelling local web service interactions and the stateful behaviour of individual web services; finally, WSDL has similarities with UML class diagrams in describing web service interfaces.

![UML Diagrams and Web Services](image)

**Figure 1.2: UML diagrams and web service specifications**

B. The Relationships of BPEL and Others

Since our work is focused on BPEL, in order to clarify the BPEL position in the
web service stack, we will discuss the relationships between BPEL and WS-CDL, BPEL and OWL-S, as well as BPEL and WSDL.

- **BPEL and WS-CDL**

  Two main approaches are currently investigated for static service composition. The first approach, referred to as web service orchestration (e.g. BPEL), combines available services by adding a central coordinator that is responsible for invoking and combining the web services. The second approach, referred to as web service choreographer (e.g. WS-CDL), does not assume the exploitation of a central coordinator but rather it defines the conversation that should be undertaken by each participant. The aim is to model the peer-to-peer interactions among the collaborating services.

  Briefly speaking, BPEL aims to model the interactions of web services with respect to a central coordinator, while WS-CDL is in a layer above BPEL and provides a conversation protocol for the global interaction of web services.

- **BPEL and OWL-S**

  Both BPEL and OWL-S are workflow languages for modelling business processes composed of a set of service invocations. The structure defines the partial order of invocation of the services and their interactions. BPEL, as an industrial standard, aims to provide rich control flow structures to integrate existing web services in a flexible way. BPEL supports exception handling and compensations, while OWL-S does not define recovery protocols. Also, BPEL provides more mature execution engines. OWL-S, as one of the standards in the Semantic Web community, aims at fully automating all stages of the web services lifecycle. By using OWL-S, Artificial Intelligence (AI) planning techniques can be used to automate the service composition.

  Therefore, BPEL supports static service composition, with a focus on representing composition where information flow and the binding between services are known in advance. OWL-S supports dynamic service composition, with a focus on modelling the preconditions and post-conditions of the process so that the evolution of the domain
1.2 Motivation

can be logically inferred. It relies on ontologies to formalise the domain concepts which are shared among services. The research groups of Standford University and Carnegie Melon University have led the work in adapting BPEL for semantic web, such as OWL-S.

- BPEL and WSDL

BPEL has close relationships with WSDL. Interactions with services are modelled as partnerLinks. A partnerLink has a partnerLinkType, which defines which WSDL portType is used in a relationship with any given partner and which portType is used when a partner interacts with the process itself. The BPEL process is exposed as a service and therefore has its own WSDL interface. The relationships between web services are defined in the partnerRole and myRole attributes of the partnerLinkType. WSDL uses messages to define and carry data types. In a BPEL process, a variable is defined to carry a data type, which is declared in WSDL.

1.2 Motivation

Recent years have seen a rapid growth in the development of web services technology. However, some issues such as the composability, compatibility, conformance and substitutability, correctness, and coordination of service compositions have not yet been thoroughly investigated. For instance, conformance and correctness should be checked to find errors as early as possible in the workflow design phase.

BPEL activity relationships can be categorised into control-flow and data-flow. Since BPEL is a semi-formal flow language, various formal semantics have been proposed, so that BPEL models can be verified rigorously. However, most current formal models only focus on modelling BPEL control flow, and do not cover the BPEL data flow analysis.

There exist two kinds of interactions of BPEL: internal and external. The external interactions between BPEL models are by message passing. The internal interactions
1.3 Aims and Objectives

between activities of a BPEL model, which are modelled explicitly by control dependencies and implicitly by data sharing. Those internal interactions caused by data sharing will be omitted if an approach does not cover the BPEL data flow analysis.

Furthermore, there are fewer efforts in the literature in using behavioural web service models (e.g. BPEL models) as the test models for deriving test cases. To our knowledge, none of the prior research studies the verification and test case generation of both BPEL control-flows and data-flows in a unified approach.

1.3 Aims and Objectives

The objectives of this work are:

- To design a formal model that can cover most features of the BPEL, and cover BPEL internal and external interactions.
- To demonstrate that it is essential to separate verification and testing of BPEL control flow and data flow.
- To show that it is suitable to apply model checking as the test generation engine to generate test cases from BPEL models.

1.4 Methodology

Existing model checking tools can be reused for the purpose of verification and testing of BPEL. Our formal model is intended to be used by such verification tools. With model checking, a BPEL model can not only be a design model for verification, but also be a test model for deriving test cases. The formal semantics proposed to date for BPEL can be categorised as process algebra based, Petri-net based, and automata based. We follow the automata-based approach, in order to facilitate the use of model checking tools. We propose a Web Service Automaton (WSA), an extension of Mealy machines, which covers data, supports message passing communication, and adapts the asynchronous interleaving semantics. We justify the suitability of WSA for BPEL
on three counts. First, its multiple-input events capture most features of the BPEL language, while most automata-based formal models for BPEL only cover the core subset features of BPEL. Second, its message passing communication provides a uniform semantics for both BPEL internal and external interactions. Third, our model supports separate analyses of BPEL control and data flows.

Based on WSA, we provide a model checking based test case generation framework for BPEL. We support the application of both SPIN and NuSMV model checkers as the test generation engines, and we encode the conventional structural test coverage criteria into LTL and CTL temporal logic. State coverage, transition coverage, and predicate coverage are used for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. Two levels of test cases can be generated to test whether the BPEL processes conform to the functionality and whether individual operations conform to the WSDL interface models. The generated test cases are executed on the JUnit test execution engine.

1.5 Contributions

Our theoretical contribution is to propose a formal model to formalise BPEL semantics. This formalisation enables us to reason about properties of BPEL processes such as their correctness and compatibility. Our practical contribution is to provide an automatic verification and test case generation framework for BPEL and WSDL. The key benefits of our approach are:

1) The multiple-input events of our formal model can capture most features of BPEL language, including the most interesting scope-based fault handling and compensation handling features of BPEL, and also can reduce unnecessary machine state space.

2) The explicit data handling of our formal model enables separating control flow testing and data flow testing of BPEL.
3) The message passing communication mechanism of our formal model provides a uniform semantics for BPEL internal and external interactions.

4) The test framework can automatically generate BPEL based test cases and WSDL based test cases.

1.6 Thesis Structure

Chapter 1 introduces background information and key concepts of web services. Chapter 2 gives a review of relevant literature. Chapter 3 presents our formal model, its static and dynamic semantics, and discusses the model’s compatibility. Chapter 4 provides detailed analysis of BPEL features in our formal model. Chapter 5 outlines our automatic test framework. Chapter 6 introduces the tool architecture and evaluates the tool with case studies. Chapter 7 summarises the thesis with conclusions and suggestions for potential future work.
Chapter 2

Literature Review

There exists an extensive literature of related work on the formal semantics and verification of process models exists. In order to maintain the focus of this thesis, we concentrate on the verification and testing of BPEL based web service models in this chapter. The aim of this section is not to thoroughly compare all the existing approaches, but to point out the motivation of our proposed work by reviewing related work.

2.1 Formal Semantics and Verification of BPEL

To a service composer, it is desirable to be able to verify that the composition is well-formed. For example, that it does not contain any deadlock or livelock which would cause the composition not to terminate under certain conditions. It is possible to verify these properties using formal notations and existing verification tools. The verification tools have the advantage that they allow one to simulate and verify the behaviour of one's model at design time, thus enabling the detection and correction of errors as early as possible. As such, the model verification phase helps increase the reliability of web services. The works of [34, 41] provide good reviews for the current verification approaches. In the literature, there are three branches of formal models for BPEL: Petri nets, Process Algebra, and automata.
2.1 Formal Semantics and Verification of BPEL

2.1.1 Petri Nets based approaches

A Petri net \( N = (P,T,F) \) consists of places \( P \) nodes, transitions \( T \) nodes, and flow relation \( F \) as directed arcs to connect places with transitions. The current state of a Petri net is represented by a set of black tokens distributed over the places. The places from which an arc runs to (resp. from) a transition are called the input places (resp. output places) of the transition, respectively. A transition is enabled if all of its input places contain tokens. An enabled transition fires by removing the tokens from its input places and adds a specified number of tokens into each of its output places. One can map BPEL based web services to Petri nets by assigning activities to transitions and process states to places. Fig 2.1 shows an example of Petri nets modelling BPEL activities [46].

![Petri Nets Example](image)

Figure 2.1: An example of Petri nets for BPEL activities [46]

Each BPEL activity associates a Petri net with an input place \( r_i \) and an output place \( f_i \). The places \( r_i, s_i, c_i, \) and \( f_i \) denote states ready, started (running), completed, and finished, respectively. Transitions are of three types. The first type, denoted as unlabelled bars, is auxiliary transition. The solid bars denotes a transition which models internal actions for checking pre-conditions or evaluating post-conditions for activities; and the other bars are used to implement composite constructs. The second and third types are denoted as labeled boxes. The second type is substitution transition, which is an abstract representation of subnets for the enclosed BPEL activities. The third type
2.1 Formal Semantics and Verification of BPEL

is an activity transition which models the events or actions of a BPEL activity. On the left of the figure, the Petri net models a BPEL basic activity $X$. If $X$ is to receive a message $m$, the transition labeled with $X$ can be replaced by $?m$. On the right of the figure, the Petri net models a BPEL flow activity, when the flow is started, subnets $A, B$ will be executed, and the flow completes after $A, B$ are completed.

In a standard Petri net, tokens are indistinguishable. In Coloured Petri nets (CP-net) \[35\], every token has a value. CP-nets extend Petri nets with the primitives for defining data types (colour) and the manipulations of data values, thus a CP-net is more concise than a Petri net. Transition eligibility depends then on the availability of an appropriately colour token in all the input places of this transition. Likewise, the output of a transition is a specifically colour token.

Web service algebra is proposed in \[28\] to define a set of web service composition operators. The authors use Petri nets as the formal semantics for the proposed web service algebra. The works of \[46, 56\] present Petri net semantics for the control flow of BPEL, with consideration of BPEL advanced features such as fault handling, event handling, and compensation handling. In \[56\], the tool BPEL2PN is developed to map BPEL code to Petri nets, and model checker LoLA \[54\] is used to verify CTL temporal logic. The author of \[37\] extends the work of \[56\] by using Petri net to capture the global interactions between BPEL processes. In \[46\], the tool BPEL2PNML is developed to map BPEL to Petri nets, and a verification tool WofBPEL is used as the analysis engine. The paper discusses how to verify the activity reachability and some pre-defined BPEL constraints. As a summary, the above works abstract from data. As shown in our motivation example, it is important to consider BPEL data dependencies.

In \[61\], they claim to capture both BPEL control and data dependencies in CP-nets, and CPN tools \[2\] can be used to verify the process. However, the paper only shows how to map a core subset of BPEL to CP-nets. There is no discussion of how to capture BPEL data dependencies, and no concern of modelling faults or compensations. They summarise a set of properties of CP-nets to be checked and their corresponding meaning in the verification of web service composition, including boundness, deadlock-freedom,
liveness, fairness, home, and reachability.

In [62], they use CP-nets as the process composition models, and apply CPN tools as the verification engine. BPEL skeleton code can be generated from the process composition model. In the CP-nets models, messages (events) and process variables are represented by tokens. Abstract colour sets are declared for the messages and variables such that each colour set is kept small to speed up the analysis. They also use an algorithm to automatically derive the conversation protocol, which is also CP-net based, from the process composition models. The conversation protocol in their context only models the interactions between the service consumer and the service provider, and hides the internal process details such as those providing data manipulation and interaction with other service partners. Instead of verifying the BPEL process, their work focuses on designing a correct CP-net based model and generating a BPEL skeleton process.

Petri nets provide the constructions for specifying synchronisation of concurrent processes. Petri net adopts interleaving semantics for concurrency, and asynchronous communication. In Petri-nets, every transition is restricted to modelling a single event, which will heavily increase the model state space.

2.1.2 Process Algebras based approaches

Process algebraic service composition aims to introduce much simpler descriptions than other approaches. The underlying semantic foundation is based on labelled transition systems, i.e. automata. Many variants have been defined and the field comes with a rich body of literature. The most well-known process algebras are CCS [42], CSP [31], and LOTOS [14], and Pi-calculus [43].

Pi-calculus extends CCS with mobile-ability, in which the basic entity is a process - it can be an empty process; that is, a choice between several I/O operations and their continuations, a parallel composition, a recursive definition, and a recursive invocation. I/O operations can be input (received) or output (sent). For example, $x(y)$ denotes receiving tuple $y$ on channel $x$; $\bar{x}[y]$ denotes sending tuple $y$ on channel $x$. Dotted
notation specifies an action sequence. Parallel process composition is denoted with $A|B$. Several processes can execute in parallel and communicate using compatible channels. Describing services in such an abstract way allows us to reason about the composition’s correctness.

In [22], a two-way mapping is defined between BPEL and LOTOS, and the model checking toolbox CADP [21] is used as the verification engine. The mapping from LOTOS to BPEL does not preserve the structure of a process, due to the expressive and flexible structure of LOTOS. The disabling operator is used to capture the BPEL interruptions. In LOTOS, the processes communicate synchronously by rendezvous.

In [23], the process algebra FSP (Finite State Processes) developed by [38] is used for the BPEL semantics and the model checker LTSA [38] as the verification engine. The web service composition specification is modelled in an MSC (Message Sequence Chart), and the implementation is modelled in BPEL. Both MSC models and BPEL processes are translated into FSPs, such that the BPEL implementation can be verified against the MSC specification by trace equivalence checking. The work of [24] extends the earlier work to verify the interacting BPEL processes and checks their compatibility. A tool LTSA-WS was implemented as an Eclipse plug-in. FSP is abstract from data, so their mapping does not cover BPEL data dependencies. Also, FSP supports synchronous communication.

In [59], they use Pi-calculus as the BPEL formal model and NuSMV model checker as the verification engine. A tool, OPAL, is developed to automate the mapping from BPEL to Pi-calculus, and from Pi-calculus to the input language SMV of the NuSMV model checker. It points out that there exist two approaches to model check Pi-calculus. One is to analyse Pi-calculus processes based on a proof system, and the other is to transform Pi-calculus into automata. The authors follow the second approach. They model a shared variable $x$ of BPEL as a shared storage register $Reg(x)$, where the stored value can be read from or written to $Reg(x)$.

In [58], they propose a language named CDL to extend WSDL for modelling the behaviour of individual web services, and propose a composition language which can
support both centralised and distributed orchestrations. The formal semantics of these two languages are based on Pi-calculus. They point out that the use of shared variables in BPEL makes it difficult to coordinate the execution in a distributed manner. Their composition language has two core concepts: a task and a process. A task is equal to a BPEL activity. For inter-task dependency, they explicitly consider control dependencies and data dependencies. The review of web service data dependency modelling will be covered in section 2.2. They further point out that the current tools support for verifying of Pi-calculus is immature. Most do not support the complete language and require a complex and error prone input syntax. A solution is to map Pi-calculus to the input languages of mature model checkers such as SPIN. Since the input languages of most mature model checkers are automaton-based, we believe an automaton-based formal model is more suitable for those input languages. There are other works that also use Pi-calculus as the formal semantics for the orchestration language [19, 26, 40].

In the above Process Algebra approaches, they consider both the core BPEL activities and the advanced BPEL features with fault handling, compensation handling, and event handling. However, since the BPEL scope based fault and compensation handling mechanism is complex, only a few works give it in-depth analysis. In [50], Pu et al. propose a BPELO language to capture the BPEL scope based compensation handling features. They propose a n-bissimulation relation, which reflects the scope-based compensation mechanism, to define the equivalence between BPELO programs. An execution engine for BPELO is developed. In [16], Bulter et al. formalise the notions of compensation in a StAC language (Structured Activity Compensation). They model the BPEL control flow using StAC and the model the BPEL data manipulation in B notation. The semantics is clean and precise, but it is not clear how to verify or test such models in an automatic way.

### 2.1.3 Automata based approaches

Automaton is a well-known formalism for system specification. There are many different kinds of automata, including finite state machines (FSMs) such as Mealy and Moore

---

2.1 Formal Semantics and Verification of BPEL
machines, which are always finite state. Label Transition systems are automata, which are generally infinite state. An FSM is a machine that, given an input symbol, transfers it through a series of states according to a transition function. A Mealy machine is an FSM that generates an output based on its current state and the input. The transitions will include both an input and output signal. In contrast, a Moore machine is a FSM that generates an output based on its current state only. A transition is associated with an input signal. For both types of machines, an input is required to trigger a transition. For each Mealy machine, there is an equivalent Moore machine, and vice versa. Mealy machines are most commonly used, and there exists a variety of models extending Mealy machines with data, hierarchy, or time.

In [57], BPEL models are mapped into deterministic finite state automata for the matchmaking of web service composition. The STSs (State Transition Systems) are used in [49] to be the BPEL formal semantics, and a tool is developed as a part of the ASTRO toolset [1]. Both of these formal models are abstracted from data.

In [25], they propose guarded automata (GA) to be the formal models for both BPEL and the conversation protocol. GA extends Mealy machines with data, and every transition is equipped with a guard in the form of an XPath expression. The model checker SPIN is used as the verification engine. BPEL processes communicate by sending asynchronous messages, and each process has a queue. A global watcher keeps track of all messages. The conversation is introduced as a sequence of messages. They propose a set of sufficient conditions so that asynchronous communication can be replaced with synchronous communication. A tool WSAT is developed to map BPEL to guarded automata, and map guarded automata to Promela (the input language of SPIN). In their approach, each BPEL activity is mapped to a GA, so the BPEL process as a whole is a composition of a set of GAs. The interleaving semantics of concurrency is used. However, they omit the inter activity data dependencies. Also, in their models, the GAs representing BPEL processes communicate by message passing, but it is not clear how the GAs represents the BPEL activities communications. In our formal model, we believe it is clearer from the theoretical view to provide a single
communication mechanism for both external and internal interactions, where machines communicate by either message passing or by data sharing, but not both. In their mapping from GA to Promela, the XPath expressions in the GA transition guards are also translated into Promela, so that the data manipulation can be verified. We believe this will decrease the speed of model checkers, and that symbolic transition guards can reach the same verification result. We will discuss this in section 5.

Similar to [25], the author of [44] transforms BPEL into an extended FSM named EFA. A tool is developed to automate the mappings from BPEL to EFA, and from EFA to Promela. EFA extends a Mealy machine with data, and it adopts asynchronous interleaving semantics of concurrency. They also have no concern of the interactions among BPEL activities due to data dependencies. Furthermore, it is not clear which communication mechanism is used when they are modelling read and write data by BPEL activities.

The above automata based approaches, only cover core subsets of BPEL activities and do not consider fault handling, compensation handling, or event handling.

2.2 Analysis of Data Dependencies in Orchestration

Referring to the works discussed in the previous section, most existing work abstracts from data and focuses attention on the control flow. When data is omitted, the transition guards and variables are left out, so selecting one of two control paths, solved by the evaluation of data, needs to be modelled by a nondeterministic choice. Even for work that does consider BPEL data, data dependencies are not modelled in an explicit way. In this section, we review work with consideration of modelling data dependencies in the orchestration models such as BPEL models. The purpose of analysing data dependencies is to ensure that data is always defined before being used.

In [47], they propose a BioOPera Flow language to model the control dependencies and data dependencies between tasks (BPEL activities) as visual flow graphs. In order to maintain consistency, they provide a set of constraints when constructing a data flow
2.2 Analysis of Data Dependencies in Orchestration

For instance, in a process data flow graph, data always flows from output to input parameters of tasks. The input parameters of a process can only be connected to input parameters of tasks, and output parameters of the process may receive data only from output parameters of tasks. A constant variable can be connected to multiple input parameters, but an input parameter bound to a constant variable cannot have any other incoming data flow edge. A toolset is developed to support the visualisation. Even though their focus is not rigorous verification of design models, they show the importance of considering control and data dependencies in separation.

In the composition language proposed by [58], each task (equal to a BPEL activity) has an input Dependencies section to describe the control dependencies and data dependencies from itself to other tasks. For instance, variable $x$ is the output data of task $tk_1$ and the input data of task $tk_2$, $tk_2$ will declare a data dependency in its input Dependencies section to specify $tk_1$ as the source which sends $x$ to it. The task which receives a message from an external web service will send the message to other 'downstream' tasks which have dependencies on this message. Their composition language is mapped to Pi-calculus. In Pi-calculus, a process denotes a web service task, channels represent data dependencies, and control dependencies are represented implicitly using the operators of Pi-calculus directly. The composition service as a whole is modelled as a parallel composition of all of these processes. Their data dependency modelling makes the data definition and usage clear. With this in mind, our WSA should also be able to capture the data dependencies of BPEL activities in an explicit way.

A grid workflow language is proposed in [20], where each activity may have a data-in port and data-out port. The data exchange describes that the data flows from data-out ports to data-in ports. They discuss the constraints added on to the data exchanges in conditional activities (e.g. BPEL switch and pick), in sequential loop activities (e.g. BPEL while), and in parallel loop activities (e.g. BPEL flow).

The authors in [11] provide a model of data flow in addition to control flow for OWL-S process models. They transform OWL-S to Promela so that SPIN model checker can be used to verify the OWL-S process model. Their scope of the data flow
2.2 Analysis of Data Dependencies in Orchestration

is limited to within a composite process. The processes in a composite process can exchange data among themselves or with the parent process. In OWL-S, a process is similar to a BPEL activity. We also add such level-based constraint to the BPEL internal data exchanges, which will be discussed in section 4.2.

For external data exchanges between orchestration models, if there is a conversation protocol available, the data dependencies between web services can be directly derived from the conversation protocol; otherwise, one needs to analyse the data exchanges to get the data dependencies. The work of [15] discusses how to analyse data exchanges between YAWL workflow models, so that the resulting data dependencies between web services can be used for service matching. In [19], they propose a OWL-P language to model both the conversation protocol as well as the orchestration models. When composing orchestration models, the designer needs to define a set of composition axioms to add constraints on the conversation protocol. A data-flow axiom states the data exchange dependency among the orchestration models. In our test framework, we do not assume a conversation protocol is available, and the data dependencies between BPEL processes need to be analysed.

In [39], they propose data nets to capture data exchange and data manipulation within an orchestration model, as well as data exchange between composition models. The control flow of an orchestration model is modelled by the STS (State Transition System) [49]. STS with data is the synchronised product of all the STSs and data nets. A tool is needed to do the experimental evaluation. Since our WSA includes data, there is no need to add a separate data model. Data flows can be derived from WSAs based on existing data flow analysis techniques.

To sum up, it is necessary to model data exchange explicitly in our formal model, in order to capture the data dependencies within a BPEL model and the data dependencies between BPEL models in a correct way.
2.3 Testing of BPEL based web services

As we can see from the previous section, there is intensive research on providing precise semantics for BPEL and verification of BPEL models. However, there is less effort on using BPEL as the test models for deriving test cases.

A framework is proposed in [12] to augment WSDL with a UML2.0 PSM (Process State Machine) for modelling web service interactions. After transforming PSM to a Symbolic Transition System, existing ioco-conformance testing tools can be applied. In [29], they use Graph Transformation Rules along with WSDL to generate test cases. WSDL-S is proposed in [55] to be the service behaviour model, which extends WSDL by adding a pre-condition and post-condition to each WSDL operation. The WSDL-S is mapped to EFSM (extended FSM) so that the existing test techniques for EFSM can be applied. Yuan et al. [60, 63] propose a XCFG (extended control flow graph) to represent a BPEL process. First, they propose a DFS (depth first search) based algorithm to generate sequential test paths from the XCFG, according to branch coverage criterion. Second, the sequential test paths will be combined into concurrent test paths based on various BPEL structures. Finally, a constraint solver is used to remove the unexecutable test paths, and to generate test data. They assume that the BPEL eventHandlers can only have on message thread, and assume that there is no interruption due to either fault propagation or process termination.

There exist other works related to web service testing, but the focuses are not on test case generation. For instance, Zhu [66] analyses the problems specific to test web services, and introduces an ontology called STOWS for testing services. The testing services themselves are web services. Based on the ontology, the testing services can be registered, requested, or retrieved. The mapping from their ontology to the OWL-S is yet to be developed.

For the purpose of providing automated test case generation and execution environment, our test framework should allow users to choose test criteria in an easy way. Test case generation techniques can be categorised as test purpose oriented and coverage ori-
2.4 Multiple Inputs Automata

Test purpose oriented techniques allow testers to define interesting scenarios as test purposes, but the testers need to have knowledge to encode test purposes into temporal logic formulas for model checkers. Instead, coverage oriented techniques release testers from writing temporal logic formula manually. Therefore, we target coverage oriented test generation. It is not new to apply model checking to achieve test coverage [30, 33, 52], but it is new to apply such a technique in the domain of web services.

2.4 Multiple Inputs Automata

The concept of finite state machines (FSM) is widely used, and there is considerable similarity between FSM notations. In particular, most of the notations allow single-input-event transitions. For those complex systems, it may not be convenient to consider the FSM to have only single input events. Multiple inputs and multiple outputs may be a more natural model, such that the state machine can react to a set of simultaneous input events. We review some works in the literature that advocate multiple inputs state machine.

In the embedded system domain, Yun et al. [64] proposes a burst-mode Mealy machine to allow multiple-input changes in a burst fashion. The input events can arrive in arbitrary order and the outputs are generating concurrently. A transition consists of an input burst with a non-empty set of input events, and an output burst with a set of output events. In their earlier work, a transition is enabled after all the input events in a input burst have arrived. Two extensions were proposed. First, by allowing directed-don't-care input events in the input burst, as long as one input event in the input burst has arrived, the transition is enabled. Second, the transition guard is added, a transition is enabled when both the input events have arrived and the guard is evaluated to true. In our WSA, their don't care concepts can be modelled by linking events with logical OR, and the transition guard concept has been supported by most state machines that consider data handling.

Luttgen et al. [27] proposes a propositional logic (with logical operators such as
AND, OR, NOT, implies, and equivalent) based step semantics for Statecharts. A step consists of a source machine configuration, a transition, and the target machine configuration. The transition has a trigger and an action, where the trigger is a conjunction of events and negated events (e.g. $e_1 \land \neg e_2$) describing under which condition a transition is enabled. The logical AND together with NOT can model the implicit priorities on transitions imposed by the state hierarchy. In WSA, we include the logical AND and NOT to model the priority events.

Alexander [53] extends the state machine by adding regular expressions of events, aiming to reduce the complexity of FSMs. A FSM's transitions based on regular expressions containing Kleene closure (i.e. if $K$ is a set then $K^*$ is also a set) or concatenation are mapped into internal FSMs. In this way, the internal FSMs can add hidden states to the set of states of the original FSM. The extended FSMs are used as the source for Java code generation. If more than one regular expression matches an incoming event, the first transition is fired. As a common rule, the multiple transitions with event expressions whose languages overlap should be properly guarded. No mathematical model is provided in the paper. Their focus is to simplify the design model, by moving the complexity from the design model to the intermediate model.

In the domain of embedded system, Girault et al. [27] present a Ptolemy framework to integrate FSM with various synchronous concurrency models. In their FSM, the components of a transition include a guard and a set of output events, where the guard is represented as a Boolean expression (logical AND, OR, NOT) of the input events, and an event can be either present or absent. Their FSM also supports hierarchy, where a state may be further refined into another FSM, and the inside and outside FSMs are called slave and master respectively. In their transitions, guards are Boolean expressions over input events. Their work can be the foundation of our web service automaton in terms of adding Boolean expressions over input events. We will consider data handling, and our transitions should be associated with two kinds of guards, one adds constrains to input events, the other puts constrains on variables. As pointed out in their paper, the hierarchical FSM makes the model more intuitive and easy to
understand, but the hierarchy does not reduce the number of states. For the purpose of testing, the *composite* states need to be flatten to explore the execution paths. So, as an intermediate operational model, web service automaton does not support composite states. Also, instead of integrating the machine with synchronous concurrency models, web service automata will target at modelling the interaction between distributed web services, so asynchronous semantics will be applied.

In summary, due to the complexity of BPEL features, which include concurrency, fault propagation, and interruption, we believe it is more natural to allow multiple-input transitions.

### 2.5 Summary

It is hard to design the interactions of asynchronously executing web services in a consistent and complete manner, so it is essential to apply automated tools in finding the inconsistencies and checking the functionalities. It is a challenge to test BPEL processes because of its semi-formal nature and complexity. Therefore, we need a suitable formalism for the BPEL language, as well as a suitable model-based-testing technique for testing the BPEL processes automatically.

Both Petri nets and process algebras are precise and well-studied formalisms that allow automatic verification of certain properties of their behaviours. Likewise, they provide a rich theory on *bi-simulation* analysis; that is to say, one can establish whether two processes have equivalent behaviours. Such analysis is useful to establish whether a service can substitute another service in a composition or to verify the redundancy of a service. Process algebra approaches have the advantage that the composition operators of process algebras are convenient in capturing the semantics of BPEL structured activities, and they support simulation and bi-simulation analyses, which are useful for model substitution and refinement. The limitation of Petri-net based approaches is the single event restriction on every transition, which will easily cause large model state space. In order to apply model checking, process algebra based approaches normally...
need three layers: the process algebra models, the automaton models, and the model checker adapters. The automata provide the operational semantics of process algebra. On one hand, this additional automata layer can reduce the cost of verifying process algebra model by model checkers, because automata are easily manipulated by graph transversal and searching strategies. On the other hand, it enables the use of wider choices of model checking engines.

As a result, for the purpose of verification and testing, the automaton formalism is especially attractive due to the straight usage of model checking tools. The model substitution and refinement is not the focus of our test framework. Therefore, we have investigated the usability of automata approaches. Thus we only need two layers for model checking: the automata models and the model checker adapters. However, most Petri net based and process algebraic models can handle compensations and exception handling, but this remains to be seen for the automata based models. Our automata based formal model needs to fill this gap. Also, due to the complexity of BPEL features, allowing multiple-input transitions is a natural choice to reduce the model state space and complexity. To summarise, in order to solve some of the current problems, it is essential to design a formal model which fulfils the following requirements: 1) The formal model should be able to fill the gap of current automata based approaches to capture not only the basic BPEL features but also the advanced BPEL features to model fault handling, event handling, and compensation handling; 2) The formal model should be able to capture both control dependencies and data dependencies between BPEL activities; 3) The formal model should reduce state space for model checking; 4) The mapping from the formal model to the input language of model checkers should contain abstraction to speed up the model checking.

In this chapter, we reviewed the formal models for BPEL including Petri-nets, Process Algebra, and automata; the verification approaches based on these formal models; the testing approaches of deriving test cases from web service behavioural models; and approaches to analyse data dependencies internal and external to an orchestration model. We discussed the motivation to propose a formal model, and the required fea-
2.5 Summary

tures of the formal model. Thereafter, we presented the reason to design and develop an automated test generation framework. In the next chapter, we will present the formal semantics of our proposed web service automaton.
Chapter 3

Web Service Automata

The previous chapter identified a number of problems with existing formal models of BPEL. In order to address these, we propose a new formalism: Web Service Automata. In this chapter, we provide the formal semantics for web service automaton (WSA). Web service automaton provides a formal description of the legal protocol for web service interaction, and following some translation steps it can be used as a reference model for test case derivation, applying well established algorithms from formal test theory. In WSA, each message is represented by a channel. Unique names for channels corresponding to each message type are assumed.

3.1 Static Semantics

The static semantics of a web service automaton extends a finite state machine with signature, data structure, and message storage schema. The dynamic semantics of a web service automaton includes machine configurations and execution traces. The formal definitions are given below.

**Definition 1.** A Web Service Automaton (WSA) $M$ is a sextuple $M = (I_M, S_M, s_{0,M}, S_fM, T_M, \delta_M)$. As a convention, we omit the subscript of $M$ so that $M = (I, S, s_0, S_f, T, \delta)$.

1) $I$ is the signature of $M$. denoted as a triple $I = (E, L, O)$, where $E, L, O$ are pair-wise disjoint and represent sets of input, internal, and output events, respectively.
3.1 Static Semantics

Let \( M_{sg} = (L \cup E \cup O) \) be the set of events, we assume that \( L \) is the disjoint union of a set \( L_{in} \) of internal input events and a set \( L_{out} \) of internal output events, and the elements of \( (E \cup O) \) will be called external events.

2) \( S \) is a set of states, \( s_0 \in S \) is the initial state, \( S_f \subseteq S \) is a set of final states.

3) \( T \subseteq (EX \cup \{\Omega\}) \times BX \times (p(AX \cup O \cup L_{out}) \cup \{\Omega\}) \) is a set of transitions. where:

- \( EX \) is the set of Boolean expressions over input event sets \( E \cup L_{in} \), linked by logical operators AND, OR, and NOT, denoted as \( \land, \lor, \) and \( \neg \) respectively. Let \( V \) be a countable infinite set of variables of \( M \). \( AX \) is the set of assignments over \( V \). \( BX \) is the set of Boolean expressions over \( V \).

- For each transition \( t = (ex, g, a) \in T \) (graphically denoted as \( e.r \), \( e.r \in EX \cup \{\Omega\} \) is the input event expression, \( g \in BX \) is the guard predicate, and \( a \subseteq p(AX \cup O \cup L_{out}) \cup \{\Omega\} \) is the action set composed of assignments and output events. \( \Omega \) indicates the omission of an input event expression or an output event. The components of transition \( t \) are denoted as \( t.ex = ex, t.g = g, t.a = a \).

- If there exist two statements \( st_1, st_2 \in AX \) where \( def(st_1) = def(st_2) \), then \( st_1 \equiv st_2 \) (see the definition of \( def \) below).

4) \( \delta \subseteq S \times T \times S \) is the transition relation (graphically denoted as \( s \xrightarrow{t} s' \)). If \( s \xrightarrow{t} s' \) with \( t = (ex, g, a) \), then if the machine is in state \( s \), the \( t.ex \) and \( t.g \) are evaluated to true (see the definitions of \( eval_t, eval_G \) below), then the machine executes the set of instructions \( a \) and change state to \( s' \).

Let \( st \) denotes a statement that represents an input event of machine \( M \), output event of \( M \), assignment, or Boolean expression. First, we define three functions:

- \( def : (AX \cup E_M) \rightarrow p(V) \), where \( def(st) \subseteq p(V) \) returns the assigned variables of a statement, i.e. the variable on the left hand side of an assignment, and the input event parameters of \( M \).
• **cases**: $(AX \cup O_M) \rightarrow \varphi(V)$, where $\text{cases}(st) \subseteq V$ returns the variables on the right hand side of an assignment, and the output event parameters of $M$.

• **puses**: $BX \rightarrow \varphi(V)$, where $\text{puses}(st) \subseteq V$ returns the variables in the Boolean expression over variables.

Next, we define two functions for assigning values to variables and for evaluating whether an event in the input event expression exists in the machine input buffer:

• Let $V$ be a countable infinite set of variables and a set $D$ of values. $Env_V$ is the set of all functions $\epsilon_V : V \rightarrow D$, where an element $\epsilon_V \in Env_V$ represents the current values of variables in some system configuration (see definition 6). Let $ST$ be a set of statements. $\epsilon^*_V : ST \rightarrow D$ evaluates a statement to obvious values in $D$ by applying $\epsilon_V$ to any free variables in the statement. For instance, for a statement $y = f(x_1, x_2,.., x_n)$, we have $\epsilon^*_V(y) = f(\epsilon_V(x_1), \epsilon_V(x_2),.., \epsilon_V(x_n))$.

• Let $\beta(E \cup L_{in})$ be the multi-set input buffer of $M$ (see definition 5). $Env_I$ is the set of all functions $\epsilon_I : E \rightarrow \{\text{true, false}\}$ that satisfies $\epsilon_I(e) = \text{true}$ iff $e \in \beta(E \cup L_{in})$. Let $ST$ be a set of statements. $\epsilon^*_I : ST \rightarrow \{\text{true, false}\}$ evaluates a statement to $\{\text{true, false}\}$ by applying $\epsilon_I$ to any free variables in the statement. For instance, for an event expression $t.ex = f(e_1, e_2,.., e_n)$, we have $\epsilon^*_I(t.ex) = f(\epsilon_I(e_1), \epsilon_I(e_2),.., \epsilon_I(x_n))$.

Accordingly to the above functions, we define an event Boolean function, a guard Boolean function, and a variable update function to evaluate the current values of $t.ex$, $t.g$, and $t.a \cap AX$ for a transition. Let $x \in V$:

1) We define a function $ev : EX \rightarrow E \cup L_{in}$ to return the input event set of an event expression. For instance, for an event expression $t.ex = (e_1 \wedge \neg e_2) \lor (e_2 \wedge \neg e_1)$, the input event set $ev(t.ex) = \{e_1, e_2\}$. Let $ev(t.ex) = \{e_1,.., e_n\}$. We define an event Boolean function $eval_I : EX \times Env_I \rightarrow \{\text{true, false}\}$ that satisfies: $eval_I(t.ex, \epsilon_I)(e_1,.., e_n) = \text{true}$ iff $\epsilon^*_I(t.ex)$ holds.
2) Let \( X = \{x_1, \ldots, x_n\} \) be the set of variables in \( st \in BX \). We define a guard Boolean function \( \text{eval}_G : BX \times Env \rightarrow \{\text{true}, \text{false}\} \) that satisfies: \( \text{eval}_G(st, \epsilon_V)(x_1, \ldots, x_n) = \text{true} \) iff \( \epsilon_V(st) \) holds. For instance, \( \text{eval}_G(x < y, \epsilon_V)(x, y) = \text{true} \) iff \( \epsilon_V(x) < \epsilon_V(y) \).

3) Let \( st_i \in t.a \cap AX \). We define a variable update function \( \text{eval}_V : AX \times Env \rightarrow Env \) that applies assignments in the obvious way. Note this well have following properties:

- If there exists \( x \in \text{cuses}(st) \) and \( x \notin \text{def}(st) \), then \( \text{eval}_V(st, \epsilon_V)(x) = \epsilon_V(x) \).
  It indicates that the value of \( x \) is unchanged when \( x \) is used and not defined.

- If for all \( \epsilon_V(x) \in Env \) we have \( \text{eval}_V(st_1, \epsilon_V)(x) = \text{eval}_V(st_2, \epsilon_V)(x) \), then \( st_1 \equiv st_2 \). It indicates that given an \( \epsilon_V(x) \) and two statements, if the updated values of \( x \) are equivalent, the two expressions are equivalent.

- Since a variable can only be defined once in a transition, this allows us to unambiguously define a set of statements \( X \subseteq t.a \cap AX \), \( \text{eval}_V(X, \epsilon_V)(x) = \begin{cases} \text{eval}_V(st, \epsilon_V)(x) = \epsilon_V(st) & \text{if } x \in \text{def}(st) \\ \epsilon_V(x) & \text{otherwise} \end{cases} \).

**Definition 2.** The data structure of machine \( M \) is a triple \((V_M, AX_M \cup E_M \cup O_M, BX_M)\), where \( AX_M, BX_M \) can be retrieved from the transition set \( T_M \). \( AX_M = \{st \in AX | \exists t \in T.st \in t.a\} \) and \( BX_M = \{st \in BX | \exists t \in T.st \in t.g\} \). \( V_M \) is the union of \( \bigcup_{st \in AX_M \cup E_M \cup O_M} (\text{def}(st) \cup \text{cuses}(st)) \) and \( \bigcup_{st \in BX_M} \text{puses}(st) \).

**Definition 3.** Suppose that transitions \( t_i, t_j \) begin with the same start state \( s \), and end with states \( s', s'' \) respectively. A WSA is deterministic iff the guard is non-overlapping \( t_i.g \oplus t_j.g \) in the case of \( t_i.c = t_j.c \). Consequently, in a deterministic WSA, \( \forall s, s', s'' \in S, \forall t \in T \), the condition \( s \xrightarrow{t} s' \land s \xrightarrow{t} s'' \Rightarrow s' = s'' \) holds.

We use symbols \( ?, !, @ \) as a convention in diagrams to indicate whether an event is input, output, or internal event, denoted as \( ?e \in E, !e \in O, @e \in L \), respectively. Conversely, going from the formal description to a diagram, \( ?, !, @ \) are introduced depending on membership of \( E, O, L \). For instance if \( M \) sends a message \( m \) to \( M' \), then
3.2 Dynamic Semantics

in the composite automaton (definition 11), ?m and !m will become @?m and @!m to indicate that the external input and output events become the internal input and output events, respectively.

**Definition 4.** We define a message $x$ to be a pair of send and receive events ($!x, ?x$). If machine $M_1$ sends message $x$ to machine $M_2$, then $!x \in O_1 \land ?x \in E_2$.

3.2 Dynamic Semantics

We have defined the static structure of a WSA. We now explain its operation semantics for dynamic semantics. We assume, first of all, that each WSA is equipped with a finite **multi-set** buffer (definition 5) $\beta(E \cup L_m)$ to store the incoming messages. In one step of the WSA, either: 1) a message $e$ is received from the environment $e \in E$, and added to the buffer $\beta(E \cup L_m)$; or 2) an enabled transition fires, possibly causing a state change and change to the values of some variables $v \in V_M$. In order to define a step precisely, we need to formalise the notion of a **configuration** $\eta$ which records the current state, the current values of the variables associated with $M$ and the current content of the buffer.

A transition $t$ is enabled in a configuration $\eta$, when: 1) its triggering event expression $t.ex$ is either empty, or is evaluated to true $eval_t(t.ex, e_1, ..., e_n) = true$ where $ev(t.ex) = \{e_1, ..., e_n\}$; and 2) its guard is evaluated to true $eval_g(t.g, e_1, ..., e_n, x_1, ..., x_n) = true$ where $\{x_1, ..., x_n\}$ is the set of variables in $t.g$. When a transition $t$ is enabled, a set of actions $t.a$ is executed, and the state machine moves from the start state to the end state of $t$. Such a transition is an **enabled transition**.

The commonly used buffering schemes can be categorised as two branches: single FIFO and multi-set buffering schemes. In the single FIFO buffering scheme, each machine has one FIFO buffer. This scheme is applied in CFSM [13] and the guarded automata [25]. A message can be consumed only when it is at the head of the FIFO buffer. There exist three types of multi-set (definition 5) buffering schemes, as follows.

1) Multi-set buffer with FIFO sub-buffers: A machine has one multi-set buffer that
3.2 Dynamic Semantics

consists of FIFO sub-buffers, where each FIFO sub-buffer corresponds to a message type. A message can be consumed when it is at the head of each FIFO sub-buffer.

2) Multi-set buffer without sub-buffers: A machine has one multi-set buffer. A message can be consumed as long as it is in the buffer. For those messages of the same type, the machine will randomly consume a message in the buffer.

3) Multi-set buffer with multi-set sub-buffers: A machine has one multi-set buffer that consists of multi-set sub-buffers, where each multi-set sub-buffer corresponds to a message type. The machine consumes messages in the same way as above 2).

The multi-set buffer with FIFO sub-buffers, i.e. type 1), is used in WSA. Note that without loss of generality, we assume the communication channels are lossless, so that every message is always received after it is sent.

A machine is associated with a set of events, and an event may have multiple event occurrences. When considering a machine's dynamic behaviour, we need to distinguish event occurrences from events. Similarly, a message sent between machines may have multiple message instances, and message instances also need to be distinguished from messages. We define a function $\lambda : Ins \rightarrow C$ to map class instances $Ins$ to classes $C$. 1) When applying $\lambda$ to event occurrences and events, $\lambda(o) = e$ returns the event $e$ for an event occurrence $o$. For an enabled transition $t$, if $?x \in t.m$ then we say $t$ corresponds to the event occurrence $o$, with $\lambda(o) = ?x$. 2) When applying $\lambda$ to message instances and messages, for a message instance $om = (!om, ?om)$ there exists message $m = (!m, ?m)$ such that $\lambda(!om) = !m$ and $\lambda(?om) = ?m$.

During machine communication, message overtaking may occur. Figure 3.1 shows two types of message overtaking in MSCs (Message Sequence Charts). For type-A, suppose messages $x = (!x, ?x)$ and $y = (!y, ?y)$, there exist message instances $ox = (!ox, ?ox)$ and $oy = (!oy, ?oy)$ where $\lambda(ox) = x$ and $\lambda(oy) = y$, respectively. Message overtaking happens when the order of receive event occurrences $?ox, ?oy$ is different.
3.2 Dynamic Semantics

from the order of send event occurrences \( l_{ox}, l_{oy} \). For type-B, suppose message \( x = (l_{x}, r_{x}) \), there exist message instances \( ox_1 = (l_{ox_1}, r_{ox_1}) \) and \( ox_2 = (l_{ox_2}, r_{ox_2}) \) where \( \lambda(ox_1) = \lambda(ox_2) = x \). Message overtaking happens when the order of receive event occurrences \( r_{ox_1}, r_{ox_2} \) is different from the order of send event occurrences \( l_{ox_1}, l_{ox_2} \).

Different buffering schemes allow different message overtaking. For the single FIFO buffering scheme, no message overtaking is allowed. For multi-set buffering schemes, the type-A message overtaking is allowed in the buffers of type 1), and both type-A and type-B message overtakings are allowed in the buffers of type 2) and type 3).

**Definition 5.** A finite multi-set is formally defined as a pair \((X, n)\), where \( X \) is some set and \( n : X \to N \) is a function from \( X \) to the set \( N \) of natural numbers. A multi-set differs from a set in that each element has a multiplicity.

For instance, \( \{a, b, b, c, b, a\} \) is a multiset, and the multiplicities are \( n(a) = 2, n(b) = 3, n(c) = 1 \), respectively. The usual set operations such as union, intersection, and sum can be generalised for multisets. We have \((D + E)(x) = D(x) + E(x), (D + y)(x) = D(x)\) if \( x \neq y \) and \((D + y)(y) = D(y) + 1\).

**Assumption 1.** A final state of a machine has no outgoing transitions. A machine \( M \) terminates when it reaches one of its final states, i.e. the lifecycle of \( M \) ends. We assume that the buffer is always empty when \( M \) is in a final state. If the buffer is not empty then the remaining messages will be discarded because they cannot be consumed within this lifecycle.

**Definition 6.** A configuration of a machine \( M \) is of the form \( \eta = (s, B, \epsilon_V) \), where \( s \in S, B \subseteq \beta(E \cup L_m) \) is the current buffer content, and function \( \epsilon_V \in Env \). The
3.2 Dynamic Semantics

set of configurations of $M$ is denoted as $confs(M)$. Let $\epsilon_0$ and $\epsilon_1$ be the variable values when the machine is in the initial state $s_0$, and a final state $s_n \in S_f$, respectively. $\eta_0 = (s_0, \emptyset, \epsilon_0)$ is the initial configuration, and $\eta_n = (s_n, \emptyset, \epsilon_1)$ is a final configuration.

**Definition 7.** A **step** is a triple $(\eta, x, \eta') \in confs(M) \times (T \cup E \cup L_{in}) \times confs(M)$. A step $\eta \rightarrow x \eta'$ changes machine $M$ from configuration $\eta$ to configuration $\eta'$ in the following forms:

1) If $x \in E \cup L_{in}$, representing the case when an arriving message $x$ is added to the buffer $B$, then $(s, B, \epsilon_0) \rightarrow x (s', B', \epsilon_1)$ iff $s = s'$, $\epsilon_0 = \epsilon_1$, $B' = B + \{x\}$.

2) If $x = t \in T$, representing the case when either the message set $\epsilon_1(t.a)$ is already in the buffer or there is no triggering event associated with $t$, then $(s, B, \epsilon_1) \rightarrow t (s', B', \epsilon_1)$ iff

   - $(s, t, s') \in \delta, \text{eval}_0(t.g, \epsilon_1) = true, B' = (B \epsilon(t.ex) + (t.a \cap L_{in}))$ where $\text{eval}_t(t.ex, \epsilon_1)(e_1, ..., e_n) = true$ where $\text{eval}(t.ex) = \{e_1, ..., e_n\}$.
   - According to 1) of definition 2, in the definition of $(s, B, \epsilon_1) \rightarrow t (s', B', \epsilon_1)$, we have $\epsilon_1 = \text{eval}_a(t.a \cap AX, \epsilon_1).

**Definition 8.** A transition sequence of $M$ is a sequence of enabled transitions $\text{tran}(\alpha) = (t_0, .., t_{n-1})$, where $t_i \in T, \alpha = (\eta_0, t_0, .., \eta_n), \eta_i \rightarrow t_i \eta_{i+1}$ for $i = 0, .., n$, and $\text{tran}(\eta_0) = \emptyset$. Here $\eta_0$ is the initial configuration and $\eta_n$ is a final configuration. We denote the set of transition sequences by $\text{trans}(M, \eta_0)$ or $\text{trans}(M)$.

We define a **trace** of machine $M$ to be a non-empty sequence of external event occurrences $\text{trace}(\alpha) = (o_1, .., o_n)$, where for all enabled transitions $t$ of a transition sequence, $\lambda(o_i) \in \epsilon(t.ex) \subseteq E$ and $\epsilon_1(\lambda(o_i)) = true$, or $\lambda(o_i) \in t.a \cap O \neq \emptyset$ holds. We denote the set of traces by $\text{traces}(M, \eta_0)$ or $\text{traces}(M)$.

**Definition 9.** Let $SReach(M) = \{s \in S | \exists o \in \text{traces}(M). s_0 \xrightarrow{o} s\}$ be the set of states reachable from the initial state of $M$, and let $TReach(M) = \{t \in T | \exists s \in SReach(M). \exists s' \in S. s \xrightarrow{t} s'\}$ be the set of reachable transitions. So a machine $M$ is fully reachable iff $SReach(M) = S \wedge TReach(M) = T$. 

33
3.3 Composition

In order for our notion of composition to be commutative and associative, we introduce the unordered Cartesian product (definition 10). The commutative property is easy to obtain based on the unordered Cartesian product, but the associative property requires further constraints on the individual WSAs. We define a general composite automaton with an interleaving semantics (definition 11). Such a composite automaton has a set of properties (lemma 1, lemma 2) but it is not guaranteed to be a WSA (definition 12). We then elicit a set of constraints on the individual WSAs for a composite automaton to be a WSA (corollary 1, theorem 1) or to be a deterministic WSA (proposition 1, theorem 2). For a composite automaton composed by strongly composable WSAs (definition 13), we derive its property (proposition 2), also we prove that the buffer content of such a composite automaton is a unique pair (lemma 3) and both the configurations and configuration sequences of individual WSAs can be projected from the composite automaton (lemma 4, lemma 5). We also prove that the composition operator is commutative (lemma 6), and it is associative when the composed WSAs are strongly composable (lemma 7). In the following, we suppose \( M_1, M_2 \) are WSAs.

**Definition 10.** If \((X_i)_{i \in I}\) is an indexed family of sets, we define \( \Pi_{i \in I} X_i \) to be the set of all functions \( \omega : I \to \bigcup_{i \in I} X_i \) satisfying \( \omega(i) \in X_i \) for each \( i \in I \). We refer to \( \Pi_{i \in I} X_i \) as the unordered Cartesian product of a set of \( X_i \). Note that \( \Pi \) is a commutative and associative binary operator.

**Definition 11.** We define the Composite Automaton \( \hat{M} = M_1 \parallel M_2 \) to be the structure \( \hat{M} = (\hat{I}, \hat{S}, \hat{s}_0, \hat{S}_f, \hat{I}, \hat{\delta}) \), where

1) \( \hat{I} = (\hat{E}, \hat{L}, \hat{O}) \), we define \( \text{com12} \) to be the common messages of \( M_1, M_2 \) by \( \text{com12} = (E_1 \cap O_2) \cup (E_2 \cap O_1) \). Now we can define \( \hat{I} \) by \( \hat{E} = E_1 \cup E_2 - \text{com12} = (E_1 - Q_2) \cup (E_2 - O_1) \), \( \hat{L} = L_1 \cup L_2 \cup \text{com12} \), and \( \hat{O} = O_1 \cup O_2 - \text{com12} = (O_1 - E_2) \cup (O_2 - E_1) \).

2) \( \hat{S} = S_1 \parallel S_2 \).

3) \( \hat{s}_0 = \{s_0^1\} \parallel \{s_0^2\} \).
3.3 Composition

4) \( \hat{S}_f = \{ S_j^1 \} \prod \{ S_j^2 \} \). A state of \( M_1 \parallel M_2, \ldots \parallel M_n \) is final if, for all machines, the local state \( s_i \) of \( M_i \) is final.

5) \( \hat{T} = T_1 \cup T_2 \). Note we assume that for all \( v \in V_i \), \( v \) is a local variable of machine \( M_i \). There are no shared variables between \( t \in T_i \) and \( t \in T_j \) where \( i \neq j \). Let \( msg(x) \) be the message with \( msg \) as the message name and \( x \) as the message parameter. Given that \( tmp = msg(x) \in com_{12} \), i.e. \( !msg(x) \in O_1, ?msg(x) \in E_2 \), there exists \( t \in T_1 \) with \( msg(x) \in ev(t.a) \), and \( t \in T_2 \) with \( msg(x) \in ev(t.e) \). After \( M_1 \) have sent message \( msg(x) \) to \( M_2 \), \( msg(x) \) is appended to the input buffer of \( M_2 \), and \( E_{I_2}(msg(x)) = \text{true} \). Here \( x \) is an output parameter of \( M_1 \) and an input parameter of \( M_2 \).

6) \( \delta \subseteq \hat{S} \times \hat{T} \times \hat{S} \). If \( t \in T_1 \) then \( (s^1_i, s^2_i) \xrightarrow{t} (s^1_j, s^2_j) \leftrightarrow s^2_i = s^2_j \land s^1_i \xrightarrow{t} s^1_j \), similarly if \( t \in T_2 \) then \( (s^1_i, s^2_i) \xrightarrow{t} (s^1_j, s^2_j) \leftrightarrow s^1_i = s^1_j \land s^2_i \xrightarrow{t} s^2_j \). It follows from the asynchronous interleaving semantics that a transition of the composite automaton is either from \( M \) or \( M' \) but not from both machines.

**Lemma 1.** If \( M = M_1 \parallel M_2 \), then (1) \( \hat{E} \cap \hat{L} = (E_1 \cap L_2) \cup (E_2 \cap L_1) - com_{12} \); (2) \( \hat{O} \cap \hat{L} = (O_1 \cap L_2) \cup (O_2 \cap L_1) - com_{12} \); (3) \( \hat{E} \cap \hat{O} = \emptyset \).

**Proof.** By \( \hat{E} \cap \hat{L} = (E_1 \cup E_2 - com_{12}) \cap (L_1 \cup L_2 \cup com_{12}) \), and pair-wise disjoint property \( E_1 \cap L_1 = \emptyset \) we can get (1). Also, (2) holds because of the symmetry of \( \hat{E}, \hat{O} \).

From \( \hat{E} \cap \hat{O} = (E_1 \cup E_2 - com_{12}) \cap (O_1 \cup O_2 - com_{12}) \), by pair-wise disjoint property \( E_i \cap O_i = \emptyset \), we can get (3).

**Corollary 1.** If \( E_1 \cap E_2 = L_1 \cap L_2 = O_1 \cap O_2 = \emptyset \) (1), then \( M = M_1 \parallel M_2 \) is a WSA.

**Proof.** Expanding \( Msg_1 = E_1 \cup L_1 \cup O_1 \) and \( Msg_2 = E_2 \cup L_2 \cup O_2 \), we get \( Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap L_2) \cup (L_1 \cap E_2) \cup (O_1 \cap L_2) \cup (L_1 \cap E_2) \cup (E_1 \cap L_2) \cup (O_1 \cap O_2) \) (2). If (1) holds, from (2) we can deduce \( (E_1 \cap L_2) \cup (L_1 \cap E_2) \cup (O_1 \cap L_2) \cup (O_1 \cap O_2) \subseteq com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \) (3). If \( x \in (E_1 \cap L_2) \cup (E_2 \cap L_1) \) then \( x \notin com_{12} \) where \( com_{12} = (E_1 \cap O_2) \cup (E_2 \cap O_1) \), because of the pair-wise disjoint \( O_i \cap L_i = \emptyset \). So from (3) we can derive \( (E_1 \cap L_2) \cup (E_2 \cap L_1) \subseteq (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \). By (1) of lemma 1, we have \( \hat{E} \cap \hat{L} \subseteq (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \cap com_{12} \). Based on
3.3 Composition

the pair-wise disjoint of $E_i, L_i, O_i$, if $x \in \hat{E} \cap \hat{L} = (E_1 \cap L_2) \cup (E_2 \cap L_1) \setminus \text{com}_{12}$ then $x \notin (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \setminus \text{com}_{12}$. So we have $\hat{E} \cap \hat{L} \cap ((E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)) = \emptyset$. Based on set theory $A \cap B = \emptyset$ and $A \subseteq B \Rightarrow A = \emptyset$, we have $\hat{E} \cap \hat{L} = \emptyset$. Similarly, $\hat{O} \cap \hat{L} = \emptyset$ holds. Finally, by (3) of lemma 1, we have $\hat{E} \cap \hat{O} = \emptyset$.

Definition 12. $M_1, M_2$ are composable iff $L_1 \cap O_2 = L_1 \cap E_2 = L_2 \cap O_1 = L_2 \cap E_1 = \emptyset$, where the internal events of each machine are disjoint from the external inputs and outputs of the other machine.

Definition 13. $M_1, M_2$ are strongly composable iff 1) $M_1, M_2$ are composable; 2) $E_1 \cap E_2 = O_1 \cap O_2 = \emptyset$, where the inputs and outputs of each machine are pair-wise disjoint from the inputs and outputs of the other machine, respectively.

Theorem 1. $\hat{M} = M_1 \parallel M_2$ is a WSA iff $M_1, M_2$ are composable, i.e. $L_1 \cap O_2 = L_1 \cap E_2 = L_2 \cap O_1 = L_2 \cap E_1 = \emptyset$ (1).

Proof. From definition 11, $\hat{E} = (E_1 - Q_2) \cup (E_2 - O_1)$, $\hat{L} = L_1 \cup L_2 \cup \text{com}_{12}$, $\hat{O} = (O_1 - E_2) \cup (O_2 - E_1)$.

Given that $\hat{M}$ is a WSA, i.e. $\hat{E}, \hat{L}, \hat{O}$ are pairwise disjoint, we need to prove (1). $E_1, O_1$). We have $\hat{E} \cap \hat{L} = \emptyset \Rightarrow ((E_1 - Q_2) \cup (E_2 - O_1)) \cap (L_1 \cup L_2 \cup \text{com}_{12}) = \emptyset$. Based on set theory $(A \cup B) \cap (C \cup D) = \emptyset \Rightarrow A \cap C = B \cap D = A \cap D = B \cap D = \emptyset$, we have $L_1 \cap (E_2 - O_1) = \emptyset \Rightarrow L_1 \cap E_2 = \emptyset$ and $L_2 \cap (E_1 - O_2) = \emptyset \Rightarrow L_2 \cap E_1 = \emptyset$. Because $O$ is symmetrical to $E$, $L_1 \cap O_2 = \emptyset$ and $L_2 \cap O_1 = \emptyset$ can be proved similarly. So (1) is proved.

Conversely, given (1), we need to prove that $\hat{E}, \hat{L}, \hat{O}$ are pairwise disjoint. By (1), we have $\hat{E} \cap \hat{L} = ((E_1 - Q_2) \cup (E_2 - O_1)) \cap (L_1 \cup L_2 \cup \text{com}_{12}) = ((E_1 - Q_2) \cup (E_2 - O_1)) \cap \text{com}_{12}$, where $\text{com}_{12} = (E_1 \cap O_2) \cup (E_2 \cap O_1)$. Based on set theory $(A \cap B) \cap (A - B) = \emptyset$, we have $((E_1 - Q_2) \cup (E_2 - O_1)) \cap ((E_1 \cap O_2) \cup (E_2 \cap O_1)) = ((E_1 \cap O_2) \cap (E_2 - O_1)) \cup ((E_2 \cap O_1) \cap (E_1 - O_2))$. By pairwise disjointment of $E_i, O_i$, we have $\hat{E} \cap \hat{L} = \emptyset$. Similarly, $\hat{O} \cap \hat{L} = \emptyset$ can be proved. Finally, $\hat{E} \cap \hat{O} = ((E_1 - Q_2) \cup (E_2 - O_1)) \cap ((O_1 - E_2) \cup (O_2 - E_1)) = \emptyset$. So, $\hat{E}, \hat{L}, \hat{O}$ are proved to be pairwise disjoint.
Lemma 2. If $\bar{M} = M_1 \parallel M_2$ is a WSA where $M_1, M_2$ are strongly composable, then $T_1 \cap T_2 = \emptyset$.

**Proof.** By definition 1, $T_1 \subseteq (Ex_1 \cup \{\Omega\}) \times BX_1 \times (p(Ax_1 \cup O_1 \cup L_{1\text{out}}) \cup \{\Omega\})$, and $T_2 \subseteq (Ex_2 \cup \{\Omega\}) \times BX_2 \times (p(Ax_2 \cup O_2 \cup L_{2\text{out}}) \cup \{\Omega\})$. When $\bar{M}$ is a WSA, we have $L_1$ (resp. $L_2$) is pairwise disjoint from $E_2, O_2$ (resp. $E_1, O_1$). If $M_1, M_2$ are strongly composable, then $L_1$ (resp. $L_2$) is pairwise disjoint from $L_2, E_2, O_2$ (resp. $L_1, E_1, O_1$): $E_1, E_2$ are pairwise disjoint; $O_1, O_2$ are pairwise disjoint. So, $T_1$ and $T_2$ are pairwise disjoint, i.e. $T_1 \cap T_2 = \emptyset$.

**Proposition 1.** $\bar{M} = M_1 \parallel M_2$ is deterministic iff $M_1, M_2$ are deterministic.

**Proof.** Suppose $(s_1, s_2) \overset{t}{\rightarrow}_{\bar{T}} (s'_1, s''_2) \land (s_1, s_2) \overset{t}{\rightarrow}_{\bar{T}} (s'_1, s''_2)$. By lemma 2, in the case of $t \in T_1$, we must have $s_1 \overset{t}{\rightarrow}_{T_1} s'_1, s_1 \overset{t}{\rightarrow}_{T_1} s''_2$ and $s_2 = s'_2 = s''_2$, but then $s'_1 = s''_2$ by deterministic of $M_1$, so $(s'_1, s'_2) = (s''_1, s''_2)$. The case $t \in T_2$ is similar. We have proved that if $M_1, M_2$ are deterministic then $\bar{M}$ is deterministic.

Next we consider the converse. For $t \in \bar{T}$, suppose $t \in T_1$ we have $s_1 \overset{t}{\rightarrow}_{T_1} s'_1 \land s_1 \overset{t}{\rightarrow}_{T_1} s''_1$. Then $(s_1, s_2) \overset{t}{\rightarrow}_{\bar{T}} (s'_1, s''_2) \land (s_1, s_2) \overset{t}{\rightarrow}_{\bar{T}} (s''_1, s''_2)$. If $\bar{M}$ is deterministic, we must have $s'_1 = s''_1$. Hence $M_1$ is deterministic, likewise $M_2$ is deterministic.

**Theorem 2.** $\bar{M} = M_1 \parallel M_2$ is a deterministic WSA iff

(1) $M_1, M_2$ are deterministic;

(2) $M_1, M_2$ are composable.

**Proof.** If $\bar{M} = M_1 \parallel M_2$ is a deterministic WSA, then (1) is proved by proposition 1. and (2) is proved by theorem 1. Conversely, the condition (2) proves that $\bar{M}$ is a WSA following theorem 1: and the condition (1) proves that $\bar{M}$ is deterministic following proposition 1.

**Proposition 2.** If $M_1, M_2$ are strongly composable, then

(1) If $M_1, M_2$ are deterministic, then $\bar{M} = M_1 \parallel M_2$ is deterministic;

(2) $E_1 \cap E_2 = E_1 \cap L_2 = O_1 \cap O_2 = \emptyset$.
3.3 Composition

(3) $M_1, M_2$ are composable.

**Proof.** By proposition 1, (1) can be proved. From definition 13, we have $E_1 \cap E_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$, and $L_1 \cap E_2 = L_1 \cap L_2 \cap E_1 = L_2 \cap O_1 = \emptyset$ (a). Also, suppose $\hat{M} = M_1 \parallel M_2$, we have $\hat{E} \cap \hat{L} = \emptyset$ (b). By (a)(b) and the pair-wise disjoint property of $E_i, L_i, O_i$, we can prove that (2) holds. (3) holds following corollary 1.

**Lemma 3.** Suppose $M_1, M_2$ are strongly composable, let $\hat{M} = M_1 \parallel M_2$ and $B \in \beta(\hat{E} \cup \hat{L})$, then there exists a unique pair $B_1 \subseteq \beta(E_1 \cup L_1)$ and $B_2 \subseteq \beta(E_2 \cup L_2)$ such that $B_1 \cup B_2 = B$ and $B_1 \cap B_2 = \emptyset$, i.e. $B = B_1 + B_2$. A buffer projection from $B$ to $B_i, i = 1, 2$ is denoted as $proj_i(B) = B_i$.

**Proof.** Let $B_i(x) = \begin{cases} B_i(x) & x \in E_i \cup L_i \\ \emptyset & \text{otherwise} \end{cases}$, we shall show that $(E_1 \cup L_1) \cap (E_2 \cup L_2) = \emptyset$, from this it follows $B_1 \cap B_2 = \emptyset$ so that a message $x \in B_1$ implies $x \notin B_2$. From definition 13, $L_i \cap E_j = \emptyset$ where $i \neq j$. From proposition 2, if $M_1, M_2$ are strongly composable then $E_1 \cap E_2 = L_1 \cap L_2 = \emptyset$, so we can prove that (1) holds as claimed.

**Lemma 4.** Suppose $M_1, M_2$ are strongly composable. Let $\hat{M} = M_1 \parallel M_2$ and $\eta = \hat{s}, \hat{B}, \hat{\epsilon}_V | \hat{V} \in \text{confs}(\hat{M})$, where $\hat{\epsilon}_V | \hat{V}$ assign values to the variables of $\hat{V}$. A configuration projection from $\text{confs}(\hat{M})$ to $\text{confs}(M_i), i = 1, 2$ is denoted as $\text{proj}_i(\eta \in \text{confs}(M_i))$ where $\text{proj}_i(, \hat{s}, \hat{B}, \hat{\epsilon}_V | \hat{V}) = (s_i, \text{proj}_i(\hat{B}), \hat{\epsilon}_V | V_i)$, i.e. $\text{proj}_i(\eta) = \eta_i$.

**Proof.** By lemma 3 we have $\text{proj}_i(\hat{B}) \in \beta(E_i \cup L_i)$, it follows that the configuration of individual automaton $\eta_i$ can be projected from the configuration of composite automaton $\eta \in \text{confs}(\hat{M})$ by $\text{proj}_i(\eta)$.

**Lemma 5.** Suppose that $M_1, M_2$ are strongly composable, let $\hat{M} = M_1 \parallel M_2$, then $\alpha \in \text{trans}(\hat{M}) \Rightarrow \text{proj}_i(\alpha) \in \text{trans}(M_i)$ (1).

**Proof.** Since traces($\hat{M}$) can be retrieved from traces($\hat{M}$), it is sufficient to prove (1) holds. Based on definition 6, we need $\eta \rightarrow^x \eta'$ to represent a step between configurations.

First, when $x = e \in \hat{E}$, we have $\eta \rightarrow^x \eta'$ iff $1) \text{In the case of } x \in E_1 \setminus \hat{L}, \text{proj}_1(\eta) \rightarrow^x \text{proj}_1(\eta')$ and $\text{proj}_2(\eta) = \text{proj}_2(\eta')$. By lemma 3 it follows $x \in \hat{E} \Rightarrow \eta_1 \rightarrow^x$...
3.3 Composition

\( \eta'_1 \land \eta_2 = \eta_2 \). From definition 7 it follows that the conditions \( s_1 = s'_1, B'_1 = B_1 + x \) and \( \epsilon_{V_1} = \epsilon'_{V_1} \) holds for \( \eta_1 \mapsto^x \eta'_1 \). 2) In the case of \( x \in E_2 \setminus \hat{L} \), \( \text{project}_2(\eta) \mapsto^x \text{project}_2(\eta') \) and \( \text{project}_1(\eta) = \text{project}_1(\eta') \). Similar to 1), we can get \( x \in \hat{E} \Rightarrow \eta_2 \mapsto^x \eta'_2 \land \eta_1 = \eta_1 \), and the conditions \( s_2 = s'_2, B'_2 = B_2 + x \), and \( \epsilon_{V_2} = \epsilon'_{V_2} \) hold for \( \eta_2 \mapsto^x \eta'_2 \).

Second, when \( x = t \in \hat{T} \), we have \( \eta \mapsto^x \eta' \) iff 1) In the case of \( x \in E_2 \setminus \hat{T} \), \( \text{project}_2(\eta) \mapsto^x \text{project}_2(\eta') \) and \( \text{project}_1(\eta) = \text{project}_1(\eta') \). Similarly, in the case of \( x \in E_2 \setminus \hat{T} \), \( \text{project}_2(\eta) \mapsto^x \text{project}_2(\eta') \) and \( \text{project}_1(\eta) = \text{project}_1(\eta') \). By lemma 4, it follows \( x \in E_2 \setminus \hat{T} \). As a result, for projection on transitions and messages, we have \( \text{project}_1(X_0, \ldots, X_{n-1}) = \text{project}_1(X_0, \ldots, X_{n-1}) \).

Lemma 6. The composition operator is commutative, i.e. \( M_1 || M_2 = M_2 || M_1 \).

Proof. Given \( M_A = M_1 || M_2 = (I_A, S_A, s_{A0}, S_Af, T_A, \delta_A) \) and \( M_B = M_2 || M_1 = (I_B, S_B, s_{B0}, S_Bf, T_B, \delta_B) \), following the definition of web service automation, the elements of \( M_1 \) are symmetrical with the elements of \( M_2 \). For instance, \( E_A = (E_1 \cup E_2) \setminus \text{com}_{12} = E_B \). Following the commutative nature of the set operators (\( \cup, \cap, \Pi \)), we can prove \( I_A = I_B, s_{A0} = s_{B0}, s_{Af} = s_{Bf}, T_A = T_B \) and \( \delta_A = \delta_B \). Hence \( M_1 || M_2 = M_2 || M_1 \) is proved as claimed.

Lemma 7. Suppose that \( M_1, M_2, M_3 \) are strongly composable, the composition operator is associative, i.e. \( (M_1 || M_2) || M_3 = M_1 || (M_2 || M_3) \).

Proof. Suppose \( M_A = (M_1 || M_2) || M_3 = (I_A, S_A, s_{A0}, S_Af, T_A, \delta_A) \) and \( M_B = M_1 || M_3 = (I_B, S_B, s_{B0}, S_Bf, T_B, \delta_B) \), following the commutative nature of the set operators (\( \cup, \cap, \Pi \)), we can prove \( s_{A0} = s_{B0}, s_{Af} = s_{Bf}, T_A = T_B \) and \( \delta_A = \delta_B \). We can derive \( I_A = I_B \) from \( E_A = E_B, L_A = L_B, \) and \( O_A = O_B \). In the following, we let \( E = E_1 \cup E_2 \cup E_3, L = L_1 \cup L_2 \cup L_3, O = O_1 \cup O_2 \cup O_3, \) and \( X = X_{12} \cup X_{13} \cup X_{23} \), where \( X_{12} = \text{com}_{12}, X_{13} = \text{com}_{13}, \) and \( X_{23} = \text{com}_{23} \). Let \( U = E \cup L \cup O, \) based on set theory, we have \( \hat{X} = U \setminus X \Rightarrow A \setminus X = A \cap \hat{X} \).

First, we need to prove \( E_A = E_B \).
3.4 Compatibility and Anti-patterns

- Let $E_a = E_{1\parallel 2} \cup E_3$ and $X_a = (E_{1\parallel 2} \cup O_3) \cup (O_{1\parallel 2} \cup E_3)$, where $E_{1\parallel 2} = (E_1 \cup E_2) \cap \bar{X}_{12}$ and $O_{1\parallel 2} = (O_1 \cup O_2) \cap \bar{X}_{12}$. We have $E_A = E_a \cap \bar{X}_a$ (1). From (1), we can derive $E_a = E \cap (\bar{X}_{12} \cup E_3)$ (2), $X_a = \bar{X}_{12} \cap (X_{13} \cup X_{23})$ and $\bar{X}_a = X_{12} \cap (\bar{X}_{13} \cup \bar{X}_{23})$ (3). From (2)(3), $E_A = E \cap ((E_3 \cap X_{12}) \cup \bar{X} \cup (E_3 \cap (\bar{X} \cup X_{12})))$ (4) can be derived. By proposition 2, $E_i, E_j$ are pair-wise disjoint, we have $E_3 \cap X_{12} = \emptyset$ (5). From (4)(5) it results $E_A = E \cap (\bar{X} \cup (E_3 \cap \bar{X})) = E \cap \bar{X}$.

- Similarly, let $E_b = E_{2\parallel 3} \cup E_1$ and $X_b = (E_{2\parallel 3} \cup O_1) \cup (O_{2\parallel 3} \cup E_1)$, where $E_{2\parallel 3} = (E_2 \cup E_3) \cap \bar{X}_{23}$ and $O_{2\parallel 3} = (O_2 \cup O_3) \cap \bar{X}_{23}$. We have $E_B = E_b \cap \bar{X}_b$ (6). From (6), we can derive $E_b = E \cap (\bar{X}_{23} \cup E_1)$ (7), $X_b = \bar{X}_{23} \cap (X_{12} \cup X_{13})$ and $\bar{X}_b = X_{23} \cap (\bar{X}_{12} \cup \bar{X}_{13})$ (8). From (7)(8), $E_B = E \cap ((E_1 \cap X_{23}) \cup \bar{X} \cup (E_1 \cap (\bar{X} \cup X_{23})))$ can be derived. Since $E_i, E_j$ are pairwise disjoint, we have $E_1 \cap X_{23} = \emptyset$, such that $E_B = E \cap (\bar{X} \cup (E_1 \cap \bar{X})) = E \cap \bar{X}$. So $E_A = E_B$ is proved.

Second, because $O$ is symmetrical to $E$, we have $O_A = O_B = (O \cap \bar{X})$ as claimed.

Third, we need to prove $L_A = L_B$. By $L_A = (L_{1\parallel 2} \cup L_3) \cup (E_{1\parallel 2} \cap O_3) \cup (O_{1\parallel 2} \cap E_3)$ and $L_{1\parallel 2} = L_1 \cup L_2 \cup X_{1\parallel 2}$, we have $L_A = L \cup X_{1\parallel 2} \cup X_a$ where $X_a = (E_{1\parallel 2} \cap O_3) \cup (O_{1\parallel 2} \cap E_3)$ (9). By (3)(9), $L_A = L \cup X_{1\parallel 2} \cup X_{1\parallel 3} \cup X_{2\parallel 3} = L \cup X$ can be derived. Similarly, by $L_B = (L_{2\parallel 3} \cup L_1) \cup (E_{2\parallel 3} \cap O_1) \cup (O_{2\parallel 3} \cap E_1)$ and $L_{2\parallel 3} = L_2 \cup L_3 \cup X_{2\parallel 3}$, we have $L_B = L \cup X_{2\parallel 3} \cup X_b$ (10), where $X_b = (E_{2\parallel 3} \cap O_1) \cup (O_{2\parallel 3} \cap E_1)$. By (3)(10), $L_A = L \cup X_{1\parallel 2} \cup X_{1\parallel 3} \cup X_{2\parallel 3} = L \cup X$ can be derived. Therefore, $L_A = L_B$ is proved as claimed.

3.4 Compatibility and Anti-patterns

In this section, we define syntax and semantic compatibility. Since syntax compatibility is easy to check from machine interfaces, we focus on checking the semantic compatibility by checking the individual machines against anti-patterns.
3.4 Compatibility and Anti-patterns

3.4.1 Compatibility

After selecting a set of candidate WSAs, we need to check whether the WSAs can interact properly as expected. Syntactic compatibility involves checking machine interfaces for the matching external events. Semantic compatibility means checking the machine behaviours for the absence of pathologies.

Definition 14. Two WSAs \( M_1, M_2 \) are syntactically compatible if \( M_1 \) sends a message \( x \) to \( M_2 \), then there exists \( x = (l, x) \) such that \( l \in O_1 \) and \( x \in E_2 \).

Definition 15. Two WSAs \( M_1, M_2 \) are semantically compatible if a) they are strongly composable, and b) \( \text{trans}(M_1 || M_2) \neq \emptyset \).

Now we discuss when the transition sequence \( \text{trans}(M_1 || M_2) = \emptyset \) holds. This condition holds iff for any \( \hat{\eta} = ((s_0^1, s_0^2), B, (e_{I_0}^1, e_{l_0}^2)) \) there is no transition \( t \in T \) such that \( \hat{\eta} \xrightarrow{t} \hat{\eta}' \), where \((e_{I_0}^1, e_{l_0}^2)\) denotes the initial values of variables. First, suppose the initial configuration is \( \hat{\eta}_0 = ((s_0^1, s_0^2), B, (e_{I_0}^1, e_{l_0}^2)) \), if \( B \in \beta(E) \) then there exists \( \alpha \) such that \( \hat{\eta}_0 \xrightarrow{\alpha} \hat{\eta}_B = ((s_0^1, s_0^2), B + \alpha, (e_{I_0}^1, e_{l_0}^2)) \), where \( \alpha \) consists solely of external input events, i.e. \( \alpha \in \hat{E} \). Second, from definition 11, it shows \((s_1^1, s_2^1) \xrightarrow{t} (s_1^2, s_2^2)\) iff a) if \( t \in T_1 \) then \( s_1^2 = s_2^2 \land s_1^1 \xrightarrow{t} s_1^2 \), or b) if \( t \in T_2 \) then \( s_1^1 = s_2^2 \land s_2^1 \xrightarrow{t} s_2^2 \). Therefore, we have \( \hat{\eta}_B \xrightarrow{t \in T_1 \cup \alpha} \hat{\eta}_B \) or \( \hat{\eta}_B \xrightarrow{t \in T_2} \). Here \( \eta_B = (s_0^1, \text{proj}(B), e_{I_0}^1) \), and \( \text{proj} \) is the buffer projection operator. Without loss of generality suppose \( \hat{\eta} \xrightarrow{t \in T_1} \hat{\eta}' \), we have \( s_0^1 \xrightarrow{t \in T_1} s_1^1, t.m \in B, \) and \( t.g \) is evaluated to true. Conversely, we have \( \neg(\hat{\eta}_B \xrightarrow{t \in T_1}) \) iff for all \( B \) either \( \neg(s_0^1 \xrightarrow{t \in T_1}) \), \( t.m \notin \hat{E} \), or \( t.g \) is evaluated to false.

As a result, condition \( \text{trans}(M_1 || M_2) = \emptyset \) holds iff for \( \forall B \in \beta(\hat{E}) \), there does not exist \( t \in \hat{T} \) such that \( \hat{\eta}_B \xrightarrow{t} \), which indicates that no transition is possible after receiving as many as external inputs.

3.4.2 Anti-patterns

According to definition 15, the condition \( \text{trans}(M_1 || M_2) \neq \emptyset \) can be checked only after constructing the composite WSA. This indicates that a thorough semantic compatibility checking has to be done by exploring the whole state space of the composite.
automaton. However we can speed up the model checking, if some obviously incompatible behaviours can be identified by only checking individual WSAs. We propose anti-patterns for such obviously incompatible behaviours. As a complementary approach to post-checking, we provide warnings so that the problematic WSA can be either re-selected or modified in the earliest stages. Furthermore, since a WSA's local ordering (definition 18) only needs to be computed once, the local ordering can be re-used for pre-checking the compatibility with other machines. After pre-checking, post-checking can be applied to thoroughly check the composite automaton for safety and liveness properties. Note that our anti-patterns are mainly used as design guidelines. Models with anti-patterns have potential to cause problems.

Referring to the event occurrences and message instances discussed in section 3.2, the anti-patterns discuss the temporal relations over event occurrences in traces of individual machines. We follow the standard definitions of strict partial order and mutually exclusive relation in set theory (e.g. [9]).

**Definition 16.** In a machine $M$, suppose $e, e' \in MsgM$, the strict partial order over event occurrences $o_1 \prec o_2$ where $\lambda(o_1) = e, \lambda(o_2) = e'$ indicates that $o_1$ happens before $o_2$ in a trace of $M$.

**Corollary 2.** Suppose there is a message $m$ from machine $M_1$ to $M_2$, if we have a message instance $om = (|om, ?om)$ where $\lambda(|om) =!m$ and $\lambda(?om) =?m$, then the strict partial order over an event occurrence pair $!om \prec ?om$ enforces that a message instance must be received after it is sent.

**Definition 17.** In machine $M$, suppose $e, e' \in MsgM$, the mutually exclusive relation on event occurrences $o_1 \not\prec o_2$ where $\lambda(o_1) = e, \lambda(o_2) = e'$ indicates that $o_1$ is branch-conflict with $o_2$ in a configuration sequence of $M$. Intuitively, two branch-conflict event occurrences cannot happen in the same trace.

**Definition 18.** The local ordering on a single machine $M_i$ is a structure $l_i = (C_i, \prec_i, \not\prec_i)$, where $C_i$ is the event occurrence set of $E_i \cup O_i$, $\prec_i$ and $\not\prec_i$ are the strict partial order and the mutual exclusion relations on $C_i$, respectively.
3.4 Compatibility and Anti-patterns

Definition 19. For multiple machines \( \{M_1, \ldots, M_n\} \), we define message orderings to be the structure \( \prec_X = \bigcup_{\lambda(\text{om}), \lambda(\text{om}) \in X} \{\text{om} < \text{om'}\} \), where \( \text{om} = (\text{lo}m, \text{ro}m) \), \( X \subseteq \bigcup_{1 \leq i,j \leq n} \text{com}_{ij} \) is the set of messages sent between machines \( \{M_1, \ldots, M_n\} \), and \( \prec \) is the strict partial order on event occurrence pairs.

Definition 20. The causal ordering for a set of machines \( \{M_1, \ldots, M_n\} \) is the structure \( \prec_c = (\bigcup_{1 \leq i \leq n} \text{li}) \cup \prec_X \), which describes the transitive closure of the set of local orderings and message orderings.

Definition 21. A machine \( M \) is said to be blocking iff there exists a state \( s \notin S_f \) and a trace \( \alpha \in \text{traces}(M) \) such that \( s_0 \xrightarrow{\alpha} s \) and \( \neg(s \xrightarrow{\alpha}) \) for \( \forall t \in T \). Referring to definition 9, let \( S_d \) be the set of all blocking states, \( S_d = \{s \in S_{\text{Reach}}(M) \setminus S_f : \forall t \in T. \neg(s \xrightarrow{\alpha})\} \), so \( M \) is blocking iff \( S_d \neq \emptyset \).

In state machine diagrams, an initial state is pointed out by a start arrow with a filled black circle, and a final state is shaded. For the anti-patterns, we suppose for two messages \( x = (!x, ?x) \) and \( y = (!y, ?y) \) sent between machines \( M_1 \) and \( M_2 \), there exist message instances \( o_x = (!o_x, ?o_x) \) and \( o_y = (!o_y, ?o_y) \) of \( x \) and \( y \), respectively, such that \( \lambda(o_x) = !x \) and \( \lambda(o_y) = !y \). Message Sequence Charts (MSCs) are used to show the anti-pattern scenarios. In the examples, we introduce index \( k \) to identify a message instance of a message in a trace, denoted as \( oe[k] = (!oe[k], ?oe[k]) \). The index \( k \) can be omitted when there is only one message instance.

Anti-Pattern 1. Suppose \( !x \in O_1, ?x \in E_2 \) and \( !y \in O_2, ?y \in E_1 \), \( M_1 \parallel M_2 \) has unspecified reception if \( ?o_y <_1 !o_x \) and \( ?o_x <_2 !o_y \) (1).

![Figure 3.2: Unspecified reception](image-url)
3.4 Compatibility and Anti-patterns

Figure 3.2 shows the corresponding MSC on the left and the causal ordering on the right. Machine $M_1$ sends message instance $ox$ to machine $M_2$, and $M_2$ sends message instance $oy$ to $M_1$. $M_1$ cannot send $ox$ until it receives $oy$, while $M_2$ cannot send $oy$ until it receives $ox$ (1). Based on message ordering and (1), the causal ordering $\prec_c$ consists of $lox < ?ox < ?oy < ?oy$ and $loy < ?oy < !ox < ?ox$. This conflict indicates that $M_1, M_2$ wait for message instances from each other but never get them. Hence, $M_1 \parallel M_2$ has an unspecified reception, where the blocking state sets of both machines are not empty.

Figure 3.3: An example of anti-pattern 1

Figure 3.3 illustrates an example. $\text{traces}(M_B)$ includes $(?oe_1[1],!oe_1[2],!oe_2)$ and $(?oe_1[1],!oe_2,?oe_1[2])$. $\text{traces}(M_C)$ includes $(!oe_1[1],?oe_2,!oe_1[2]), (!oe_1[1],?oe_2,?oe_3), (!oe_1[1]),$ and $(?oe_3)$. The partial order $?oe_1[2] <_B !oe_2$ and $?oe_2 <_C !oe_1[2]$ can be obtained from one of the traces of $M_B$ and the trace of $M_C$, so according to anti-pattern1, $M_B \parallel M_C$ will deadlock when $?oe_1[2]$ happens before $!oe_2$ in $M_B$ and $?oe_2$ happens before $?oe_1[2]$ in $M_C$. The blocking states are $S^B_d = \{B_2\}$ and $S^C_d = \{C_2\}$.

**Anti-Pattern 2.** Suppose $!x \in O_1, ?x \in E_2$ and $!y \in O_1, ?y \in E_2, M_1 \parallel M_2$ has non-local branching choice if $!ox \neq !oy$ (2).

Figure 3.4: Non-local branching 1
3.4 Compatibility and Anti-patterns

Figure 3.4 shows the MSC and causal ordering. $M_1$ sends message instances $ox, oy$ to $M_2$. In $M_1$, a send event occurrence of $ox$ is in branch-conflict with a send occurrence of $oy$ (2). Two cases may exist in $M_2$:

1) If $?ox <_2 ?oy$, $M_2$ waits for both message instances $ox$ and $oy$ from $M_1$, but $M_1$ cannot send these message instances in the same run due to (2). We have $\prec: ox \land oy \prec ox \land oy \prec oy \land ox \prec oy$. By $\lor x \mid oy$ and message ordering, we have $ox \Rightarrow \neg(\lor y) \Rightarrow \neg(\lor y)$ and $oy \Rightarrow \neg(\lor x) \Rightarrow \neg(\lor x)$, so $?ox <_2 ?oy$ does not hold.

2) If $?ox ?_2 ?oy$, we have causal ordering $\prec: ox \land oy \lor x \prec ox \land oy \prec oy \land ox \prec oy$. If $\lor x$ (resp. $\lor y$) happens in $M_1$ and $\lor y$ (resp. $\lor x$) happens in $M_2$, then $M_2$ will wait for message instance $oy$ (resp. $ox$) forever due to (2). The blocking state set of $M_2$ is not empty.

Figure 3.5 shows an example, in which we have $!om_2 \mid !om_3$. In case 1, we have $?om_2 <_B ?om_3$, so $M_A \parallel M_B$ has non-local branching choice with $S^B_a = \{B_1\}$ or $S^B_a = \{B_5\}$. In case 2, we have $?om_2 ?_B ?om_3$, so $M_A \parallel M_B$ has non-local choice with $S^B_a = \{B_2\}$ or $S^B_a = \{B_4\}$.

Figure 3.5: An example of anti-patterns 2 and 3

**Anti-Pattern 3.** Suppose $!x \in O_1, ?x \in E_2$ and $!y \in O_2, ?y \in E_1$. $M_1 \parallel M_2$ has non-local branching choice if $\lor x \mid ?y$ and $\lor z \mid \lor y$ (3).

Figure 3.6 shows the MSC and causal ordering. $M_1$ sends message instance $ox$ to $M_2$, and $M_2$ sends message instance $oy$ to $M_1$. The causal ordering is $\prec: ox <_C ?ox$. 


Figure 3.6: Non-local branching 2

\( \text{Figure 3.6: Non-local branching 2} \)

\[ \begin{array}{c|c|c}
M1 & \text{M2} & \text{M3} \\
\hline
!ox & !oy & !oz \\
?oy & ?ox & ?oz \\
\end{array} \]

\( !oy < ?oy, !ox ? oy, \text{and } ?ox ! oy. \) If \( ?oy \) happens in \( M_1 \) and \( ?ox \) happens in \( M_2 \), machine \( M_1 \) (resp. \( M_2 \)) will wait for message instance \( oy \) (resp. \( ox \)) forever due to (3). The blocking state sets of both machines are not empty.

In Figure 3.5, \( !om_3 \downarrow A \) and \( ?om_3 \uparrow B !om_4 \) hold. When \( ?om_4 \) happens in \( M_A \) and \( ?om_3 \) happens in \( M_B \), \( M_A \parallel M_B \) has non-local choice and each machine has blocking states \( S_d^A = \{ A_4 \} \) and \( S_d^B = \{ B_2 \} \). Similarly, \( !om_2 \downarrow A \) and \( ?om_2 \uparrow A \) will cause non-local choice. In this example, \( om_2 \downarrow A \) will also cause non-local choice with the type of anti-pattern 2.

### 3.5 Summary

The formal static semantics and dynamic semantics of Web Service Automaton are given in this chapter. The message-passing mechanism is used for machine communications, so that the internal interactions of BPEL activities and the external interactions of BPEL processes can be model in an uniform way. We believe the web service automaton is more suitable to model the BPEL features, because of our consideration of multiple-input events and data, and machine communication. Also, anti-patterns are identified for web service interactions. In the next chapter, according to the proposed web service automaton, we will analyze the BPEL features in details and show how to model them in web service automata.
Chapter 4

BPEL Analysis and modelling

Based on the web service automaton presented in the previous chapter, we will analysed the BPEL features and demonstrate how to model these features in web service automata. BPEL consists of two categories of activities: basic and structured activities. Basic activities are atomic actions, including receive, reply, assign, invoke, throw, terminate, empty, and wait. As with programming languages, the structured activities impose control flow dependency constraints on the executions of either the basic or structured activities within them. A structured activity can contain an arbitrary depth of sub-activities. The structured activities include pick, switch, while, sequence, flow, scope, eventHandlers, faultHandlers, compensationHandler. For data handling, BPEL uses the blackboard approach, where a set of variables is shared by all activities within the same scope.

We will analyze the BPEL control dependencies and data dependencies in separation, and then we will illustrate how to capture these dependencies in web service automata. Thereafter, we will show how to model BPEL activities in web service automata. The BPEL scope-based error handling features are also modeled, which include scopes, eventHandlers, faultHandlers, and compensationHandlers. These features are important because they affect much of the behaviour of BPEL models that deal with exceptional cases. We do not model the BPEL correlation sets and time related features.
We use \textit{machine} as shorthand for a web service automaton, and call the machine associated with BPEL x activity as x machine. In state machine diagrams, an initial state is pointed to by an arrow started with a filled black circle, and a final state is shaded.

\section*{4.1 BPEL Control Flows}

We will analyse the BPEL control dependencies and illustrate how to capture the behaviour semantics of BPEL basic activities and structured activities in \textit{WSA}. The error handling in business processes is important to support long running transactions. In BPEL, the \textit{scope}, \textit{compensate}, \textit{faultHandlers}, and \textit{compensationHandlers} activities are used for error handling. However, existing automata based approaches omit the modelling and verification of error handling related BPEL activities. We aim to fill such gaps by capturing the semantics of both normal BPEL activities and error-handling related BPEL activities in \textit{WSA}.

\subsection*{4.1.1 Hierarchical BPEL activities}

Since a WSA has no hierarchy, we simulate the hierarchical BPEL activities by the parent and child relationships between machines. If machine $M_A$ sends a start message to machine $M_B$, then $M_A$ is the parent machine of $M_B$, and $M_B$ is the child machine of $M_A$. A child machine can optionally send a done message to its parent machine when reaching one of its final states. Each machine has 0..1 parent machine, and 0..* child machines. Since the BPEL basic activity is atomic and a BPEL structured activity is hierarchical, the machine for a BPEL basic activity has no child, and the machine for a BPEL structured activity has 0..* children.

Fig 4.1 shows parent and child relationships between the machines of the loanapproval example that is described in detail in section 6.1.1. A machine without an incoming dark arrow (start message) is the \textit{process} machine. The machine without an incoming hollow arrow (done message) is a \textit{basic} machine. The \textit{process} is the parent of
flow, and the flow is the parent of receive\textit{linkWrapper}, which in turn is the parent of receive.

![Diagram of machine relationships]  

Figure 4.1: An example of machines with parent and child relationships

A machine without a child machine is called \textit{simple machine}, otherwise a machine with any child machine is called \textit{compound machine}. Let $A_B, A_S$ be the BPEL basic and structured activities, respectively. $A_B, A_S$ are associated with simple or compound machines according to the following rules:

- For $A_B$ without source or target link, it is associated with a simple machine.
- For $A_B$ with source or target links, it is associated with a simple core machine and a compound linkWrapper machine (see section 4.1.3).
- For $A_S$ without source or target link, it is associated with a compound machine.
- For $A_S$ with source or target links, it is associated with a compound core machine and a compound linkWrapper machine.

A simple machine has two kinds of FIFO queues: 1) an adminQueue for start and stop messages; and 2) a set of messageQueues where each messageQueue corresponds to a exchange message type. Each compound machine has four kinds of FIFO queues: 1) a adminQueue for start and stop messages; 2) a faultQueue for fault messages propagated from its child machines; 3) a set of doneQueues where each doneQueue corresponds to
4.1 BPEL Control Flows

a child machine; and 4) a set of message Queues where each message Queue corresponds to a exchange message type.

**Common compound machine layout:** With consideration of faults and interruptions, the compound machines for BPEL structured activities have a common layout, shown in Fig 4.2. A machine can have one or more child machines. Each compound machine has a *stopStatus* as a local variable. Suppose *M* is a compound machine, we can derive three scenarios from this common layout: 1) when *M* receives a fault from its children and no stop message arrives, it forwards the fault to its parent (t4.3), and the 2) scenario is followed; 2) when *M* is interrupted by receiving a stop message, it propagates the message to its children (t4.0) and updates the *stopStatus* to true. Upon receiving the child machines’ done messages, if the current *stopStatus* is true, then the machine enters an abnormal final state (t4.1); 3) when *M* receives the children’s done messages and no fault or stop message arrives, it ends normally (t4.2). In a fault message *fault(f)*, the message parameter *f* contains *faultName* and *faultVariable* elements.

<table>
<thead>
<tr>
<th>T1: s1-&gt;s1, parent?stop(tm)/stopStatus:=true, child1?stop(tm)/... childn?stop(tm)</th>
<th>t2: s1-&gt;s1, child1?done...&amp;...(childn?done/parent?stop(tm))</th>
<th>t3: s1-&gt;s1, (child1?fault(f)...&amp;...(childn?fault(f)) &amp; ~parent?stop(parent?fault(f))</th>
</tr>
</thead>
</table>

Figure 4.2: The common layout of compound machines

It can be observed that the priority of the incoming events from high to low are: stop, fault, and done messages. In transition t4.2, we do not use the predicate *stopStatus! = true* to guard the transition. In the case that both done and stop messages have arrived, the *stopStatus* is updated to true only when the stop message is consumed, so the predicate *stopStatus! = true* has no impact on the selection of consuming done or stop message. Without using the priority constraints on events, the machine only consumes one of them randomly. So, we introduce multiple-input events to the transition.

50
4.1.2 Concurrency, Fault Propagation, and Interruption

In this subsection, we show that a machine’s input events with logical AND, OR, NOT can capture various BPEL features.

![Figure 4.3: Multiple-input events and common machine structure](image)

**Concurrency:** BPEL flow, scope, and eventHandler activities allow the enclosed activities to perform concurrently. We use flow activity as an illustration here. When the flow enters, all the enclosed activities start. The flow ends when all the enclosed activities end. We model this by two transitions, shown in (1) Fig 4.3. On the left of (1), the flow machine starts all its children as a transition action, so that all child machines will start at the same time. On the right of (1), a logical-AND operator is added to the transition input events, so that the flow machine will not end until all its children end by sending done messages.

**Fault propagation:** When a structured machine receives a fault message from its children, it forwards the fault message to its parent. Suppose the structured activity encloses more than one activity. The fault is propagated as long as one of the enclosed activities raises a fault. We model this by adding a logical-OR operator to the transition input events, shown in (2) of Fig 4.3. Instead of using a queue for each fault, we use one FIFO queue to store all fault messages, so the fault message sent from the activity machine to its parent depends on which child’s fault comes first.

**Interruption:** BPEL has two kinds of Interruptions. First, when a termination message is thrown when a terminate activity is reached, the process machine ends abnormally, and a stop message is propagated downstream from the process machine. Second, when a fault is thrown by a throw activity or an invoke activity, the fault will be propagated upstream until it can be caught by a scope or process activity that
has the faultHandlers to handle this fault. The scope or process activity will stop its normal activities before enabling the faultHandlers. The stop message is propagated downstream from the scope or process machine. When a structured activity is stopped, all its children need to be stopped first. This is modelled by propagating a stop message downstream. The priority of a stop message is captured by adding logical-AND together with logical-NOT to transition input events. A stop message has higher priority than a fault message, which in turn has higher priority than a normal message. In (3) of Fig 4.3, transition $t_{i,0}$ is triggered when a stop message arrives. The transition $t_{i,1}$ will be triggered when it receives a fault message from its child, and only when no stop message arrives. It indicates that a fault will not be propagated when the machine is asked to stop. The transition $t_{i,2}$ indicates that a fault or interruption message has higher priority than a normal incoming message.

4.1.3 Synchronisation of Activities and Dead-Path-Elimination

A set of links can be declared in the flow construct to express the synchronisation dependencies between activities within a flow. A link is a Boolean variable, and each link is associated with a pair of source activity and target activity. For instance, if $M_A$ and $M_B$ are source and target activities of a link $l_1$, respectively, then $l_1$ is $M_A$'s outgoing link with source tag, and $M_B$'s incoming link with target tag.

The synchronisation between source and target activities is realised by setting and getting the link value. The source activity sets the link to be true or false, and the target activity gets the link value. The target activity can start when 1) all the incoming links' values are defined by the source activities, and 2) its associated join-condition is satisfied, which is a user-defined logical constraint on link values. The default logical constraint is OR. If the join-condition is false, the target activity will not be executed and this effect will be propagated downstream in the flow model. This is called Dead-Path-Elimination in BPEL. We capture the dead-path-elimination feature by updating the related links to false, and sending the link-related data exchange messages to the target activity machines.
The target tag and source tag are standard elements of BPEL constructs, indicating every BPEL activity may or may not have incoming links and outgoing links. It would be too complicated to consider handling for each activity, so we use a supporting linkWrapper machine to handle links. When an activity has incoming or outgoing links, it will associate with a linkWrapper machine and a core machine. The linkWrapper will be the core machine's parent. When an activity has no link, it is only associated with a core machine. This separation simplifies the structure of a machine, and allows BPEL activities to share a common machine structure for link handling. Fig 4.1 shows the linkWrapper machine structures, which covers the cases when an activity 1) has source links but no target link, 2) has target links but no source link, 3) has target links and source links.

Suppose \( M_B \) is the core machine and \( M_A \) is the linkWrapper machine for an activity. Several scenarios can be derived from the machine structure. For case 1), when an activity has source links but no target link, two normal scenarios follow the paths \( \langle t_{0,1}, t_{3,1} \rangle, \langle t_{0,1}, t_{3,1}, \ldots, t_{3,n} \rangle \). The machine \( M_A \) starts by receiving a start message from its parent, and it starts the child machine \( M_B \) \( (t_{0,1}) \). After \( M_B \) has finished, \( M_A \)
send a done message to its parent. If the activity has one source link $l_1$, $M_A$ sets the link $l_1$ to true and sends the link exchange message $msg(s_{l1})$ to the machine of the link’s target activity. If the activity has more than one source link $s_{l1}, \ldots, s_{ln}$, for the transitionCondition of link $s_{l_x}$ is evaluated to true, only one link $s_{l_x}$ is set to true and the rest links are set to false. Thereafter, $M_A$ sends the link messages $msg(s_{l1}); \ldots; msg(s_{ln})$ to the machines of the links’ target activities. For case 2), when an activity has target links but no source link, a normal scenario follows the path $(t_{0.2}, t_{2.1}, t_{3.1})$. $M_A$ receives link messages $msg(t_{l1}), \ldots, msg(t_{lm})$ from the machines of the links’ source activities. $M_A$ waits for all the link messages to arrive and check whether the links satisfy the joinCondition. For case 3), when an activity has target links and source links, machine $M_A$ handles the target links in the same way as the case 2), and the machine handles the source links the same as the case 1).

When all the link messages have arrived and the links do not satisfy the joinCondition, a $joinFailure$ occurs and $M_A$ ends abnormally by updating all the source links to false and sending the link messages to the machines of links’ target activities $(t_{2.0})$. Alternatively, when $M_A$ receives a fault from $M_B$ $(t_{3.02})$, $M_A$ propagates the fault to its parent. As long as $M_A$ receives a stop message from its parent $(t_{3.00})$, $M_A$ propagates the stop to $M_B$. After having received $M_B$’s done message, $M_A$ ends abnormally by setting all source links to false and sending the link messages $msg(s_{l1}); \ldots; msg(s_{ln})$ to the machines of the links’ target activities $(t_{3.01})$.

### 4.2 BPEL Data Flows

The current automata based approaches omit the BPEL data dependencies. In order to address this shortcoming, we demonstrate how to model BPEL data dependencies in our proposed web service automata.

When the flow or scope’s variableAccessSerializable attribute is set to yes, the flow or scope provides concurrency control in governing access to shared variables between event handling threads inside a scope. For those shared variables that are not protected
as such, we assume that their read and write order is not determined at the design time. Since general handling of shared variables protected by variableAccessSerializable attribute is difficult and it will not be covered in our transformation.

4.2.1 Motivating Example

In BPEL, variables and links may affect the control flow, variables may appear in expressions on the conditions in switch and while, and may also be used in the conditions to fire particular links in the source element. There are two types of variables explicitly declared in a BPEL model: BPEL variables and flow links. BPEL variables are declared in the variables construct of either process or scope activities. Flow links are Boolean variables declared in the link tags of the flow activity. The output link of an activity is defined as true if the associated activity completes normally, otherwise the link is defined false. BPEL variables and links can be used and defined by the process or scope enclosed activities, and the flow enclosed activities, respectively.

![Diagram of unreachable and deadlock activities](image)

Figure 4.5: Unreachable and deadlock activities

Fig. 4.5 shows the importance of analysing BPEL data flow. The boxes, the solid arcs, and the dashed arcs denote BPEL activities, control flow, and data flow, respectively. The process encloses a flow, which in turn includes pick, switch, and E running concurrently. The example contains unreachable and deadlock activities. Firstly, B never instantiates variable var1, due to the interaction between data flow and control flow. In the pick, A and B are mutually exclusive in control flow, but the output of A is the input of B in data flow, so B cannot instantiates var1. E is unreachable due to the faulty design of links. In the switch, C and D are mutually exclusive in control
flow, so that link1 and link2 cannot be both true to satisfy the AND-join condition. Therefore, E can never be executed. Secondly, there is a deadlock between switch and E, which is caused by the cyclic data flow between them. On one hand, E waits for both link1 and link2 to be true but this condition can never be satisfied. On the other hand, the switch waits for its input var2 to be defined but var2 cannot be defined by E because of the falsity of either link1 or link2. This illustrates the necessity to verify both control and data dependencies of BPEL models.

4.2.2 BPEL Data Dependencies

In a system, a BPEL process P is seen as a component, and the partnerLinks declared in P correspond to the components interacting with P. In the loan-approval example that will be used later in this paper, the BPEL process loanapproval includes three partnerLinks, so the system has four components: loanapproval, customer, assessor, and approver. From the testing point of view, when more than one BPEL model is considered, the system boundary needs to be included. The components within the system boundary are called SUT (service under test), and a component outside the system boundary is called tester or environment. In the following, for a message $msg(x) \in E_{M_1} \cup O_M$, $msg$ and $x$ denote the message name and input/output parameter, respectively.

Let $\{M_m..M_n\}$ be the set of machines selected as the SUT, a message $msg(v)$ sent from machine $M_1$ to machine $M_2$, and a transition $t$ associated with variable $x$, we have:

- $t$ is annotated with $df(x)$ if a) $x$ is defined in an assignment action of $t$, i.e. 
  \{x \in def(exp)|exp \in t.a\}; or b) $x = v$ is the input parameter of $M_2$ where $M_1$ and $M_2$ are tester and SUT respectively, i.e. \{x \in def(exp)|t \in T_{M_2} \land exp \in t.e x\}, $M_2 \in \{M_m..M_n\}$; $M_1 \notin \{M_m..M_n\}$.

- $t$ is annotated with $us(x)$ if a) $x$ is used in an assignment action or guard of $t$, i.e. \{x \in uses(exp)|exp \in t.a\} or \{x \in uses(exp)|exp \in t.g\}; or b) $x = v$ is
the output parameter of $M_1$ where $M_1$ and $M_2$ are tester and SUT respectively, i.e. \{x \in \text{cases}(\text{exp}) | t \in T_{M_1} \land \text{exp} \in t.a \cap O_{M_1}, M_2 \in \{M_m \ldots M_n\}, M_1 \notin \{M_m \ldots M_n\}\}.

- Let $P_1, P_2$ be two BPEL processes. $t$ is annotated with $\text{idf}(x)$ if $x = v$ is the input parameter of $M_2$ where $M_1, M_2$ belong to different BPEL processes but both are SUT, i.e. \{x \in \text{def}(\text{exp}) | t \in T_{M_2} \land \text{exp} \in t.e \ldots x\}, M_1, M_2 \in \{M_m \ldots M_n\}\}.

- Let $P_1, P_2$ be two BPEL processes. $t$ is annotated with $\text{ius}(x)$ if $x = v$ is the output parameter of $M_1$ where $M_1, M_2$ belong to different BPEL processes but both are SUT, i.e. \{x \in \text{cases}(\text{exp}) | t \in T_{M_1} \land \text{exp} \in t.a \cap O_{M_1}, M_1, M_2 \in \{M_m \ldots M_n\}\}.

The $i$ in $\text{idf}(x), \text{ius}(x)$ means internal. For simplicity, a transition $t$ is called with $\text{def}-x$ if $t$ can be either annotated with $\text{df}(x)$ or $\text{idf}(x)$, while $t$ is called with $\text{use}-x$ if $t$ can be either annotated with $\text{us}(x)$ or $\text{ius}(x)$.

**Definition 22.** A variable $x$ is globally defined and used if there exist transitions $t_1 \in T_{M_1}, t_2 \in T_{M_2}$, where $t_1, t_2$ are with $\text{def}-x$ and $\text{use}-x$, respectively. The variable $x$ is locally defined and used if: 1) there exist transitions $t_1, t_2 \in T_{M_1}$, where $t_1, t_2$ are with $\text{def}-x$ and $\text{use}-x$ respectively; or 2) there exits a transition $t_1 \in T_{M_1}$ where $t_1$ is with $\text{def}-x$ and $\text{use}-x$.

According to the BPEL 1.1 [10], in a WSA the case 1) and 2) will not exist for those variables explicitly declared in BPEL models (i.e. BPEL variables and flow links). Therefore, we only consider globally defined and used variables, such that a machine can either have a transition with $\text{def}-x$ or have a transition with $\text{use}-x$ but not both. As a result, the $\text{def}-x$ (resp. $\text{use}-x$) annotation of a machine $M$ can be retrieved from the $\text{def}-x$(resp. $\text{use}-x$) of a transition $t \in T_M$. A $\text{exdu-pair}$ of $x$ is a transition pair $(t_i, t_j)$ where $t_i$ is with $\text{def}-x$ and $t_j$ is with $\text{use}-x$. The pair of machines with $\text{def}-x$ and $\text{use}-x$ is called $\text{machine-exdu-pairs}$.
4.2.3 Data Exchange Models

An internal data exchange model is used for a single BPEL process to specify the relation between inputs and outputs of BPEL activities. An external data exchange model is used to capture how messages are transferred from one BPEL process to other BPEL processes. When a single BPEL process is selected as the SUT, an internal data exchange model is enough to capture the BPEL data semantics. When multiple BPEL processes are selected as the SUT, a global data exchange model which is the union of the internal and external data exchange models is required to capture the BPEL data semantics.

A. Internal Data Exchange Model

In this section, we identify different types of data dependencies of BPEL activities, and discuss how to capture these data dependencies in WSAs.

Rule 1. In a BPEL process $P$, let $x$ be a BPEL variable or a flow link explicitly declared in $P$, and $M_i$ be the WSA associated with BPEL activity $B_i$. BPEL activities can be categorised into four types.

1) When $B_i$ receives $msg(x)$ from an external BPEL process, given that the partner which sends $msg(x)$ is a tester and $B_i$ is SUT, $M_i$ is with def-$x$. The BPEL activities belonging to this type include: receive activity, invoke activity with $x$ as outputVariable, pick activity, and eventHandler activity.

2) When $B_i$ uses $x$ in a predicate or assignment, it reads the value of $x$ and $M_i$ is with use-$x$. The following BPEL activities belong to this type: assign activity with $x$ on the right of assignment expressions, while activity and switch, and an activity with $x$ as a targetLink.

3) When $B_i$ sends a message $msg(x)$ to an external BPEL process, given that the partner which receives $msg(x)$ is a tester and $B_i$ is SUT, $M_i$ is with use-$x$. The
4.2 BPEL Data Flows

BPEL *invoke* activity with *x* as the inputVariable, and *reply* activity belong to this type.

4) When *Bi* defines *x* in an assignment, it writes a value to *x* and *Mi* is with def-*x*.

Two BPEL activities belong to this type: *assign* activity with *x* on the left of assignment expressions and an activity with *x* as a sourceLink.

**Rule 2.** In a BPEL process *P*, the data can only exchange between two machines *M1, M2 ∈ P* if one of the following conditions is satisfied:

1) *M1, M2* have a same parent machine, i.e. they are same-level machines;

2) *M1, M2* are parent and child, or vice-versa.

For simplicity, condition 1) will be checked first, and condition 2) will be checked when 1) is false.

The rationale behind rule 2 is illustrated by an example in Fig 4.6. The BPEL model fragment is shown in Fig 4.6(a), where the solid lines denote the node hierarchy of BPEL activities. It has a *sequence* activity that encloses *flow, while,* and *switch* activities. The *flow* activity encloses *A* and *B* activities. The *while* activity encloses *C* activity.

The *switch* activity encloses *C* and *D* activities. In the example, *MA* is used to denote the machine for activity *A*, and *Mf, Mw, Ms* are short for *Mflow, Mwhile, Mswitch*. By analysing model(a) according to rule 1, suppose we have *MA, MC* with def-*x*, *Mw, Ms* with use-*x*, *MB* with def-*y*, and *MD, ME* with use-*y*.

To capture data semantics of the BPEL model in WSAs, Fig 4.6 (b) and (c) show two ways of modelling data exchanges. The dashed and solid lines denote the machine data flows and the control flows, respectively. In (b), for an arbitrary variable *v*, the machine with def-*v* will send message *msg(v)* to the machine with use-*v* directly. Based on the data exchange model (b), three problems may exist:

1) Sending intermediate data values.

2) Sending data to unreachable machines.
4.2 BPEL Data Flows

Problem 1) exists in the example; machine $M_w$ receives $msg(x)$ from $M_A$ and uses $x$ in its predicate $pred$. If $pred$ is true, it starts the child machine $M_C$. $M_C$ re-defines $x$. On one hand, $M_C$ sends $msg(x)$ back to $M_w$ for re-evaluating $pred$. On the other hand, $M_C$ sends $msg(x)$ to $M_s$. The while loop continues until $pred$ becomes false. Since $M_C$ is in a while loop, everytime $M_C$ is executed, it will send message $msg(x)$ to $M_s$. If the while loop iterates $n$ times, then $M_s$ will receive $n - 1$ times of $msg(x)$ with intermediate values of $x$. Nevertheless, $M_s$ only needs the value of $x$ in the final loop, so the decision of sending $msg(x)$ should be made by $M_w$.

Problem 2) also exists in the example. $M_s$ has two child machines $M_D, M_E$, where in a given time only one of the child machines can be executed but not both. Therefore, either $M_D$ or $M_E$ needs to receive $msg(y)$. However, since $M_B$ cannot decide the choice of $M_D$ or $M_E$, $M_B$ will send $msg(y)$ to $M_D$ and $msg(y)$ to $M_E$. Suppose $M_E$ is not chosen to execute, $M_B$ will send $msg(y)$ to an unreachable machine. Therefore, $M_B$ should send $msg(y)$ to $M_s$ which can decide which child machine should receive the data. Furthermore, problem 2) may exist for a BPEL activity $B$ with flow links, which will be associated a linkWrapper machine $M_{Bwp}$ and a core machine $M_B$. Since $M_{Bwp}$ is the parent of $M_B$, $M_{Bwp}$ decides whether $M_B$ can be started. $M_B$ can be started only when the predicate for the targetLinks is evaluated to be true in $M_{Bwp}$. So if $M_B$ requires a data $v$, the message $msg(v)$ should be sent to $M_{Bwp}$ which decides whether to forward it to $M_B$. 
The above two problems can be solved by adding the constraints of rule 2, shown in Fig 4.6 (c). For the first problem, \( M_C \) sends \( msg(x) \) to its parent \( M_w \), and \( M_u \) forwards \( msg(x) \) to the same-level machine \( M_s \). For the second problem, \( M_B \) sends \( msg(y) \) to its parent \( M_f \), and \( M_f \) forwards \( msg(y) \) to the same-level machine \( M_s \), which in turn forwards \( msg(y) \) to one of \( M_D, M_E \). Comparing (b) and (c), the (c) approach is clearer and more precise even though it requires additional message transfers.

According to rule 2, Fig 4.7 below shows an algorithm to generate a data exchange path (machine sequence) for a machine-exdu-pair \( (d_{node}, u_{node}) \). The idea is to find a common ancestor node \( cp \). The message \( msg(x) \) is sent upstream from \( d_{node} \) to a child node \( M_{C1} \) of \( cp \), while \( msg(x) \) is received downstream from a child node \( M_{C2} \) of \( cp \) to \( u_{node} \). Finally \( M_{C1} \) sends \( msg(x) \) to \( M_{C2} \). The worst-case time complexity of the algorithm is \( O(n^2) \).

```
1 foo (d_node, u_node): path {
2     IF (d_node == u_node) RAISE ERROR!
3     d_path.add(d_node);
4     u_path.add(u_node);
5     d_cur = d_node;
6     u_cur = u_node;
7     WHILE ((d_par != d_cur.parent()) != NULL OR (u_par != u_cur.parent()) != NULL) {
8         IF (d_par != NULL) {
9             IF (d_par IN uyath) {
10                IF (dyar == u_path[0]) RETURN d_path.add(d_par);
11                ELSE {
12                    u_path = u_path.getNodesBefore(d_par);
13                    RETURN d_path.concat(reverse(u_path));
14                }
15            }
16            ELSE {
17                d_path.add(d_par);
18                d_cur = d_par;
19            }
20        }
21        IF (u_par != NULL) {
22            IF (u_par IN d_path) {
23                IF (u_par == d_path[0]) RETURN u_path.insert(u_par,0);
24                ELSE {
25                    u_path = u_path.getNodesBefore(u_par);
26                    RETURN d_path.concat(reverse(u_path));
27                }
28            }
29        }
30        ELSE {
31            u_path.add(u_par);
32        }
33        RETURN NULL;
}
```

Figure 4.7: Internal data exchange algorithm
4.2 BPEL Data Flows

In Fig 4.7, given \((d_{node}, u_{node})\) as input, two sequences \(d_{path}, u_{path}\) are created (line 2-3). Starting from \(d_{node}, u_{node}\), it iteratively gets the parent nodes from the current nodes, denoted by \(d_{par}, u_{par}\), until both root nodes are reached (line 7). The while contains two parts. First, it checks whether a common ancestor node is reached. 1.1) \(d_{par}\) is in \(u_{path}\)(line 9-14): if \(u_{node}\) itself is the parent of \(d_{node}\), then the output path is \((d_{node}, u_{node})\) (line 10); otherwise, the output path is the reversed elements of the \(u_{path}\) before the common node. 1.2) \(d_{par}\) is not in \(u_{path}\)(line 15-18), \(d_{par}\) is added to \(d_{path}\) and \(d_{par}\) becomes the current node. Second, similarly it checks whether a common ancestor node is reached. 2.1) \(u_{par}\) is in \(d_{path}\)(line 21-26): if \(d_{node}\) itself is the parent of \(u_{node}\), then the output path is \((d_{node}, u_{node})\) (line 22); otherwise, the output path is the reversed elements of the \(u_{path}\) before the common node. 2.2) \(u_{par}\) is not in \(d_{path}\)(line 27-30), \(u_{par}\) is added to \(u_{path}\) and \(u_{par}\) becomes the current node.

We use the variable \(x\) in Fig 4.6 as an example. From (a) we get the machine-exdu-pairs for \(x\) are \((M_A, M_w), (M_A, M_s), (M_C, M_w),\) and \((M_C, M_s)\). When applying the algorithm to the example, the data exchange paths for the above machine-exdu-pairs would be \((M_A, M_f, M_w), (M_A, M_f, M_s), (M_C, M_w),\) and \((M_C, M_w, M_s)\), respectively.

B. External and Global Data Exchange Models

Since the communication scheme between web services is message passing, BPEL processes exchange data by passing messages. There exist two ways to capture the interactions between BPEL processes: top-down and bottom-up approaches. The top-down approach is firstly to design a conversation protocol to capture global interactions of BPEL processes, and secondly to design the BPEL models to implement the conversation protocol. The bottom-up approach is firstly to design BPEL processes, and secondly to derive the global interactions from the BPEL processes. [25] points out the advantage of the top-down approach over the bottom-up approach. However, from the testing point of view, it is especially important to verify the correctness of the BPEL interactions when a conversation protocol is missing. In our framework, we assume
4.2 BPEL Data Flows

that there is no conversation protocol to guide the design of BPEL process interactions. Instead, we verify the correctness of BPEL process interactions by deriving a BPEL external data exchange model from individual BPEL processes. The external data exchange model can be easily constructed from BPEL activities by identifying which partnerLink a message is sent to or received from.

In Fig 4.8, the example on the left shows the messages passing between BPEL processes for the loan approval example, and the example on the right shows the derived BPEL external data exchange model where customer component is chosen as a tester and the rest of the components are selected as a SUT.

Figure 4.8: External data exchange model

A global data exchange model is the union of the internal data exchange models of individual BPEL processes and the external data exchange model of these BPEL processes. See section for the loan approval example.

4.2.4 Deriving BPEL Data Flows

After modelling how data exchanges within a BPEL process and across BPEL processes, data flows can be derived for each variable so that we can check whether a defined variable will be used later and whether a used variable has been previously defined. Let $x$ be a variable. A du-pair of $x$ is a transition pair $(t_i, t_j)$ where $t_i$ is with $df(x)$ and $t_j$ is with $us(x)$. A def-clear path with respect to $x$ is a transition sequence $(t_1, t_2, ..., t_n)$ where there is no $df(x)$ in any of the transitions $t_2, t_3, ..., t_{n-1}$. A data flow (or du-path) of $x$ is a transition sequence that $(t_i, t_j)$ is a du-pair and there is a def-clear path from $t_i$ to $t_j$ with respect to $x$. 
Definition 23. Let $M_1, M_2$ be two machines, there is a data flow from $M_1$ to $M_2$, i.e. $t_2 \in T_2$ is data dependant on $t_1 \in T_1$, iff there exists a variable $x$ such that

1) $t_1$ is with $df(x)$. 
2) $t_2$ is with $us(x)$. 
3) There exists a def-clear path from $t_1$ to $t_2$ with respect to $x$.

The data flows for a variable $x$ can be constructed by identifying du-pairs, and checking whether there is a def-clear path between the du-pairs. The data flows for a variable $x$ can be derived automatically by model checking techniques, based on the annotations $df(x), us(x)$ of transitions. For each du-pair $(t_i, t_j)$ of variable $x$, where $t_i \in T_{M_i}, t_j \in T_{M_j}$. In $t_j$, $x$ needs to be asserted that $x$ has been defined previously. For the purpose of illustration, here we use machine-du-pairs retrieved from du-pairs, and use machine sequences retrieved from the transition sequences of data flows. The pair of machines with $df(x)$ and $us(x)$ is called machine-du-pairs. Based on the du-pairs of a variable, model checking techniques can be used to derive data flows for the variable. See section for the loan approval example.

### 4.3 BPEL Basic Activities

In this section, we will show the usage of WSA to model BPEL basic activities: `receive`, `reply`, `assign`, `invoke`, `throw`, `terminate`, `empty`; the time related `wait` activity is not modelled in WSA. We use $msg(x)$ to denote the internal data exchange message for variable $x$. Let $M_1, M_2$ be two machines, if $M_1$ sends $msg(x)$ to $M_2$, then $M_1$ is a `dataSender` of $M_2$ and $M_2$ is a `dataReceiver` of $M_1$, with respect to $x$. For simplicity, we use $dataSenders(x)$ to denote $(dataSender_1?msg(x);..;dataSender_n?msg(x))$, where a machine may have 1..$n$ dataSenders. Similarly, we use $dataReceivers(x)$ to denote the sending event list $(dataReceiver_1?msg(x);..;dataReceiver_n?msg(x))$, where a machine may have 1..$n$ dataReceivers.
4.3 BPEL Basic Activities

4.3.1 Receive and Reply

The *receive* and *reply* activities have the same machine layout as shown in Fig 4.9. For *receive* machine, after it is started, it can be interrupted before the message arrives \((t_{2.1})\), or it receives a message \(op(x)\) from an external partner. In the latter case, it sends \(msg(x)\) to its dataReceivers and ends \((t_{2.0})\). For *reply* machine, after being started and receiving \(msg(y)\) from its dataSenders, it can be interrupted before sending the reply message, or it sends the reply message with \(y\) as parameter to the external partner \((t_{2.0})\).

BPEL code:
\[
\begin{align*}
&\text{<receive partnerLink=pl portType=pt operation=op variable=x/>} \\
&\text{<reply partnerLink=pl portType=pt operation=op variable=y/>}
\end{align*}
\]

WSA transitions:
\[
\begin{align*}
&12.1 : s2 \rightarrow s1, \text{parent?stop/parent!done} \\
&12.0 : s2 \rightarrow s3, \text{pl?op(x) \& parent?stop/dataReceivers(x); parent!done} \\
&\text{receive:} \\
&10.1 : s0 \rightarrow s2, \text{parent?start} \\
&12.0 : s2 \rightarrow s3, \text{pl!op(y); parent!done} \\
&\text{reply:} \\
&10.1 : s0 \rightarrow s2, \text{parent?start \& dataSenders(y)} \\
&12.0 : s2 \rightarrow s3, \text{parent?stop/dataSenders(y); parent!done}
\end{align*}
\]

Figure 4.9: The machine of the receive and reply activities

4.3.2 Assign, Throw, Terminate, Empty, Compensate

The *assign*, *throw*, *terminate* and *empty* activities have the same machine layout, again shown in Fig 4.9. For assign machine, suppose \(x\) is a BPEL variable, the machine receives \(msg(x)\) from its dataSenders and then it sends \(msg(x)\) to its dataReceivers. The *throw* machine sends a fault message to its parent machine. The *terminate* machine sends a termination message to the process machine directly, so that the whole process can be stopped immediately. The *empty* machine ends after it is started.

BPEL provides named and unnamed *compensate* activities. The named *compensate* activity is used to start compensation on a specified inner scope that has already completed successfully, and the unnamed *compensate* activity is used to start compensation on all inner scopes that have already completed successfully. A *compensate*
activity can only be enclosed in a *faultHandlers* activity, *compensationHandler* activity, or implicit created *defaultTerminationHandler* activity. Such activities are called FCT for short. When the *compensate* machine is started, it sends the compensate message to the machine of the nearest enclosing FCT activity, with a parameter which holds either the target scope name or a null value. As long as the FCT machine receives a compensate message from the enclosing compensate machine, FCT machine starts a compensateManager (see section 4.4.9) that take charge of the invocations of compensationHandlers.

![Diagram of BPEL code](image)

---

**Figure 4.10:** The machine of the assign, throw, terminate, and empty activities

### 4.3.3 Invoke

Invocation comes in two forms, asynchronisation, in which the requester does not wait for a reply, and synchronisation, in which the requester does wait for a reply. The *invoke* machines with and without fault and interruption are shown on parts (1)(2) of Fig 4.11, respectively. Let $M_I$ be the invoke machine. In the synchronisation case, $M_I$ receives $msg(x)$ from its dataSenders and sends request message $op(x)$ to the external partner $((t_0, t_2, 2))$. With interruption, $M_I$ ends before sending the request. In the asynchronisation case, a normal scenario without fault or interruption is $\langle t_0, t_2, t_3, 0 \rangle$. After receiving $msg(x)$ from its dataSenders, $M_I$ sends the request $op(x)$ to the external
partner. $M_I$ waits for a response message $op(y)$ from the partner, and sends $msg(y)$ to its dataReceivers. Alternatively, when $M_I$ receives a stop message, it stops immediately ($t_{2.1}, t_{3.1}$). When $M_I$ receives a fault message from the external partner, it propagates the fault to its parent ($t_{3.3}$). Note that the fault parameter of the invoke's response message is not explicitly declared in the BPEL model but declared in related WSDL files.

**BPEL code:**

```xml
<invoke partnerLink=pl portType=pt operation=op inputVariable=x/>
<invoke partnerLink=pl portType=pt operation=op inputVariable=x outputVariable=y/>
```

**WSA transitions:**

1.1: $s_0 \rightarrow s_2$, parent?start & dataSenders(x)

(A) If invoke has no @outputVariable, then:

- $t_2: s_2 \rightarrow s_4$, parent?stop(tm)/pl!op(x); parent!done

(B) If invoke has @outputVariable, then:

- $t_0: s_2 \rightarrow s_3$, parent?stop(tm)/pl!op(x); parent!done

- $t_0: s_3 \rightarrow s_4$, pl!op(y) & parent?stop(tm)/pl!op(y); parent!done

- $t_1: s_3 \rightarrow s_1$, parent?stop(tm)/parent!done

- $t_3: s_3 \rightarrow s_3$, pl?op(y) & parent?stop(tm)/parent!fault(f)

Figure 4.11: The machines of synchronous and asynchronous invoke activities

In the case of asynchronous form of invoke activity, it may have inline faultHandlers and a compensationHandler. To make our modelling easier, the `invoke` code is reformed to an equivalent version without inline handlers in an intermediate BPEL code. An example is shown in Fig 4.12. The left is the invoke activity with inline handlers and the right is the equivalent version without inline handlers. The semantics of `scope`, `faultHandlers`, and `compensationHandler` activities will be covered in section 4.4.

### 4.4 BPEL Structured Activities

BPEL structured activities include `sequence`, `flow`, `while`, `switch`, `pick`, `scope`, `evenHandlers`, `faultHandlers`, and `compensationHandlers`. In WSA, each structured activity corresponds to at least one compound machine, as stated in section 4.1.1. Note that the machines of `switch`, `pick`, and `faultHandlers` activities have the same machine lay-
4.4 BPEL Structured Activities

out, but of course, different semantics. In order to simplify the mapping from BPEL to WSA, some additional code will be introduced in the intermediate BPEL code: 1) for the switch activity, the otherwise segment is added when it is missing; 2) for the scope activity, the terminationHandler segment is added, and the default compensationHandler segment is added when it is missing; 3) for the faultHandlers activity, the catchAll segment is added when it is missing.

For BPEL structured activities, we mainly illustrate the control dependencies, and for simplicity, do not include the messages related to internal message exchanges in the diagram. A BPEL structured activity may enclose one or more BPEL activities. We introduce the following for the purpose of clarification.

- Let $A_1, A_2, A_3$ be BPEL activities. If there is no activity sequentially in-between $A_1$ and $A_2$, then $A_2$ is said a direct enclosed activity of $A_1$. If $A_2$ is in-between $A_1$ and $A_3$, then both $A_2, A_3$ are enclosed in $A_1$, and $A_3$ is directly enclosed in $A_2$ which in turn is directly enclosed in $A_1$.

- Let $Q_1, Q_2$ be two BPEL scope activities. When there is no scope, faultHandler, or compensationHandler activity in-between $Q_1$ and $Q_2$, then $Q_1$ is called a direct outer scope of $Q_2$, and $Q_2$ is called a direct inner scope of $Q_1$. 

Figure 4.12: Invoke activity with and without inline handlers
4.4 BPEL Structured Activities

4.4.1 Sequence

In the sequence activity, the direct enclosed activities execute in sequential order. The full version of the sequence machine is shown in part (1) of Fig. 4.13. The normal scenario is modelled in part (2) of Fig 4.13 with transition sequence \( (t_{0.1}, t_{2.2}, \ldots, t_{(n+1).2}) \) where \( n \) is the number of direct enclosed activities of the sequence. Let \( M_{\text{seq}} \) denote the sequence machine. When \( M_{\text{seq}} \) is started, it starts the first child machine \( (t_{0.1}) \). After receiving the current child’s done message, \( M_{\text{seq}} \) starts the next child machine, and this step continues until the last child is started \( (t_{2.2}, \ldots, t_{n.2}) \). When the last child finishes, \( M_{\text{seq}} \) ends normally \( (t_{(n+1).2}) \). Alternatively, \( M_{\text{seq}} \) ends abnormally at \( s_1 \) when an interruption arises \( ((t_i.3, t_i.0, t_i.1)) \) or \( (t_{i.0}, t_{i.1}) \), \( 2 \leq i \leq n + 1 \).

![Sequence Activity Machine Transitions]

Figure 4.13: The machine of the sequence activity

4.4.2 Flow

The flow activity enables concurrency. A flow starts all direct enclosed activities at the same time when it has been started, and completes when all the direct enclosed activities have completed. In a flow, the links specify the synchronisation dependencies between activities. Each link has exactly one source activity and one target activity. We do not model links in a flow machine, we create a linkWrapper machine to capture
4.4 BPEL Structured Activities

the synchronisation feature for each activity with links, which is covered in section 4.1.3.

A normal scenario is modelled in part (2) of Fig 4.14, with the transition sequence \( (t_{0.1}, t_{2.2}) \). Let \( M_f \) denote the flow machine. When \( M_f \) is started, it sends start messages to all its child machines \( (t_{0.1}) \). \( M_f \) ends normally after receiving all the done messages from its children \( (t_{2.2}) \). Alternatively, part (1) of Fig 4.14 shows the full version of flow machine, with \( s_1 \) as the abnormal end state when an interruption occurs. As long as one of the child machines raises a fault, the flow forwards the fault to its parent machine. If the flow receives faults from more than one machine, it propagates the first arrived fault (i.e. the first fault in the FIFO faultQueue) to its parent machine.

\[ \]

\[ \]

**Figure 4.14: The machine of the flow activity**

4.4.3 While

The *while* activity defines looping behaviour. The while condition is evaluated at the beginning of each iteration. If the condition is true, the direct enclosed activity executes repeatedly; otherwise the loop ends. Two normal scenarios are modelled in part (2) of Fig 4.15, with transition sequences \( (t_{0.1}, t_{2.2}, t_{3.2}, t_{2.3}) \) when the condition holds at least once, and \( (t_{0.1}, t_{2.3}) \) when the condition never holds. Let \( M_w \) denote the while machine. \( M_w \) evaluates the condition when it is started. If the condition holds, it enters a loop to start the child machine and wait for child’s done message \( ((t_{2.2}, t_{3.2})) \).
The loop continues until the condition is evaluated to false. The while machine with consideration of faults and interruptions is shown in part (1) of Fig 4.15.

4.4.4 Switch

The switch activity provides conditional behaviour, where each direct enclosed activity is guarded by a condition. A switch consists of 1..* case segments and 0..1 otherwise segment. The case conditions are evaluated in the order in which they appear, while the otherwise segment executes if no case condition holds. In order to make our modelling easier, when the otherwise segment is missing, a default otherwise segment is added to the switch activity in the intermediate BPEL code. A normal scenario is modelled in part (2) of Fig 4.16, with transition sequence \( (t_{0,1}, t_{2,j}, t_{(j+2),2}) \) where \( 1 \leq j \leq n \), \( n \) is the sum number of case and otherwise tags in switch. To fulfill the sequential condition evaluation, the guard of each branch \( (t_{2,i}) \) is modelled according three rules defined in the WSA transitions of Fig 4.16. Part (1) of Fig 4.16 shows the switch machine with consideration of faults and interruptions.

Figure 4.15: The machine of the while activity
4.4 BPEL Structured Activities

4.4.5 Pick

Similar to the *switch* activity, the *pick* activity also provides conditional behaviour, but each direct enclosed activity of it is guarded by a message. A pick activity consists of 1..* onMessage* segments and 0..* onAlarm* segments. Each onMessage segment waits for a matching incoming message, and each onAlarm segment waits for a matching time-related alarm. In WSA, we do not model the time related onAlarm behaviour.

When a message arrives, the onMessage segment with matching message executes. When multiple messages arrive simultaneously, one message will be selected, and the choice of message is implementation dependent. A normal scenario is modelled in part (2) of Fig 4.17, with transition sequence \( \langle t_{0.1}, t_{2.1}, t_{(j+2).2} \rangle \) where \( 1 \leq j \leq n \), \( n \) is the number of onMessage segments. Part (1) of Fig 4.17 shows the pick machine with consideration of faults and interruptions.

![Figure 4.16: The machine of the switch activity](image)
4.4.6 Scope

A. Overview

The scope activity provides a behaviour context for declaring variables, event handling, fault handling, and compensation handling. The fault handling and compensation handling features are important to support long running business transactions. A scope allows the declaring of variables which are accessible to any activity inside the scope. The outmost BPEL process provides a global scope, called process scope, which shares similar semantics of scope (see section 4.4.7). A scope must have a primary activity, and optionally contain an eventHandlers activity, a faultHandlers activity, and a compensationHandler activity. The primary activity can be a BPEL basic or structured activity. When a scope is entered, its primary activity, eventHandlers, and faultHandlers starts to run concurrently. When a scope finishes successfully (i.e. no fault or interruption occurs), the scope starts the compensationHandlers.

Scope and Event Handlers: In a scope, the eventHandlers activity runs concurrently with the primary activity, the eventHandlers activity is enabled (resp. disabled)
when the primary activity starts (resp. completes). In WSA, we use start and disable messages to model such enablement and disablement, respectively. Note that the eventHandlers activity continues running until it is disabled or stopped by the scope.

**Scope and Fault Handlers:** Upon receiving a fault, the scope forwards the fault to its faultHandlers. The scope enables the faultHandlers after the scope’s running primary activity and eventHandlers have been stopped. If the scope’s faultHandlers cannot handle a fault, the fault will be re-thrown to the scope, so that the scope can propagate the fault to the direct enclosing activity. A scope ends abnormally if a fault arrives, no matter whether or not the fault can be handled or re-thrown by its faultHandlers.

**Scope and Compensation Handler:** It is pointed out in [10] that a scope’s compensationHandler is installed only when the scope completes normally. However, it is not clear whether the scope completing normally means that: 1) the scope’s primary activity completes successfully, or 2) both scope’s primary activity and eventHandlers activity complete successfully. We adopt the latter interpretation, because the eventHandlers are considered as a part of the normal processing of a scope, and a fault thrown from the eventHandlers will be forwarded to the current scope’s faultHandlers. Thus, a scope is said to complete successfully when all its normal processing activities complete normally. Thereafter, the compensationHandler is installed for possible later invocation. If the process reaches a point where compensation will no longer be required, compensation activities can be forgotten.

A scope needs to be compensated when its compensationHandler is explicitly or implicitly invoked by a `compensate` activity of the direct outer scope’s faultHandlers or compensationHandler. In the case of a scope that needs to be compensated but its compensationHandler activity is missing, an implicit compensation handler will run. Such an implicit compensation handler is introduced in section 13.4.1 of BPEL 1.1 [10] and is explicitly defined as a `default compensationHandler` in section 12.5.1 of BPEL 2.0 [6], as shown in the BPEL code of Fig 4.19. It encloses an unnamed `compensate` activity that invokes the compensationHandlers of corresponding scopes.
The semantics of a \textit{compensationHandler} activity will be covered in section 4.4.9. When a compensationHandler is missing, such a default compensationHandler activity will be added to the scope activity in the intermediate BPEL code.

\textbf{Scope Interruption:} As stated in section 4.1.2, there exist two causes of interruptions. An activity can be interrupted by its direct enclosing activity due to: 1) a fault propagation, or 2) a terminate message. For simplicity, we call them fault-stop and terminate-stop, respectively. Except for the \textit{scope} activity, the other BPEL activities handle these two kinds of interruptions in the same way, by stopping their enclosing activities and then ending abnormally.

When a terminate activity is reached, all currently running activities must be terminated immediately without any fault handling or compensation behaviour. Therefore, when receiving a terminate-stop, the scope handles it the same as other activities, by stopping their direct enclosing activities and ends abnormally. The direct enclosing activities include the running primary activity, eventHandlers, and faultHandlers. Regarding the compensationHandler, all the compensationHandlers will be stopped directly by the \textit{process scope}.

When receiving a fault-stop, the \textit{scope} activity can control such interruption to some degree. By receiving a fault-stop, if the scope’s faultHandlers activity is running, it will do nothing but wait for the faultHandlers to complete; otherwise, if its fault-Handlers activity is not running, then the scope will start a special fault handler. This special faultHandler is introduced in section 13.4.2 of BPEL 1.1 [10] and explicitly defined as default \textit{terminationHandler} in section 12.5.1 of BPEL 2.0 [6], as shown in Fig 4.18. When the default terminationHandler is started, it starts the unnamed \textit{compensate} activity. By receiving a compensate message, it enables a compensateManager machine \textit{CH} that manages the invocations of compensationHandlers (see section 4.4.9). The default \textit{terminationHandler} activity will be added to the scope activity in the intermediate BPEL code.

\textit{B. Scope in WSA}
Let $M_Q$ be the scope machine, and $PA, EHS, FHS, CH, TH$ be the machines of the primary, eventHandlers, faultHandlers, compensationHandler, default TerminationHandler activities, respectively. The $FHS, CH, TH$ are called $FCT$ for short. Also a scope machine has a special child compensateManager $CM$ as the supporting machine to manage the compensation invocation. Note that a scope machine will start its $CM, CH$, but it will not wait for them to finish. Since a started $CM$ may not be enabled, and a started $CH$ may or may not be invoked, if the scope waits for them to finish, the parent machine of scope that waits for the scope’s done message cannot progress, which in turn causes other related machines to wait. The $CM$ is enabled by the scope’s $FCT$, so $CM$ can propagate fault to, receive stop from, and send done message to $FCT$. The $CH$ can propagate fault to, receive stop from, and send done message to the $CM$ of the current scope’s direct outer scope. The semantics of compensateManager machines and compensationHandler activities will be covered in section 4.4.9.

The scope machine has three Boolean variables $faultStop, termStop, childFault$, with false as default values. The $faultStop, termStop$ indicate whether an incoming stop message is a fault-stop or terminate-stop. The $childFault$ with true value indicates that the scope receives a fault from $PA$ or $EHS$. In this case the scope cannot start the compensationHandler.

**Scope with faultHandlers:** The scope machine with faultHandlers $FHS$ is shown in Fig 4.19. When a scope ends normally without receiving a fault from its $PA$
or EHS, the scope is in a successful-completed mode; otherwise the scope is in an unsuccessful-completed model when it receives a fault or it is interrupted. Two normal scenarios are modelled in part (2) of Fig 4.19. Firstly, if EHS exists, $M_Q$ starts $PA, EHS, FHS, CM$, after $PA$ has finished, the scope disables EHS. After EHS has finishes, $M_Q$ discharges $FH$. Finally, when the $FH$ has finished, $M_Q$ ends normally in the successful-completed mode and starts $CH ((t_{0.1}, t_{2.2}, t_{3.2}, t_{5.2}))$. Secondly, if EHS does not exist, $M_Q$ starts $PA, FHS, CM$, after $PA$ has finished, $M_Q$ discharges $FHS$. Thereafter, $M_Q$ ends normally in a successful-completed mode and starts $CH ((t_{0.1}, t_{2.4}, t_{5.2}))$.

The full version scope machine with faults and interruptions is shown in part (1) of Fig 4.19. When receiving a fault from either $PA$ or EHS, $M_Q$ stops both of them and forwards the fault to $FHS (t_{2.3}, t_{3.3})$. At state $s_4$, $M_Q$ waits for the running $PA, EHS$ to finish before enabling $FHS (t_{4.0})$. In this case, $M_Q$ ends normally in an unsuccessful-completed mode ($t_{5.4}$). When receiving a fault re-thrown from $FHS$, $M_Q$ forwards the fault to its parent machine ($t_{5.3}$). For interruption, $M_Q$ responds to fault-stop and terminate-stop messages differently, as follows. First, if $M_Q$ receives a fault-stop when $FHS$ has not been enabled, $M_Q$ stops the running $PA, EHS$, discharges $FHS$, and starts $TH$. After the running child machines have finished, $M_Q$ ends abnormally($((t_{2.0}, t_{2.1}), (t_{3.0}, t_{3.1}))$). Second, when receiving a terminate-stop, $M_Q$ stops the running $PA, EHS, FHS$, and ends abnormally after the running child machines have finished($((t_{2.00}, t_{2.01}), (t_{3.00}, t_{3.01}), (t_{5.00}, t_{5.01}))$). As long as the scope ends abnormally, it is in an unsuccessful-completed mode.

**Scope without faultHandlers:** Since the scope has no faultHandlers, the scope can either end abnormally in an unsuccessful-completed mode or end normally in a successful-completed mode. Two normal scenarios are modelled in part (4) of Fig 4.20. Firstly, if EHS exists, $M_Q$ starts $PA, EHS, CM$, $M_Q$ disables EHS after $PA$ has finished. When EHS has finishes, $M_Q$ ends normally in the successful-completed mode and starts $CH$, i.e. $((t_{0.1}, t_{2.2}, t_{3.2}))$. Secondly, if EHS does not exist, $M_Q$ starts $PA, CM$. After $PA$ has finished, $M_Q$ ends normally and starts $CH ((t_{0.1}, t_{2.4}))$. 


The full version scope machine with faults and interruptions is shown in (3) of Fig. 4.20. When the scope receives a fault, it propagates the fault to its parent machine (t2.3, t3.3). For interruption, when receiving a fault-stop, MQ stops the running PA, EHS and starts TH. After its running child machines have finished, MQ ends abnormally ((t2.0, t2.1), (t3.0, t3.1)). When receiving a terminate-stop, MQ stops the running PA, EHS and ends abnormally after the running child machines have finished ((t2.00, t2.01), (t3.00, t3.01)).
4.4 Process Scope

The process activity shares most similar semantics of a general scope activity with some differences. We illustrate the differences in this section. The partnerLinks (the set of BPEL processes with which the current process interact) must be declared in the process. Since a process activity cannot be enclosed in a flow activity, it cannot contain any link. Thus, no linkWrapper machine is associated with a process activity. A process activity must have a primary activity, and optionally eventHandlers and faultHandler activities. In section 13.3.1 of BPEL 1.1 [10], it states that a process can be compensated after normal completion by platform-specific means. However, it is not clear how to invoke the compensationHandler of process scope. In section 12 of BPEL 2.0 [10], it states that a compensationHandler cannot be attached to a process. We adopt the latter semantics. For fault handling, the process handles a fault in the same way as scope but if a fault is re-thrown from the process's faultHandler, the process ends without propagating the fault. Hence, the process only can receive...
a terminate-stop but no fault-stop. As a result, different from the scope activity, the default compensationHandler and the default terminationHandler activities will not be added to the process activity in the intermediate BPEL code.

The process machine is shown in Fig 4.21 below, where the machines of process with and without faultHandlers are shown in parts (1)(2) and (3)(4), respectively. Let $M_P$ be the process machine, and $PA, EHS, FHS$ be the machines of the primary, eventHandlers and faultHandlers activities, respectively. A process machine has similar structure as the scope machine. We mainly shows the difference here. $M_P$ has a Boolean variable $tm$ with false as the default value, which is the parameter of a stop message so that the scope machines can identify whether the incoming stop message is a fault-stop or terminate-stop. Since a process activity is the top-level activity, so a process machine cannot receive a terminate-stop or fault-stop message from the parent machine, the variables $faultStop$, $termStop$ are not needed in $M_P$. Also, since the process activity is not associated with any compensationHandler, the variable $childFault$ is not needed in $M_P$ to distinguish whether the process ends normally in a successful-completed mode or not.

Whenever $M_P$ is entering a final state, it no longer requires any compensation. If a $CM$ or a $CH$ is still running when a process is going to finish, this indicates that the $CM$ has not been enabled and the $CH$ has not been invoked by any enabled $CM$. However, it is too complex to identify which $CM$s, $CH$s are running. We use a simple solution to stop all the $CM$s, $CH$s of the enclosed scopes.

**Process with faultHandlers:** Part (2) of Fig 4.21 shows the normal scenarios of the process machine, which are similar to the ones of the scope machine, except that $M_P$ will stop all running compensationHandlers of the enclosed scopes, if any, and ends normally. The scenarios follow the transition sequences $(t_0,1, t_2,2, t_3,2, t_5,2), (t_0,1, t_2,4, t_5,2)$. The process machine with faults and interruptions is shown in (1) of Fig 4.21. When $M_P$ receives a fault from either $PA$ or $EHS$, it stops both of them and forwards the fault to $FHS$ $(t_2,3, t_3,3)$. When $M_P$ receives a fault re-thrown from $FHS$, it does not propagate the fault but ends abnormally $(t_3,3)$. Since if a terminate activity is reached
then all running activities must be terminated immediately; a terminate machine will send the termination message directly to the process machine. When $M_P$ receives a termination message, it stops all the running child machines by sending stop messages with true-value $tm$, i.e. terminate-stop ($t_2.0, t_3.0, t_5.0$). After receiving the done notifications from its child machines, $M_P$ ends abnormally ($t_2.1, t_3.1, t_5.1$).

**Process without faultHandlers:** Part (4) of Fig 4.21 shows the normal scenarios of process machine, which are also similar to the ones in scope machine. They follow the transition sequences $\langle t_0.1, t_2.2, t_3.2 \rangle$, $\langle t_0.1, t_2.3 \rangle$. The full machine with fault and interruption consideration is shown in part (3) of Fig 4.21. As long as the $M_P$ receives

**Figure 4.21:** The machine of the process activity
4.4 BPEL Structured Activities

a fault, it stops the running P A, E H S and ends abnormally (t_{2.3}, t_{3.3}). For interruption.
when receiving a terminate message, M P stops the running P A, E H S. After its running
child machines have finished, M P ends abnormally ((t_{2.0}, t_{2.1}), (t_{3.0}, t_{3.1})).

4.4.8 Fault Handlers

The faultHandlers activity is used explicitly to catch faults and handle them by ex-
ecuting the direct enclosed activity. Similar to the switch and pick activities, the
faulthandlers activity also provides conditional behaviour, but each direct enclosed ac-
tivity of it is guarded by a fault message. The faulthandlers activity is always directly
enclosed by a scope or process activity. A faultHandlers has 1..* catch segments and
0..1 catchAll segment. Each catch segment handles a matching fault message, and the
catchAll segment handles any fault that cannot be caught by any other catch segment.
If the catchAll segment is missing, an implicit fault handler will run. Such an implicit
fault handler is introduced in section 13.4.1 of BPEL 1.1 [10] and explicitly defined
as default faultHandler in section 12.5.1 of BPEL 2.0 [6], shown in the BPEL code of
Fig 4.22. When the default faultHandler is entered, it starts the unnamed compensate
activity and rethrows the unhandled fault. When the catchAll segment is missing, the
default faultHandler will be added to the faultHandlers activity in the intermediate
BPEL code.

Let FHS be the machine of the faultHandlers activity. The normal scenario is
modelled in part (2) of Fig 4.22, with transition sequence \( \langle t_{0.1}, t_{2.j}, t_{(j+2).2} \rangle \) where
1 \( \leq j \leq n \), n is the sum number of catch and catchAll segments. To fulfill the
fault name and fault variable matching evaluation, the guard of each segment \( t_{2.j} \)
is modelled according two rules defined in the WSA transitions of Fig 4.22.

Part (1) of Fig 4.22 shows the full machine version with consideration of compen-
sation, faults, and interruptions. When FHS receives a compensate message from
the compensate machine, it enables the compensateManager machine CH that takes
charge of the invocations of compensationHandlers(see section 4.4.9) \( t_{(j+2).1} \), and
waits for CH to finish. At state s_{2}, FHS is started but not enabled, when receiving
4.4 BPEL Structured Activities

BPEL code:
<faultHandlers>
<catch faultName="name" faultVariable="name">
activity1
</catch>
<catchAll>
activity2
</catchAll>
</faultHandlers>

Default fault handler:
<catchAll>
<sequence>
<compensate />
<rethrow />
</sequence>
</catchAll>

WSA transitions:
In a faultHandlers, let \( j \) denote the index of catch tags when \( 0 < j < n \), and the index of catchAll tag when \( j = n \), where \( n - 1 \) is the number of catch tags. Let \( i = j + 2 \) be a state ID. Let CM denotes compensateManager.

We have:
1. If \( j < n \), then guard\( j \): \( \text{guard} j = \text{guard}(\text{index} = \text{faultName} \land \text{faultVariable} = \text{faultVariable}) \lor (\text{index} = \text{faultName} \land \text{faultVariable} = \text{null}) \lor (\text{faultVariable} = \text{faultVariable} \land \text{faultName} = \text{null}) \)

2. If \( j = n \), then guard\( n \):

\( \text{guard}(\text{guard} 1 \ldots \text{guard} n \ldots) \)

Figure 4.22: The machine of the faultHandlers activity

a terminate-stop or a discharge message, \( FHS \) ends without executing any branches \((t_{2,0})\). At states \( s_3 \ldots s_{n+2} \), \( FHS \) can be interrupted by a terminate-stop \((t_{4,0}, t_{4,1})\).

Note the faultHandler only receives terminate-stop, because when the corresponding scope is interrupted by a fault-stop, the scope will not stop faultHandlers but wait for it to complete (see scope interruption in section 4.4.6). As a consequence, when \( FHS \) receives a fault from its child machine, it stops the child and the running compensateManager and propagates the fault to its parent \((t_{4,3})\).
4.4.9 Compensation Handlers Invocation

In order to support long running transactions, error handling in business processes relies heavily on the concept of compensation to reverse the effects of previous committed activities. BPEL provides compensationHandler and compensate activities to control the reversal, by allowing the user to define specific undo activities. The BPEL compensation only targets local transaction within a single business process, and a transaction protocol language is required for multiple BPEL processes [10]. The compensation-Handler activity attempts to undo the work of a completed scope by the pre-defined activities enclosed in the compensationHandler. A scope’s compensationHandler can only be invoked when a compensate activity executes. The compensate activity must be enclosed in the faultHandlers, default terminationHandler, or compensationHandlers of current scope’s direct outer scope.

An example of Fig 4.23 shows the hierarchy of invoking compensationHandler activities. Let $Q_i$ be a scope and $P_i, F_i, C_i, T_i$ be the primary, compensationHandler, faultHandlers, and default terminationHandlers activities of $Q_i$. $Q_1$ is the direct outer scope of $Q_2$ and $Q_3$, so $C_2$ and $C_3$ can be invoked when there exists a compensate activity in $F_1, C_1, T_1$. Also, $Q_2$ is the direct outer scope of $Q_4$, so $C_1$ can be invoked when there exists a compensate activity in $F_2, C_2, T_2$. Note that $C_4$ cannot be invoked by the handlers of $Q_1$ and $Q_3$ as they are not direct outer scopes of $Q_4$. For the compensate activity in $F_1, C_1, T_1$, if it is named $Q_2$ then only $C_2, C_3$ will be invoked; otherwise, if it is unnamed, then both $C_2, C_3$ will be invoked in the order that they are started.

![Figure 4.23: The hierarchy of the compensationHandler invocations](image_url)
In order to model this hierarchy, we define a compensateManager machine to associate each scope activity. For a scope $Q_i$, let $FHS_i, CH_i, TH_i$ be the machines for the faultHandlers, compensationHandler, and default TerminationHandler activities. When the scope $Q_i$ is started, the compensateManager $CM_i$ starts. When one of the $FHS_i, CH_i, TH_i$ receives a compensate message from a compensate machine, $CM_i$ is enabled. Because $FHS_i, CH_i, TH_i$ cannot run at the same time, $CM_i$ will be enabled once. In Fig 4.23, $CM_1$ can invoke $CH_2, CH_3$, and $CM_2$ can invoke $CH_4$.

A. Compensation Handler

A scope’s compensationHandler is available for invocation only when the scope completes normally. Invoking a compensationHandler that has not been installed is equivalent to an empty activity. In Fig 4.24, let $CH_i, CM_i$ be the compensationHandler and compensationManager of scope $Q_i$. Suppose the current scope is $Q_1$ and $Q_0$ is the direct outer scope of $Q_1$. The current compensationHandler machine is $CH_1$.

The normal scenario is shown in part (2) of Fig 4.24. When $CH_1$ is started, it sends an ok message to $CM_0$ for registering its availability ($t_{0.1}$). This message is important because in the case that $Q_1, Q_2$ are enclosed in a switch activity, if $CM_0$ needs to invokes $CM_1$ but $Q_1$ is not chosen to execute, then $CM_0$ will wait for the response from $CM_1$ forever. Thereafter, $CH_1$ starts its child machine after receiving a compensate invocation from $CM_0$ ($t_{2.0}$), and $CH_1$ ends when its child has finished.

Part (1) of Fig 4.24 shows the full machine version with consideration of compensation, faults, and interruptions. When $CH_1$ receives a stop message from the process machine before receiving a compensate invocation, it ends abnormally. When $CH_1$ receives a compensate message from the compensate machine, it enables the current compensateManager $CM_1$ ($t_{3.4}$) and waits for $CM_1$ to finish ($t_{3.2}$). At this stage, when $CH_1$ receives a stop message, it stops its child machines and the running compensationManager ($t_{3.0}$). According to section 13.4.3 of BPEL 1.1 [10], when a compensationHandler receives a fault, the fault will be propagated to the one that invoked compensationHandler. Hence, when $CH_1$ receives a fault, it propagates to the $CM_0$.
(t3.3).

Figure 4.24: The machine of the compensationHandler activity

Note that the compensationHandler machine \(CH_i\) does not return its done message to the parent machine scope \(Q_i\), instead \(CH_i\) returns the done message to its invoker, i.e. a compensateManager machine \(CM_0\).

B. CompensateManager Machine

Due to the tight relationship between a compensationHandler and its invoker, we define a compensateManager machine specially to handle the invocation. When a FCT (faultHandlers, compensationHandler, or default terminationHandler activity) receives a compensate message from its enclosing compensate activity, FCT enables the compensateManager. Let \(CH_i, CM_i\) be the compensationHandler and compensationManager of scope \(Q_i\). Suppose the current scope is \(Q_0\), and \(Q_1, \ldots, Q_n\) are the direct inner scopes of \(Q_0\). Let \(CM_0\) be the current compensateManager machine, shown in Fig 4.25. For the sake of simplicity, the full machine version with consideration of faults and inter-
ruptions is only included in the machine transition description but not shown in the machine layout. Its normal behaviour can be categorised into two cases.

**Case 1:**
- Let $Q_0$ be the current scope, it has direct inner scopes $Q_1, .., Q_n$.
- For compensate with scope name $Q_1$.
- Transition sequence: $(t_{0.2}, t_{1.2}, t_{1.3}, t_{2.0})$.
- At state $s_2$, $CM_0$ waits for the registration message $ok$ from $CH_1$. The ok message indicates that $CH_1$ is ready to be invoked.

**Case 2:**
- Let $Q_0$ be the current scope, it has 3 direct inner scopes $Q_1, Q_2, Q_3$.
- Suppose the completion order is $Q_3, Q_2, Q_1$.
- Transition sequence: $(t_{0.2}, t_{1.2}, t_{1.3}, t_{2.0}, t_{2.1}, t_{2.2}, t_{2.3}, t_{2.4})$.
- If it is the last CH, i.e. $j = n$, then:
- $t_{2.4}$: $s_4 \rightarrow s_5$, $CH_3 ? done \& (CH_1 ? fault(f) \lor FCT ? stop(tm)) \lor FCTdone$
- If it is the first CH, i.e. $j = 1$, then:
- $t_{2.2}$: $s_2 \rightarrow s_3$, $CH_1 ? ok \& (CH_1 ? fault(f) \lor FCT ? stop(tm)) \lor FCTdone$
- If it is not the first or last CH, i.e. $j \neq 1, j \neq n$, then:
- $t_{2.3}$: $s_i \rightarrow s_{i+1}$, $CH_i ? done \& CH_{i+1} ? ok \& (CH_i ? fault(f) \lor FCT ? stop(tm)) \lor FCTcompensate$

Figure 4.25: The compensateManager machine
message \((t_{2.0})\); otherwise \(CM_0\) ends without sending the invocation \((t_{2.2})\).

In the second case, there is no specific scope to be compensated. One of the normal scenarios is with transition sequence \((t_{0.2}, t_{2.2}, \ldots, t_{(n+1).2})\), where \(n\) is the number of the direct inner scopes. Such a machine layout is the same as the machine of the \textit{sequence} activity, where the corresponding compensationHandlers can be invoked one by one. To illustrate the alternative scenarios, we assume the current \(Q_0\) has direct inner scopes \(Q_1, Q_2, Q_3\) with the completion order \(Q_3, Q_2, Q_1\). The idea is similar to the first case: if the ok message from the current \(CH_j\) is not available, it checks whether the ok message of the next one \(CH_{j+1}\) is available, if it is available then it invokes \(CH_{j+1}\). Since the scope \(Q_0\) starts its compensateManager \(CM_0\) after \(Q_0\) has been started, and \(Q_0\) always starts earlier than \(Q_1, Q_2, Q_3\), this indicates that \(CM_0\) must start earlier than \(CH_1, CH_2, CH_3\). As a result, this ensures that a compensateManager is always started when the corresponding compensationHandlers want to register by sending ok messages.

Another interesting point here is to get the completion order of the direct inner scopes. For two arbitrary scopes \(Q_i, Q_j\), the completion order is decided based on the nearest common enclosing machine of \(Q_i, Q_j\). We illustrate this using the example in Fig 4.23, as follows.

- If \(P_i\) is a \textit{sequence} activity, then \(Q_2\) completes before \(Q_3\). The invocation order is \(CH_3, CH_2\).

- If \(P_i\) is a \textit{flow} activity, there exist two cases. Firstly, if \(Q_2\) and \(Q_3\) are data independent, then they can complete independently. In this case the invocation order can be either \(CH_2, CH_3\) or \(CH_3, CH_2\). Secondly, if \(Q_3\) is data dependent on \(Q_2\), which means either \(Q_2, Q_3\) are source and target activities of a \textit{flow} link respectively, or \(Q_3\) needs the data from \(Q_2\). In this case, since \(Q_2\) completes earlier than \(Q_3\), the invocation order is \(CH_3, CH_2\).

- If \(P_i\) is a \textit{switch} or \textit{pick} activity, then only one of \(Q_2, Q_3\) will execute. There is no order applied, only the compensationHandler of the executed scope will send
an ok message to the invoker.

4.4.10 Event Handlers

The eventHandlers activity consists of a set of concurrent activities `onMessage` and `onAlarm` activities. The `onMessage` and `onAlarm` activities handle external message events and system alarm events, respectively. An alarm event is carried out at most once, while a message event can occur multiple times when the scope is active. We do not model the time related `onAlarm` activities in WSA. In order to model the multiple messages and multi-thread behaviours of eventHandlers, we associate the eventHandlers activity with `EHS` machines, each `onMessage` activity with a message event handler machine `MEH`, and each thread of `onMessage` activity with a `MEH` thread machine. Each thread machine takes care of one message instance. We do not illustrate the thread machine here because it can be a machine for any BPEL activity, whose parent machine is `MEH`. For an `EHS` machine, its parent is a scope machine and its child machines are `MEHs`.

BPEL code:
```xml
<eventHandlers>
    <onMessage partnerLink="partner1" portType="pt1" operation="m1" variable="para1">
        activity1
    </onMessage>
    ....
    <onMessage partnerLink="partnern" portType="ptn" operation="mnn" variable="paran">
        activitynn
    </onMessage>
    <onAlarm for="duration">
        activityyn
    </onAlarm>
</eventHandlers>
```

Machine transitions:
1.0: s0 -> s2, parent?start & MEH1.start ... MEHn.start
1.1: s2 -> s1, parent?stop(fm) / stopStatus=true & MEH1.stop(fm) ... MEHn.stop(fm)
1.2: s2 -> s3, parent?disable & (MEH1?fault(f) ... MEHn?fault(f)) & parent?stop(fm) / MEH1?disable ... MEHn?disable
1.3: s2 -> s2, (MEH1?fault(f) ... MEHn?fault(f)) & parent?stop(fm) / parent?fault(f)
2.0: s3 -> s3, parent?stop(fm) / MEH1?stop(fm) ... MEHn?stop(fm) / stopStatus=true
2.1: s3 -> s1, parent?disable & MEH1?done & MEHn?done & parent?stop(fm) / parent?done
2.2: s3 -> s3, (MEH1?fault(f) ... MEHn?fault(f)) & parent?stop(fm) / parent?fault(f)
2.3: s3 -> s4, (MEH1?fault(f) ... MEHn?fault(f)) & parent?stop(fm) / parent?fault(f)

Figure 4.26: The machine of the eventHandlers activity
4.4 BPEL Structured Activities

The EHS machine is shown in Fig 4.26. The normal scenario is illustrated in (2) with transition sequence \((t_0.1, t_2.2, t_3.2)\). When EHS is started, it starts all MEH machines; the EHS will not end until it receives a disable message from the scope. When EHS receives a disable message, it forwards the message to its MEH machines. EHS ends after all MEH machines have completed. Alternatively, when EHS receives a fault from any MEH machine, it propagates the fault to its parent \((t_2.3, t_3.3)\). When EHS is interrupted, it stops the MEH machines and ends abnormally \(((t_2.0, t_2.1), (t_3.0, t_3.1))\).

Figure 4.27: The machine of the MEH for eventHandlers activity

Fig 4.27 shows the MEH machine, since a MEH may start a thread for each message event instance, and the thread number is unknown, we model this by adding a variable \(\text{count}\) for the current thread number. Part (2) Fig 4.27 shows the normal scenarios. The first normal scenario is \((t_0.1, t_2.1, t_2.4, t_3.0, t_3.2)\). When MEH is started, the \(\text{count}\) is initiated to zero \((t_0.1)\). The machine waits for an external message event to arrive at \(s_2\), a new thread machine is started for each message instance. When it receives an external message, it increases \(\text{count}\) by 1 and starts a new thread machine.
4.5 Summary

(t2.1). When it receives a disable message from its parent and the count is not zero, it enters state s3 to wait for its thread machine to finish (t2.4). When one of its thread machines has finished, it decreases the count by 1, until all the thread machines have finished (t3.0). When the count is zero, the MEH ends normally (t3.2). The second normal scenario is (t0.1, t2.5), where the MEH is disabled when count is zero, i.e. it has not started any thread machine.

Two alternative scenarios contain the transition sequence (t4.0, t4.1, t6.0, t6.1): when MEH is interrupted (t2.2, t3.1), it stops its thread machines one by one until all the thread machines have been stopped (t4.0), then it enters state, s6, to wait for the thread machines to finish (t4.1). When a thread machine has finished, it decreases the count by 1 until all the thread machines have finished ((t6.0, t6.1)).

4.5 Summary

In this chapter, we analysed the BPEL control dependencies, and illustrated how to capture these dependencies in our proposed web service automaton (WSA). We demonstrated how to model BPEL basic activities and structured activities in WSA. The most interesting BPEL features, including scoping, fault handling, and compensation handling, are also modelled. In addition, we also analysed the BPEL internal and external data dependencies. From our modelling BPEL in WSA, we believe our WSA is more suitable to model BPEL behaviour than the existing automata-based semantics, in that it can model most features of BPEL and it allows verification of BPEL data dependencies in addition to control dependencies. In the next chapter, we will present our automatic test framework, where the BPEL control and data dependencies can be verified, and then control flow and data flow testing can be manipulated.
Chapter 5

Automatic Test Framework

As well known, it is hard to define a logically consistent and complete set of interaction rules for asynchronously executing processes in a distributed system, so it is essential to apply automated validation tools in finding the inconsistencies of the interactions. Also, it is tedious, time-consuming, and error prone to create test cases manually from design models, especially for complex modelling languages like BPEL. It is desirable to apply existing model-based-testing techniques in the domain of web services, so that the functionality of the BPEL processes can be checked.

In the previous chapter, we analysed and modelled BPEL features in our proposed web service automata. Based on this analysis, we present a model checking based test framework for BPEL processes, as model checking is an effective technique with dual advantages of verification and test case generation.

In our framework, in the first phase the general properties of the BPEL processes will be verified, and in the second phase the functionalities of the BPEL processes will be checked. NuSMV [18] and SPIN [32], two of the most mature model checkers, are used as two alternative engines in our framework.
5.1 Model Checking Background

The idea behind model checking is to check whether a given model satisfies a given property, by exploring all alternatives of the given model. The model checking process is illustrated in Fig 5.1. As the two inputs of a model checker, the model is based on a finite state machine, and the property is expressed as a temporal logical formula. When the given property is not satisfied, the model checker outputs a set of counterexamples. A counterexample is an execution path that will take the finite state model from its initial state to a state where the violation occurs.

![Figure 5.1: The model checking approach](image)

The SPIN model checker supports LTL temporal logic, and the NuSMV model checker supports both LTL and CTL temporal logic. LTL (Linear Time Temporal Logic) views time as a sequence of states with no choice as to which state is next. The choice of next state is deterministic. CTL (Computation Tree Logic) views time as branching, so from a given branch alternative states may be reached. In our framework, we will use the LTL temporal operators $\Box$ (always), $\Diamond$ (eventually), $X$ (next), and $U$ (until); and the CTL temporal operators $A$ (for all paths), $E$ (there exists some path), $G$ (always), $F$ (finally), $X$ (next), and $U$ (until).

5.1.1 In Verification

The aim of the verification phase is to find errors in a design model, where the system properties for model checking will be encoded as desired properties. A desired property describes the expected behaviour of a given model, i.e. what the model should do. By checking a BPEL model against a desired property, if there is an error in the BPEL model that will violate the property, the counterexamples will be generated to show the causes of the error.
A formal classification of design errors is based on the terms safety and liveness. The intuition of a safety property is that something bad never happens. A safety property can be verified by evaluating individual properties of states. This verification can always be done by a reachability analysis. When a safety condition is violated, the violation can be detected after a finite number of steps. For instance, a safety property can be a vending machine never dispenses coffee and tea at the same time. A general safety property can be described as $\Box \neg \phi$ in LTL, and $AG \neg \phi$ in CTL. The intuition of liveness property is that something good eventually happens. A liveness property can be verified by looking at a complete execution. For instance, a liveness property can be the vending machine eventually supplies coffee after the coffee-button is selected. A general liveness property can be described as $\Diamond \phi$ in LTL, and $AF \phi$ in CTL.

The majority of properties belong to safety properties, such as deadlock-free and invariant preservation. An example of liveness properties is livelock-free. A livelock indicates that there exists a non-progress cycle. Both deadlock and livelock can be detected by reachability analysis. In section 5.4, we discuss some design errors that can be automatically detected in our framework.

### 5.1.2 In Testing

The attraction of applying model checking in testing is that model checking can automatically produce counterexamples, which can be the basis for test cases. Proposals for applying model checking in coverage-based testing were made by [30, 33, 52]. The idea is to use a model checker to find test cases by formulating test purposes as trap properties. An example of a test purpose is 'a state is reachable'. A trap property is the negation of the original desired property, such that counterexamples can be generated for a non-error test model. The test case generation process is summarised as three steps: 1) a given test purpose is encoded into a trap property; 2) the model checker checks the given model against the trap property, and generates counterexamples; 3) test cases can be retrieved from the counterexamples. Test coverage can be achieved by repeating such process for each test purpose for the given model. For instance,
suppose the test criterion is state coverage for a machine $M$ with states $s_1, s_2, s_3, s_4$. this requires four test purposes where each corresponds to state $s_i$ is reachable. By encoding the test purposes into trap properties, the model checker will search for counterexamples for each test purpose.

5.2 Framework Overview

In this section, we will elaborate our proposed integrated framework for BPEL verification and test case generation, shown in Fig 5.2. The framework provides three functions: verification of BPEL models, test cases generation for BPEL models, and test cases generation for WSDL models. On one hand, since BPEL is an orchestration language, the BPEL based test cases should be at the level of integration testing, which check whether the composed web services conforms to functional requirement. On the other hand, since WSDL is an interface language, the WSDL based test cases should be at the level of unit testing, which check the message types and responses of individual operations in remote web services.

![Diagram](image)

Figure 5.2: Framework architecture

In the verification phase, the user can check the correctness of the design BPEL models. If there is any outputted counterexample, the user can modify the BPEL models and repeat the checking until no counterexample is produced. In the testing phase, the user can choose the verified BPEL models as test models to derive test cases, where the test cases can be automatically retrieved from counterexamples. Furthermore, the
user can choose WSDL models to generate test cases. The two levels of generated test cases can be run on the common JUnit test execution engine. The test cases enable the user to input test data.

In both phases, the BPEL models are analysed and transformed into our proposed web service automata (WSA), which in turn are transformed into Promela or SMV models. Promela and SMV are the input languages of SPIN and NuSMV model checkers, respectively. In order to tackle the state space explosion problem of model checking in the memory and to speed up the checking performance, some model simplification techniques will be introduced for WSA.

A. Verification of BPEL

The user selects one or more BPEL models as the system under test (SUT), and chooses a model checker. Inside the framework, after the model transformations, the model is model-checked against our pre-defined system properties such as deadlock-free and live-lock free (see section 5.4). If the model violates the properties, the framework outputs counterexamples. Thereafter, the user can modify and refine the BPEL models based on the counterexamples. This verification process continues until no counterexample exists.

B. Test Case Generation

The user selects one or more verified BPEL models as the SUT, picks a pre-defined test coverage criterion, and chooses a model checker. Inside the framework, the selected test coverage criterion is encoded into a set of trap properties. After the model transformations, the model is model-checked against the trap properties (see section 5.5). A set of counterexamples will be generated. The transition IDs can be retrieved from the counterexamples. By the transition IDs, we can get the inputs, guards, and outputs of the corresponding transitions from WSA. Also, the message types of the inputs and outputs can be extracted from the WSDL interface. As a consequence, the test framework will produce BPEL based test cases that enable the user to input test
5.3 Model Transformation

In this section, we will first introduce how to simplify WSA for the purpose of improving the performance of model checking. Afterwards, we will briefly describe the mapping from WSA to Promela and from WSA to SMV. Finally, we will demonstrate how to enforce model checkers to explore all alternative paths, in the absence of actual input data values.

5.3.1 Model Simplification in WSA

Abstraction is important for formal analysis with model checking techniques. In WSA, the complex data type and the concrete data value of BPEL variables will be abstracted; also the BPEL predicates will be abstracted. The abstraction will not hinder the model checking but will help to speed up the model checking. To further simplify the model, those redundant transitions and states related to faults and interruptions will be removed. The model reduction can alleviate the state explosion problem inherent of model checking techniques.

A. Abstraction of BPEL Predicates

We introduce a symbolic predicate for those decision points where the no concrete value of message is available by the time of analysis. From the point of view of model verification and testing, such a decision point could be considered a symbolic predicate which may equally be true or false. For example, in the loanapproval
example, the receive activity has two guarded outgoing links, where the guards are $request.amount < 10000$ and $request.amount \geq 10000$. Here $request$ is a BPEL variable, which is assigned a value by a message from an external service. Since the actual value of $request$ is not determined by the time of static analysis, we use symbolic predicates $pred_1, pred_2$ to abstract the actual guards. Since the values of symbolic predicates would be equally true or false, the values of $pred_1, pred_2$ can be $(1, 0), (0, 1), (1, 1), (0, 0)$ where $1, 0$ denote true and false respectively. However, $pred_1, pred_2$ is not allowed to be true or false at the same time, because the guards of outgoing links in an activity should be mutual exclusive. Therefore, some logical constraints need to be added to the symbolic predicates. In the example, the logical constraint is $pred_1 \neq pred_2$.

A symbolic predicate will be introduced at the control decision points of a BPEL model, where the condition expressions are explicitly modelled. The control decision points include the condition expressions in the $transitionCondition$ attribute of a sourceLink, in the $condition$ of a while activity, and in the case and otherwise constructs of a switch activity. The predicate logical constraints can be automatically derived from the BPEL model, according to the following rules:

- For the $transitionConditions$ of sourceLinks in an activity, a symbolic predicate is introduced for each transitionCondition. These predicates should be mutual exclusive. Suppose $p_1, p_2, p_3$ are the predicates, the logical constraint should be $(p_1 \land \neg(p_2 \lor p_3)) \lor (p_2 \land \neg(p_1 \lor p_3)) \lor (p_3 \land \neg(p_1 \lor p_2))$.

- For the $condition$ of a while activity, two mutual exclusive symbolic predicates are introduced. Let $p_1, p_2$ be the predicates. The logical constraint should be $(p_1 \land \neg p_2) \lor (p_2 \land \neg p_1)$.

- For the condition expressions in the case and otherwise constructs of a switch activity, a symbolic predicate is introduced for each of them. Let $p_i$ denote a condition expression, where $p_i, 1 \leq i \leq n-1$, and $p_n$ are the condition expressions of case and otherwise, respectively. The logical constraint should be $(\neg p_1 \land p_2) \lor (\neg p_1 \lor p_2 \land p_3) \lor (\neg(p_1 \lor p_2 \lor p_3) \land p_4) \lor \ldots \lor \neg(p_{n-2} \lor p_{n-1} \lor p_n)$.
5.3 Model Transformation

Note that the above rules aim not to derive the logical constraints for all predicates in a BPEL model. A specific tool is required to compute the predicate logical relationships precisely.

B. Abstraction of BPEL Variable Types and Values

A BPEL variable can be declared as one of the three types: WSDL message type, XML Schema element, and XML Schema type. A WSDL message type and a XML element must be a complex type, while a XML Schema type can be either a simple or complex type. A complex type is a type with enclosed elements. Without consideration of data type abstraction, two problems occur: 1) If the variable is a WSDL message type, we need to search the WSDL model for the declaration of such a message type, so that such a variable can be declared as a complex type in Promela or SMV; 2) In SMV modules, for a variable with a complex type in an assignment expression or in a message sending, all the enclosed elements need to be parsed and handled one by one. Since such additional modelling heavily complicates the Promela and SMV models and slows down the model checking, we abstract the complex types into abstract types without enclosed elements. An abstract type corresponds to the BPEL variable's type name itself. For instance, if a BPEL variable is declared as a WSDL message type orderDetails, the corresponding abstract type is orderDetails. So the WSDL model does not need to include data type definitions in the mapping from BPEL to WSA.

Since the data type is abstracted, the data value also needs to be abstracted. For instance, a BPEL variable request has data type creditInformationMessage, which is a complex type with element amount. There may exist request.amount in the model, we simply abstract it into the BPEL variable itself request. In the case that a BPEL variable is assigned a value, we abstract the actual data value into defined. This abstraction will influence the predicate evaluation. With the above symbolic predicates, the abstraction of data value has no side-effect for model checking.

C. Removal of Fault and Stop Propagations
5.3 Model Transformation

As illustrated in section 4.4, the models with consideration of fault and interruptions are significantly larger than the ones with only normal scenarios. If no fault can be thrown in a BPEL model, then any fault propagation related transitions and states of compound machines can be left out. Furthermore, if no fault is thrown and no terminate activity exists in a BPEL model, then the transitions and states related either to fault or stop propagation can be left out. As a result, the model size can be drastically reduced.

5.3.2 From BPEL to WSA

The detailed mapping from BPEL to WSA has been covered in chapter 4. The WSA has two purposes: 1) to define the operational semantics for BPEL models, and 2) to clarify the BPEL verification and test case generation problem at hand. We believe it is essential to introduce the WSA as the intermediate representation between BPEL and model checkers. When the intermediate layer is not included, the complex BPEL features need to be analysed and modelled in the input languages of different model checkers. Also we need to consider how to simplify the model for each input language in order to alleviate the state explosion problem. This heavily complicates the model transformation, which may increase the possibility of error model transformations. When the intermediate layer is included, the BPEL features only need to be analysed and modelled in WSA once. As a non-hierarchical automaton model, WSA can be easily transformed to the automata-based input models of most model checkers. Therefore, this additional automaton layer not only reduces the cost of model checking BPEL, but also enables a wider choice of model checking engines.

5.3.3 From WSA to Promela

Promela is the input language of SPIN model checker. A Promela model can contain three different types of objects: processes, variables, and message channels. All processes are global objects. The scope of a variable is global if it is defined outside a process declaration and local if it is defined within a process declaration. The data type
5.3 Model Transformation

*chan* specifies message channels. The behaviour of a process is declared in a *proctype* block. The *atomic* sequence allows the enclosed statements to execute as one indivisible unit, non-interleaved with any other processes. The symbolic names used in the model can be declared either globally or locally in a *mtype* block. A user-defined type can be declared in a *typedef* block. An *init* process is required in each Promela model to start the *proctype* processes. Since Promela supports message communication via channels, it is straightforward to transform WSA to Promela. We can model the propositional operators on input events indirectly in Promela. For events with logical-AND $e_1 \land e_2$, it is modelled as $e_1; e_2$. For events with logical-OR $e_1 | e_2$, an *if* construct can be used. For events with logical-NOT $\neg e_2$, it can be modelled as $e_1; \text{empty}(\text{channel}_{e_2})$ where the function $\text{empty}(\text{channel}_{e_2})$ checks whether the queue for $e_2$ is empty.

In our transformation, each WSA machine corresponds to a *proctype* process. The incoming queues of each machine are declared as a set of global *chan* types. The incoming and outgoing events are denoted as $\text{channel}_i ? e, \text{channel}_j! e$ respectively, where $\text{channel}_i, \text{channel}_j$ correspond to the incoming queues of machine $M_i, M_j$ respectively. Each process consists of two parts: 1) the variable declaration part, and 2) the behavioural modelling part. In the first part, the states, transition IDs (e.g. $t_i$), and local variables of a machine are declared. In the second part, the transition relations are modelled, which are enclosed in a *do* loop. The *if* construct enables the selection of an enabled transition, or the selection of a true-value guard. In a transition, if 1) the incoming events are available and the propositional logic of the events are true, and 2) the guard is evaluated to true, then an atomic unit will be executed: the transition is enabled, the action is taken, and the machine moves to the next state.

An example is shown in Fig 5.3. On the left is the Promela code for the *process* machine of the loan approval example. A global *mtype* declares the symbolic names of states, messages, and BPEL variable types that need to be used in the model. The *process* machine has two incoming queues, where the *adminQ* receives start or stop messages from other machines, and the *doneQ* receives the flow machine’s done message. It can send a start message to the *adminQ* of the flow machine. On the right
5.3 Model Transformation

is a code fragment to show the case where the guards are explicitly declared.

5.3.4 From WSA to SMV

SMV is the input language of NuSMV. A SMV model is composed of a set of modules with the keyword MODULE. The module corresponds to a state machine that has two sections VAR and ASSIGN, where the VAR section declares the static part and the ASSIGN section declares the behavioural part. We call such a module an SMV machine for convenience. A statement FAIRNESS running needs to be included to ensure each module corresponding to a state machine can execute fairly. A user-defined data type can be also declared as a module. A main module is required in each SMV model to start the SMV machines. Since SMV language has no support for channels, the input queues of a WSA machine need to be modelled explicitly. When the queue is implemented with FIFO manipulation, the state space will increase dramatically. Thus, we model a queue with a simplified structure which holds only one message. The SMV model supports the propositional operators on input events directly, by using & | for AND, OR, NOT respectively.

Figure 5.3: An example of Promela model
In our transformation, each WSA machine corresponds to a SMV module. The different queue types are defined as SMV modules. The machine input queues are declared as one of the defined queue types. In a SMV machine, the states, transition IDs, incoming queues, input events, output events, and local variables of a machine are declared in the VAR section. The transition relations are captured in the ASSIGN section. The $\text{next}(x)$ denotes the next value of the $x$ variable, where $x$ can correspond to any variables declared in the VAR section. A transition is enabled by evaluating the Boolean value of $\text{next}(\text{state}) = s_i \& \text{next}(e) \& \text{pred}_i$, where $s_i$ is the current state, $\text{next}(e)$ is the incoming event, and the $\text{pred}_i$ is the guard. When the transition is enabled, the actions such as assigning values to variables and sending messages can be taken. The machine moves from one state to another by evaluating the Boolean value of $\text{state} = s_i \& t_i$, where $s_i$ is the current state and $t_i$ is the enabled transition. An incoming event becomes true when its queue has been set to true, indicating that the event has occurred. The parameter of an incoming event, if any, can be read from its queue when the incoming event has become true. On the left of Fig 5.4, the SMV code for the process machine of the loan approval example.

In a SMV machine, a message with a parameter can be received in two steps: 1) the event is set to true when its queue has become true, 2) the parameter value is read from its queue when the event has become true. The right of Fig 5.4 shows a fragment of a SMV machine. When the machine receives a message $\text{msg(request)}$, the event corresponds to an event name $\text{msg.request}$ and a queue $\text{msg.request.Q}$, and the parameter corresponds to a parameter name $\text{request}$ and a queue $\text{request.Q}$. After the $\text{next}(\text{msg.request})$ has been set to true, the $\text{request.value}$ can be read from the $\text{next}(\text{request.Q.data.value})$. The machine sends a message with parameter to a partner machine following similar steps, it first sets the partner’s event queue to true, and then send the parameter value to the parameter queue.
5.4 BPEL Verification

In model checking, two kinds of system properties can be verified: general properties and model-specific properties. We do not consider model-specific properties because such properties are user-defined, so the user needs to have knowledge of temporal logic.

Figure 5.4: An example of SMV model
5.4 BPEL Verification

5.4.1 Pre-checking Illegal Cross-boundary Links

In our framework, the illegal cross boundary links can be pre-checked in the first stage of analysing BPEL, so that the errors can be identified in the earliest stage. According to section 12.5 of BPEL1.1 [10], a link is said cross the boundary of a syntactic construct if the source (resp. target) activity for the link is nested within the construct but the target (resp. source) activity is not. A link must not cross the boundary of a while activity, a serialisation scope, an eventHandler, or a compensationHandler. Also, a link that crosses a faultHandler boundary must be outbound.

This checking can be done by getting the machine sequences for the flow links from the internal data exchange model (see section 4.2.3). Let \( l \) be defined in machine \( M_1 \) and used in machine \( M_4 \). Suppose the machine sequence for \( l \) is \( \langle M_1, M_2, M_3, M_4 \rangle \), this indicates that \( M_1, M_4 \) must not be the machines at the same level, if \( M_1, M_4 \) are same-level-machines then \( M_1 \) will send the \( msg(l) \) to \( M_4 \) directly. Thus, one of \( M_2, M_3 \) or both of them may be the parent nodes of \( M_1 \), or be the parent nodes of \( M_4 \). In either case, the link is cross boundary. Therefore, in the machine sequence for link \( l \), if there exists a sub-sequence with one or more machines between the machine that defines \( l \) and the machine that uses \( l \), then it indicates that \( l \) is cross boundary.

In the case that the above sub-sequence contains a machine corresponding to the while, serialisation scope, eventHandler, or compensationHandler activity, then the cross boundary link is illegal. In the case that the above sub-sequence contains a machine \( M \) corresponding to the faultHandler activity, and \( M \) is the parent node of the machine that uses \( l \), then such cross boundary link is illegal and must be outbound. In either case, an error will arise in the machine sequence.

5.4.2 Model Checking BPEL General Properties

In our framework, we embedded a set of model-independent properties, so that they can be checked automatically during the model verification phase.

A. Deadlocks
5.4 BPEL Verification

A deadlock is defined as two machines waiting for input from each other; the system is unable to perform a transition. A state $s$ is deadlock when $s$ has no next state (i.e. no outgoing transition) and $s$ is not a final state. A machine is deadlock-free if it has no deadlock state. A machine is terminating if one of its final states is finally reached. Since a machine with deadlock states cannot terminate as it cannot reach any of its final states, this indicates that a terminating machine is deadlock-free. So we can use such a machine terminating property to indicate a machine is deadlock-free. The BPEL process is said to be deadlock-free if all related machines are terminating, which can be described as $\{\diamond s_{f_1}\}$ in LTL, and $\{EF s_{f_1}\}$ in CTL, where $s_{f_1} = s_{f_1} \lor \ldots \in s_{f_{in}}$ is one of the final states of machine $M_i$.

SPIN checks the deadlock and livelock by default, so no explicit temporal logic formula or configuration setting is required. In NuSMV, CTL formula can be used for deadlock and livelock detection. Let $M$ be a machine with final states $s_{f_1}, s_{f_2}$, the deadlock-free formula for this machine is: $\text{SPECEF}(M.\text{state} = s_{f_1} \lor M.\text{state} = s_{f_2})$. Alternatively, the NuSMV 2.4 provides the check fsm command to check the transition relation for totality. If the transition relation is not total then a potential deadlock state is shown.

B. Unreachable Activities

In the context of BPEL, we are only interested in checking the reachability of BPEL activities. A BPEL activity is reachable if it can eventually be executed, which means that the initial state of its corresponding machine can be reached. In WSA, we have two kinds of supporting machines for BPEL activities: linkWrapper and compensateManager machines. They need not be checked for the reachability of a BPEL activity. On one hand, an activity with target links can execute only when its join-condition is satisfied by the incoming links. In WSA, all the incoming and outgoing links of an activity are handled in a linkWrapper machine. In the linkWrapper machine, if the join-condition is evaluated to true, then it starts the core machine. Therefore, to check whether an activity is executable, we only need to check whether its core machine can
be started, so that its initial state can be reached. On the other hand, since compensateManager machine is started to handle compensateHandler invocation, it will not influence the reachability of any BPEL activity.

The reachability of all BPEL activities can be described as \( \{() s_{0_i}\} \) in LTL and \( \{EFs_{0_i}\} \) in CTL, where \( s_{0_i} \) is the initial state of machine \( M_i \), and \( M_i \) is not a linkWrapper or compensateManager machine. This property is subsumed by deadlock-free checking.

C. Uninstantiated Data

An assertion for a variable \( x \) enforces that as long as \( x \) is used, \( x \) needs to be previously defined. An assertion will be added for BPEL variables and target links. A false assertion of a BPEL variable indicates that the corresponding activity \( B \) that defines the variable is missing, \( B \) is not reachable, or \( B \) is deadlock. A false assertion of a target link indicates that a corresponding source link is missing, or there exist cyclic link dependencies. A cyclic link dependency can cause an unreachable activity. In [11], they point out that the assertions can also detect the interaction between BPEL control dependencies and data dependencies. An example is that a switch activity that encloses activities \( A \) and \( B \), where \( B \) needs to use a variable \( x \) defined by \( A \). By adding an assertion for \( x \) in \( B \), the assertion can never be true such that the bad interaction between control and data dependencies can be detected.

It can be observed that checking BPEL data dependencies for errors is very useful. In Promela, an \( assert(x! = \text{nil}) \) for variable \( x \) needs to be added after receiving message \( msg(x) \), so that the SPIN can automatically check the assertions. If the assertion is not true, SPIN will report errors. In NuSMV, an CTL formula based assertion can be appended to a module: \( SPECAF(x! = \text{null}) \).

D. Illegal Simultaneously-enabled Transitions

In section 11.4 of BPEL 1.1 [10], the semantics is undefined when a process in which two or more receive activities for the same partnerLink, portType, operation
and correlation sets may be simultaneously enabled. This constraint is also applied to the onMessage clause in a pick or eventHandler activity. There can never be two simultaneously running activities that write the same data. This problem has been pointed out in [46]. For multiple concurrently running machines, in their composed machine, there may exist a state with multiple simultaneously enabled outgoing transitions. Fig 5.5 shows an example. Let machine $M_0$ contains a transition $t_0$ from $s_{02}$ to $s_{03}$, and machine $M_{01}$ contains a transition $t_1$ from $s_{12}$ to $s_{13}$. When they run concurrently, the corresponding composed machine has a state $(s_{02}, s_{03})$ with simultaneously enabled transitions $t_0$ and $t_1$. The simultaneously enabled transitions can be categorized into three cases: 1) $t_0, t_1$ have no receive event but the guards that can simultaneously true; 2) $t_0, t_1$ have different receive event lists $e_1$ and $e_2$; 3) $t_0, t_1$ have the same receive event list $e$. From definition 3 of section 3, the state $(s_{02}, s_{03})$ of case (1)(3) have non-deterministic transitions. In case (2), if both $?e_1, ?e_2$ are available, then the machine consumes one of them randomly. According to the above BPEL semantics, it allows case (1)(2) but not case (3). So we need to define a property for case (3) to detect such illegal transitions, which can be described as $(\neg(t_{i,1} \land t_{i,2} \land \ldots t_{i,n})$ in LTL, and $AG-(t_{i,1} \land t_{i,2} \land \ldots t_{i,n})$ in CTL, where $t_{i,1}, t_{i,2}, \ldots, t_{i,n}$ is the set of outgoing events of a state $s_i$ that have the same receive event list.

\[\text{Figure 5.5: Simultaneously enabled transitions}\]

\[E. \text{Illegal Inter-process Interactions}\]

In the inter-process interactions, the causal relationships between incoming and outgoing messages must be maintained. One solution is to choose multiple BPEL models as the SUT, and check the composite machines against deadlock. A lightweight
solution is to pre-check the possible conflicts before composition, using the anti-patterns proposed in section 3.4.2.

5.5 BPEL Test Case Generation

We apply the structural test coverage criteria to multiple machines. State and transition coverages are used for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. The variables and flow-links declared in BPEL models will be considered in data flow testing. We are interested in testing the whole BPEL model. According to the machine hierarchy of the previous section, a test case should start and end with the BPEL process machine $M_{proc}$. Following the propagation of the start and done messages between parent machines and child machines, we may assume without loss of generality that in the machine hierarchical graph, every machine is reachable from the BPEL process machine, and that the BPEL process machine is reachable from every machine.

In the following definitions, let a BPEL process $P$ associated with a set of machines $\{M_1, \ldots, M_n, M_{proc}\}$.

Definition 24. A test case (or test path) of a BPEL process $P$ starts from the initial state of $M_{proc}$, and ends at one of the final states of $M_{proc}$.

Definition 25. A test suite covers a state $s$ if there is at least one test case in the test suite that executes $s$. A test suite is said to achieve state coverage of a BPEL process $P$ if it covers each state $s \in S_{M_i}, M_i \in \{M_1, \ldots, M_n, M_{proc}\}$.

Definition 26. A test suite covers a transition $t$ if there is at least one test case in the test suite that executes $t$. A test suite is said to achieve transition coverage of a BPEL process $P$ if it covers each transition $t \in T_{M_i}, M_i \in \{M_1, \ldots, M_n, M_{proc}\}$.

Data flow testing is interesting because it stimulates the sequences of operations which define and subsequently use variable values. It is an effective systematic method for exposing faults. For the du-path coverage, we adopt the definition from [51]. Let
5.5 BPEL Test Case Generation

Let $x$ be a variable. A du-pair of $x$ is a transition pair $(t_i, t_j)$ where $t_i$ is with $df(x)$ and $t_j$ is with $us(x)$. A *def-clear* path with respect to $x$ is a transition sequence $(t_1, t_2, ..., t_n)$ where there is no $df(x)$ in any of the transitions $t_2, t_3, ..., t_{n-1}$. A *data flow* (or *du-path*) of $x$ is a transition sequence such that $(t_i, t_j)$ is a du-pair and there is a def-clear path from $t_i$ to $t_j$ for $x$.

Note that we are only interested in the BPEL variables explicitly declared in a BPEL model (denoted as $V_{bpel}$), and the data dependency between BPEL activities. Therefore, in our all-du-path coverage criterion, we only consider du-pairs $\{(t_i, t_j)|t_i \in M_i, t_j \in M_j, i \neq j\}$ with respect to $v$ where $v \in V_{bpel}$.

**Definition 27.** A test suite covers a du-path $ph$ of a variable $v \in V_{bpel}$ if there is at least one test case in the test suite that executes $ph$. A test suite is said to achieve **all-du-path coverage** of a BPEL process $P$ if it covers each du-path of each $v \in V_{bpel}$.

### 5.5.1 Test Coverage Criteria in Trap Properties

**Test Coverage Criteria in Trap Properties.** Now we can encode the test coverage criteria into CTL and LTL temporal logic. [3:3] gives a detailed study of encoding various structural test coverage criteria into CTL. Based on their work, the negation of state, transition, and all-du-path coverage criteria are encoded into the CTL formulas as follows. Here $M$ is a web service automaton.

- $\{\neg EF(s_i \land EF s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.
- $\{\neg EF(t_i \land EF s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.
- $\{\neg EF(t_i \land EX E [-d(v)U(t_j \land EF s_f)]\}$ where $v \in V_M, t_i \in d(v), t_j \in u(v), s_f \in S_{fMproc}$.

Suppose $M$ is a web service automaton. The negation of state, transition, and all-du-path coverage criteria are encoded into the LTL formulas as follows.

- $\{\neg \Diamond (s_i \land \Diamond s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.
5.5 BPEL Test Case Generation

- \{¬\diamond(t_i \land \diamond_s f)\} where \(t_i \in T_M, s_f \in S_{f_{M_{proc}}}\).

- \{¬\diamond(t_i \land X(¬d(v)Ut_j) \land \diamond_s f)\} where \(v \in V_M, t_i \in d(v), t_j \in u(v), s_f \in S_{f_{M_{proc}}}\).

In SPIN model checker, each LTL formula needs to be converted into a Buchi Automaton enclosed in a never claim. Since a never claim is to negate the enclosed Buchi automata, the input LTL formula for SPIN model checker should be the original property (the non-negated one). In Promela, we attach a never claim, corresponding to the user selected test coverage criterion, to the Promela model generated from WSA, and use \texttt{#define} to declare the elements required in the never claim. Fig 5.6 (a) shows a never claim for covering a state and a final state of the process machine. \(\diamond(p \land \diamond q)\) is the LTL formula for the enclosed Buchi Automaton. \(p\) denotes a state of a machine and \(q\) denotes a final state of the process machine. According to the above LTL state coverage, a set of the negations of such LTL formulas can provide state coverage of the BPEL model. Since SPIN can only verify one property in a run, SPIN needs to run \(n\) times for \(n\) pairs of \((p, q)\).

NuSMV model checker accepts either LTL or CTL formulas, and a formula starts with the keyword \textit{SPEC} in SMV models. Fig 5.6 (b) shows a CTL formula declaration for covering a state and a final state of the process machine. According to the above CTL state coverage, a set of such CTL formulas can provide state coverage of the BPEL model. Since NuSMV can verify more than a single property in a run, SMV only needs to run once for a set of CTL formulas.

\begin{verbatim}
#define p (loanapproval_flow_receive.state == s2)
#define q (loanapproval.state == s3 || loanapproval.state == s1)
(a) never ( "<>(p && q)" )
    TO_init: if
        : (p) && (q): goto accept_all
        : (1): goto TO_init
    fi
    accept_all:
        skip
    }
(b) SPEC !EF loanapproval_flow_receive.state = s2,
& EF loanapproval.state = s3) loanapproval.state=s1)
\end{verbatim}

Figure 5.6: An example of state coverage
5.5.2 Symbolic Test Case Generation

In a state machine with symbolic predicates (see section 5.3), in order to enforce a model checker to explore alternative paths, the predicates of different paths need to be true alternatively. This requires the values of symbolic predicates to be equally true or false. We apply Gray code (e.g. [8]) to compute the combinations of predicates, where two successive values differ in only one digit. With all the combinations, the model checkers can explore all the paths of a model. For two symbolic predicates \( \text{pred}_1, \text{pred}_2 \), the two-bit Gray code matrix is \((\text{pred}_1, \text{pred}_2) : (0, 0), (0, 1), (1, 1), (1, 0)\). Here \( 1, 0 \) denote Boolean true and false, respectively. After all the possible combinations of predicates have been calculated, we can apply the logical constraints introduced in section 5.3 to remove illegal combinations. In the case that \( \text{pred}_1, \text{pred}_2 \) are mutually exclusive, the combinations \((0, 0), (1, 1)\) can be removed.

The corresponding Promela code is shown below on the left of Fig 5.7. First, each symbolic predicate \( \text{pred}_i \) is declared as a global Boolean variable. Second, a Gray code matrix is constructed based on the declared symbolic predicates, the logical constraints on the predicates, if any, can be applied to the Gray code matrix. Third, two processes will be inserted into the Promela model: a runner process that assigns predicate values based on the above Gray code matrix, and a chooser process that chooses the current combination of predicates. The chooser process will be started by the init process.

```promela
MODULE runner
VAR
\text{pred}_1 : boolean;
\text{pred}_2 : boolean;
\text{true}\text{false}
ASSIGN
next(\text{pred}_1) = case
i = 1 : \text{true};
\text{false};
\text{true};
\text{false};
:: i = 2 : \text{true};
\text{false};
\text{true};
\text{false};
end;
next(\text{pred}_2) = case
i = 1 : \text{true};
\text{false};
\text{true};
\text{false};
:: i = 2 : \text{true};
\text{false};
\text{true};
\text{false};
end;
FAIRNESS running
```

Figure 5.7: An example of predicate handling

Similarly, the SMV code is shown on the right of Fig 5.7. A runner module declares
the symbolic predicates. After the Gray code matrix has been constructed based on the declared symbolic predicates, the logical constraints on the predicates, if any, can be applied to the Gray code matrix. Thereafter, the ASSIGN section will be inserted into the runner module, so that the runner can choose the current combination of predicates. The runner process will be started by the main module.

5.5.3 From Counterexamples to Test Cases

The test cases retrieved from counterexamples are called BPEL based test cases. The BPEL based test cases focus on the sequencing of invocations of the provided services. The transition names \((t_i)\) are modelled explicitly in Promela and SMV models, so that a transition name list can be retrieved from the generated counterexample. A test case can be derived from the transition name list, by extracting the corresponding transition input events, guards, actions, and output events from the associated WSA model and getting the message types of the inputs and outputs from the WSDL interface. A test case consists of a set of execution paths of the BPEL. The BPEL test generation is a kind of white-box testing that analyses the internal process definition.

5.6 WSDL Test Case Generation

The test cases generated from WSDL cover validation of single operations. The execution of test cases will invoke remote operations provided by the services. This kind of test case checks whether the implemented service operations conform to the published service modelled in WSDL. Note that the WSDL test generation is a kind of black-box testing that analyzes the service interface definition.

5.7 Summary

In this chapter, we presented an integrated verification and test framework for BPEL using model checking. The framework allows the user to choose the service under test (SUT), so that one or more BPEL models can be verified and be used to generate test
cases. For verification, both the illegal interactions of an intra-process or inter-processes can be verified. For test case generation, one or multiple BPEL processes can be tested. In the verification phase, a set of errors is automatically checked, including illegal cross-boundary links, deadlocks and livelocks, unreachable activities, non-instantiate data, and illegal simultaneously-enabled transitions. In the testing phase, to test a BPEL program thoroughly, we need to cover different execution scenarios. So we apply the conventional test coverage criteria such as state, transition, and du-path coverage to thoroughly test the BPEL program. The generated BPEL based test cases can check the conformance of web service behaviour. The user can also test the conformance of the functions of the web services with the published WSDL interfaces. In the next chapter, we will introduce the tool architecture of our test framework, and demonstrate our approach via case studies.
Chapter 6

Case Studies and Tool Support

In the previous chapter, we presented our model-checking based test framework that can test single or multiple BPEL processes, by verifying general model properties, and generating test cases based on structural coverage criteria. In this chapter, we will use classic BPEL case studies to illustrate the modeling of BPEL processes in web service automata, and to evaluate the effectiveness of our test framework. Then, we will provide a brief description of the tool support.

6.1 Case Studies

Our test framework supports verification of general properties and test case generation for BPEL processes. In the verification phase, firstly the pre-checking checks that there are no illegal cross boundary links in the SUT BPEL process. Secondly, by running the model checker, three general properties will be checked: deadlock-freedom, un-instantiated data freedom, and illegal simultaneously-enabled transitions freedom. Since the checkings for general properties in the verification phase are done automatically, we will focus on test case generation in the case studies.

Every web service shall have its own business logics, so a web service should be associated with a behavioral model in addition to the web service interface description. BPEL is a well known language for web service orchestration, and it is rich enough to
6.1 Case Studies

describe the behaviours of single services. We suppose a web service is associated with two models: a BPEL process as the behavioural model, and a WSDL description as the interface. In BPEL standard [10], four examples are provided: loan approval service, ordering service, shipping service, and auction service. The loan approval service covers flow activities with links. The ordering service covers sequence and flow activities with cross boundary links. The shipping service covers sequence and switch activities. The auction service covers flow and sequence activities. We will use the loan approval service as the main running example to illustrate how to model the BPEL processes in our web service automata, and show the difference between choosing single and multiple BPEL processes as the service under test (SUT). We will illustrate the ordering service and shipping service in brief.

As covered in section 4.1.1, the hierarchical BPEL activities can be modeled as the relationships between parent and child machines. For the sake of simplicity, in this chapter, we use tree diagrams to represent both. For BPEL activities, a tree node denotes an activity, and a tree edge denotes the enclosure of activities. For WSA machines, a tree node denotes a machine, and a tree edge denotes the parent and child relationship of machines.

6.1.1 Loan Approval

The loan approval example includes four web services: customer, approval, assessor, and approver. The approval service acts as the orchestration service to interact with other services. The BPEL standard provides a BPEL process approval for the loan approval example. The business scenarios of the approval service is illustrated in part (1) of Fig 6.1. The approval service starts by receiving a request from the customer. If the request is less than 10000, it forwards the requests to the assessor; otherwise it forwards the request to the approver. The assessor evaluates the request and returns a risk result. The approver evaluates the request and returns an approved result. When the returned risk from the assessor is low then the approval service replies the customer by a granted loan; otherwise if the returned risk from the assessor is high,
the approval service calls the approver. After getting response from the approver, the approval service replies the customer of the loan decision. In addition to the approval service, the assessor and approver must have some business logics to decide whether an incoming request has high risk or low risk, and whether a high risk request can be granted. We add BPEL processes for the assessor and approver services. The business scenarios are shown in parts (2)(3) of Fig 6.1.

Figure 6.1: The business scenarios of the loan approval example

Fig 6.2 shows the hierarchical BPEL activities of the approval process. On the left hand side, the approval service uses a flow activity with links to control the enclosed concurrently running activities. On the right hand side, the assessor or approver service has a sequence activity to control the execution sequence of the enclosed activities, and uses a switch activity to decide whether to assign a high risk (resp. yes info) or a low risk (resp. no info) as the result.

Figure 6.2: The BPEL activity hierarchy of approval, assessor, and approver

To model the BPEL activities in WSA machines, basically each BPEL activity is associated with a machine. An activity with source or target flow links is mapped to a
6.1 Case Studies

linkWrapper machine and a core machine. The parent and child relationships of WSA machines is shown in Fig 6.3.

Figure 6.3: The machine relationships of approval, assessor, and approver services

A. Single BPEL Process as the SUT

When the approval BPEL process is selected as the SUT, the customer, assessor, and approver become testers. According to rule 1 of the section 4.2.3, the machine du-pairs and machine exchange sequences can be derived for BPEL variables and flow links. The graphical view of the internal data exchange model for the approval process is shown in the middle of Fig 6.4.

Figure 6.4: Global data exchange model

- For variable req, the machine du-pairs are \((M_1, M_4), (M_1, M_8)\). The machine sequences are \((M_1, M_2, M_3, M_4)\) and \((M_1, M_2, M_7, M_8)\).
6.1 Case Studies

- For variable risk, the machine du-pair is \((M_4, M_3)\). The machine sequence is \(\langle M_4, M_3 \rangle\).

- For variable info, the machine du-pair are \((M_6, M_{10})\), \((M_8, M_{10})\). The machine sequences are \(\langle M_6, M_7, M_9, M_{10} \rangle\) and \(\langle M_6, M_5, M_9, M_{10} \rangle\).

- For links \(l_1, l_2, l_3, l_4, l_5, l_6\), the machine du-pairs are the same as the machine sequences, which are \(\langle M_2, M_3 \rangle\), \(\langle M_2, M_7 \rangle\), \(\langle M_3, M_7 \rangle\), \(\langle M_3, M_5 \rangle\), \(\langle M_5, M_9 \rangle\), \(\langle M_7, M_9 \rangle\), respectively.

According to the du-pairs of BPEL variables and links, the assertions for BPEL variables flow links will be added to transitions where the variables are used.

The machines for the approval process is shown in Fig 6.5. In the diagram, we use LW as short for linkWrapper. In the receive, invokeAssessor, invokeApprover, and reply machines, when it interacts with a machine of the external BPEL process and that process is not a part of the SUT, then the partner machine is simply named tester. In WSA machines, four symbolic predicates are introduced: the receive linkWrapper has predicates \(pred_1, pred_2\) where \(pred_1 \neq pred_2\), and the invokeAssessor linkWrapper has predicates \(pred_3, pred_4\) where \(pred_3 \neq pred_4\).

When the assessor process is chosen as the SUT, the graphical view of the internal data exchange model for the approval process is shown on the left of Fig 6.4. For \(req\): \(\langle M_1, M_2 \rangle\). For \(risk\): \(\langle M_3, M_2, M_5 \rangle\) and \(\langle M_4, M_2, M_5 \rangle\). According to the internal-data-exchange machine sequences and the mapping rules of BPEL to WSA, the assessor process can be transformed into the state machines in Fig 6.6. Similarly, the machine can be derived in the same way when the approver process is chosen as SUT. Since all the BPEL features are analysed and captured in WSA, the transformation from WSA to Promela or SMV are straight away.

Next, we can enter the stage of test case generation. Fig 6.7 shows three scenarios when the approval, assessor, and approver are selected as the SUT respectively. The edge with arrow from a tester to the SUT denotes a test input to the SUT, and the edge with arrow from the SUT to a tester denotes a test output from SUT. The test
framework can generate a set of test cases based on the selected SUT, where each test case corresponds to an execution scenario of a BPEL process. The message types of the test inputs and test outputs of a SUT are based on the message type declared in WSDL. In a test case, each test output from SUT is asserted to be not null, so that a test case fails if not an assertion is violated.

Since a test case covers a sequence of BPEL activities, in the following, we use a sequence of BPEL activities to illustrate a test case. First, when the approval BPEL process is selected as the SUT, the test inputs to the SUT would be req, risk, app, and the test outputs from the SUT would be req, app. When choosing a control-flow based test coverage criterion, three test cases will be generated:

Figure 6.5: The machines for the approval BPEL model
6.1 Case Studies

Figure 6.6: The machines for the assessor or approver BPEL model

Figure 6.7: A single BPEL process as the SUT

- \( tc1 \) : <receive, invokeAssessor, assign, reply>. The allowed data ranges for test inputs \( req, risk \) are \( req < 1000, risk = low \), respectively.

- \( tc2 \) : <receive, invokeAssessor, invokeApprover, reply>. The allowed data ranges are \( req < 10000, risk = high \), and \( info = yes \) or \( info = no \).

- \( tc3 \) : <receive, invokeApprover, reply>. The allowed data ranges are \( req >= 10000 \), and \( info = yes \) or \( info = no \).

For example, the test input and output sequence of test case \( tc1 \) is: 1) the tester customer inputs a \( req \) to the approval, 2) the approval outputs the \( req \) to the tester assessor, 3) the tester assessor inputs a \( risk \) to the approval, 4) the approval outputs an \( info \) to the customer tester.

When choosing the du-path test coverage criterion for BPEL variables, the paths
for each BPEL variable are as follows. For req: \(<\text{receive, invokeAssessor, assign, reply}>\), \(<\text{receive, invokeAssessor, invokeApprover, reply}>\), and \(<\text{receive, invokeApprover, reply}>\); for risk: \(<\text{receive, invokeAssessor, assign, reply}>\) and \(<\text{receive, invokeAssessor, invokeApprover, reply}>\); for info: \(<\text{receive, invokeAssessor, invokeApprover, reply}>\) and \(<\text{receive, invokeApprover, reply}>\). After merging these paths, the generated test cases are still the tc1, tc2, tc3 as above.

B. Multiple BPEL Processes as the SUT

Since the internal-data-exchange model of a BPEL process is fixed, when more than one BPEL process is selected as the SUT, the state machines associated with approval, assessor, and approver services are similar to the ones in Fig 6.5 and Fig 6.6. The difference is that if an activity A interacts with an external service’s activity B and the external service is a part of SUT, then in A its partner machine is named B (instead of naming it as tester). For instance, if approval and approver services are selected as SUT, then the invokeApprover machine of approval service will interact with the receive and reply machines of the approver service. In the testing phase, Fig 6.8 shows an example when the approval, assessor, and approver are selected as SUT.

![Diagram](image)

Figure 6.8: Multiple BPEL processes as the SUT

There exists a test input req to SUT and a test output info from SUT. The interactions between approval, assessor, and approver are internal and not observable by the tester. Let S, L, A denote the shorthands for assessor, approval, and approver services. When choosing a control-flow based test coverage criterion, five test cases will be generated:

- tc1: \(<L.\text{receive}, L.\text{invokeAssessor}, S.\text{receive}, S.\text{switch}, S.\text{assign1}, S.\text{reply}.
L.\text{invokeAssessor}, L.\text{assign}, L.\text{reply}>\).
6.1 Case Studies


When choosing the du-path test coverage criterion for BPEL variables, the req is defined by the tester and used by S.switch; the risk is defined by S.assign1 or S.assign3, and used in L.invokeAssessor; the info is defined by L.assign, A.assign1 or A.assign2, and it is used by L.reply. After merging the test pathes for req, risk, app, the generated test cases are the same as above tc1, tc2, tc3, tc4, tc5.

6.1.2 Purchase Order

The purchase order example includes five web services: ordering, purchasing, scheduling, invoicing, and shipping. The BPEL standard provides the ordering BPEL process. We will consider the case when the ordering process is selected as the SUT. The ordering service acts as the orchestration service, and its business scenarios are shown in Fig 6.9. The ordering service starts by receiving a purchase order request PO from the purchasing service. Then, the process executes three sequences concurrently. The first sequence assigns shippingRequest with PO and sends shippingRequest to the shipping service, and then it gets the shipping price shippingInfo and the shippingSchedule from the shipping service. The second sequence sends PO and shippinginfo to the invoicing service for the order price and shipping price calculation, and then it gets the
invoice from the invoicing service. The third sequence sends PO and shippingSchedule to the scheduling service for scheduling the product and shipment. The order services ends by providing an invoice to the purchasing service.

By using the flow link ship-to-invoice, the process needs to get the shipping price shippingInfo from the shipping service before sending it to the invoicing service. By using the flow link ship-to-scheduling, the process needs to get the shippingSchedule from the shipping service by sending it to the scheduling service.

```
PO

assign211

invoke212

receive213

invoice

invoke222

receive223

invoke231

receive213

shipping

Schedule

PO

scheduling

service

Figure 6.9: The business scenarios of the ordering service
```

For the ordering process, the relationships for the BPEL activities and the WSA machines are shown on the left and right of Fig 6.10, respectively.

The machine du-pairs for BPEL variables and links are listed as follows:

- For variable PO: \( (receive1,assign211),(receive1,invoke212),(receive1,invoke231) \);
- For variable invoice: \( (receive223,reply3) \);
- For variable shippingRequest: \( (assign211,invoke212) \);
- For variable shippingInfo: \( (invoke212,invoke222) \);
- For variable shippingSchedule: \( (receive213,invoke232) \);
- For link ship-to-invoice: \( (invoke212LinkWrapper, invoke222LinkWrapper) \);
6.1 Case Studies

Figure 6.10: The BPEL activity hierarchy and machine relationships of ordering service

- For link ship-to-scheduling: (received213LinkWrapper, invoke232LinkWrapper).

According to the above machine du-pairs, the internal-data-exchange machine sequences for BPEL variables and links are listed as follows:

- For PO: <receive1,flow2,sequence21,assign211>, <receive1,flow2,sequence22, invoke221>, <receive1,flow2,sequence23,invoke231>;

- For invoice: <receive223,sequence22,flow2,reply3>;

- For shippingRequest: <assign211,invoke212LinkWrapper,invoke212>;

- For shippingInfo: <invoke212,invoke212LinkWrapper,sequence21,sequence22, invoke222LinkWrapper,invoke222>;

- For shippingSchedule: <receive213,receive213LinkWrapper,sequence21,sequence23, invoke232LinkWrapper,invoke232>;

- For link ship-to-invoice: <invoke212LinkWrapper,sequence21,sequence22, invoke222LinkWrapper>;

- For link ship-to-scheduling: <received213LinkWrapper,sequence21,sequence23, invoke232LinkWrapper>.
Accordingly to the derived machine du-pairs, the assertions can be added to the transitions where the variables or links are used. Since there is no branch in this example, no symbolic predicate will be introduced. As a consequence, one test case will be generated: \texttt{tc1: <receive1, assign211, invoke212, receive213, invoke231, invoke232, invoke211, invoke222, receive233, reply3>}.

6.1.3 Shipping Service

The shipping service interacts with a customer only, so the shipping process acts as the SUT. Its business scenarios are shown in Fig 6.11. The process starts by receiving a \texttt{shipRequest} from the customer. If such request is completed, the process returns a \texttt{shipNotice} to the customer. If such request is not completed, the process resets the \texttt{itemsShipped} and enters a loop. In the loop, the process assigns the number of items in a \texttt{shipNotice} and returns the \texttt{shipNotice} to the customer, as a partial shipment. Thereafter, the \texttt{itemShipped} is increased. The loop continues when the \texttt{itemShipped} is smaller than the request amount.

![Diagram of the business scenarios of the shipping service](image)

Figure 6.11: The business scenarios of the shipping service

The relationships for the BPEL activities and the WSA machines are shown on the left and right of Fig 6.12, respectively.
6.1 Case Studies

Figure 6.12: The BPEL activity hierarchy and machine relationships of shipping service

The machine du-pairs the internal-data-exchange machine sequences for the BPEL variables and links include:

- For variable shipRequest: the machine du-pairs are (receive1, assign211), (receive1, while222); the machine sequences are <receive1, switch2, sequence21, assign211> and <receive1, switch2, sequence22, while222>.

- For variable shipNotice: the machine du-pairs are (assign211, invoke212), (assign221, assign211), (assign22211, assign22212), (assign22211, assign22213); the machine sequences are the same as machine du-pairs.

- For variable itemShipped: the machine du-pairs are (assign221, assign22213), (assign221, while222); the machine sequences are <assign221, while222, sequence221, assign22213> and <assign221, while222, sequence221, assign22213>.

Accordingly to the derived machine du-pairs, the assertions can be added to the transitions where the variables or links are used. Four symbolic predicates are introduced. The switch2 machine has pred1, pred2 where pred1 = pred2, and the while222 machine has pred3, pred4 where pred3 = pred4. Three test cases can be generated, as follow.

- tc1: <receive1, switch2, assign211, invoke212>;
6.2 Tool Support

- tc2: `<receive1, switch2, assign221, while222, assign22211, invoke22212, assign22213, while222>`;

- tc3: `<receive1, switch2, assign221, while222>`.

In Fig 6.13, when choosing different BPEL processes as the SUT, it summarizes the corresponding state space, predicate combination number, du-pairs for BPEL variables and flow links, and the number of test paths.

<table>
<thead>
<tr>
<th>BPEL (Features)</th>
<th>Loan approval service</th>
<th>Shipping Service (sequence, switch, while)</th>
<th>Ordering Service (flow, links, sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUTs</td>
<td>Approval</td>
<td>Assessor/Approver</td>
<td>Approval, Assessor</td>
</tr>
<tr>
<td>State Space</td>
<td>283</td>
<td>91</td>
<td>296</td>
</tr>
<tr>
<td>Predicates</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>(combination)</td>
<td>(4)</td>
<td>(2)</td>
<td>(6)</td>
</tr>
<tr>
<td>Def-Use Pairs</td>
<td>11</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>Num. of test paths</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 6.13: Case Study Summary

In the work of [44], the author also uses these case studies from the BPEL standard to evaluate their SPIN based verification tool. In terms of state space, they shows that the state space of the approval process is the largest, resulting in 3516 states, due to the asynchronous semantics of concurrency. It can be observed from the case study summary that for those BPEL processes with concurrency feature, the state space of our web service automata are a lot smaller, because of its support for multiple input events.

6.2 Tool Support

Our test framework has been implemented as an Eclipse plugin, which is a component of the DBESStudio for the EU project DBE [7]. The plugin includes a BPEL test case wizard and a WSDL test case generator.

The BPEL test case wizard consists of three components: the user interface, the transformation engine, and the model checker manager. A snapshot of the user in-te...
6.2 Tool Support

Figure 6.14: A snapshot of the BPEL test wizard

The face is shown in Fig 6.14. It allows the user to choose a BPEL process as the service under test (SUT), and choose the WSDL interface model. The partner web services of the selected BPEL process can be chosen as a component of the SUT. Then a model checker can be selected to verify the general properties and to generate test cases. Inside the test framework, the XSLT transformation engine manages the mappings between models, which is using Saxon’s XSLT 2.0 processor. For pre-processing, the engine manages the mappings for the model checkers, including BPEL-to-WSA, WSA-to-PROMELA, WSA-to-SMV. For post-processing, the engine manages the mapping from counterexamples to transition sequences, and the mapping from transition sequences to test cases. The mapping to test cases is based on the transition sequence, the web service automata, and the WSDL description. The WSDL provides the test
cases with message types for the test input and test output of a service under test (SUT). The Java model checker manager takes charge of the invocation of the model checker and handling the predicate combinations. The WSDL test case generator is developed in the context of the DBE environment; it parses test cases from the WSDL interface. Then, the user can remotely invoke the functions of a published web service. Both BPEL based test cases and WSDL based test cases are in JUnit format. Hence, the test cases can be run against the JUnit execution engine directly. The user allows entering the test input (data) via a user interface.

6.3 Summary

In this chapter, we introduce the tool developed for our test framework. We use the case studies from the BPEL standard to evaluate the effectiveness of our framework. Since state space explosion is the main issue of model checking techniques, in order to improve the performance of a model-checking based tool, it is essential to reduce the model state space. We have shown by case studies that our web service automaton can alleviate the state space explosion problem when the machines are concurrently running. This fits in with the web service paradigm because web services are designed to execute independently in a distributed environment. Also, our abstraction of message type, message value, and predicates helps to speed up the model checking.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we have defined an operational semantics for the BPEL language, and presented an automatic test generation framework for BPEL models. We addressed the research question of applying model-based-testing techniques to the domain of BPEL-based web service orchestration, so that the functionality of the design model can be systematically verified.

The current formal semantics proposed for BPEL can be categorised under three branches: Process Algebra, Petri-net, and Automata. Our goal is to automate the generation and execution of test cases for composed web services. Model checking is an effective technique for automated test generation, and most mature model checkers use automata as the underlying formal model. Hence, we follow the automata branch.

Web service automaton (WSA) has been proposed with two purposes: 1) to define the operational semantics for BPEL models, and 2) to clarify the BPEL verification and test case generation problem at hand. The Boolean expression over input events is introduced for the purpose of modelling concurrency, fault propagation, and interruption features of BPEL language in a more natural way, and also as a means to reduce unnecessary state space. By analysing the control dependencies and data dependencies of BPEL activities, we demonstrated how to model the most interesting BPEL features in
WSA, such as concurrency, synchronisation, dead-path-elimination, and scope-based fault handling and compensation handling. Based on web service automata, a test framework is presented based on model checking techniques for verifying the general properties and for generating test cases of BPEL models.

Referring to the literature review chapter, we are not the first ones to propose formal semantics and apply model checking to verify BPEL processes, but we are the first to thoroughly model BPEL features in terms of automata, and to provide a test framework that can test the functionality of both single and multiple BPEL processes. Our main contributions lie in the following aspects. First, we modelled the most advanced features of BPEL thoroughly in terms of automata, a formalism which is most suitable to model checking tools. Second, we analysed the control dependencies and data dependencies of BPEL separately so that control flow testing and data flow testing can be applied. By modelling the data dependencies, we made the internal interactions of BPEL activities explicit. Third, we provided a unified message passing semantics for the interactions within a single BPEL process and between multiple BPEL processes. This enables us to test not only the behaviour of single BPEL processes, but also the compatibility of multiple BPEL processes. Fourth, in addition to verifying the general properties of BPEL processes, we applied the structural test coverage criteria to multiple machines. Accordingly, test cases can be generated for single or multiple BPEL processes, so that the functional conformance of the BPEL processes can be checked. The framework also supports test case generation for WSDL in the context of DBE environment [7], so that the interface and type conformance can be verified.

In chapter 2, we reviewed the Petri-net, Process Algebra, and automata based approaches as the formal semantics for BPEL; we discussed the current test cases generation approaches and testing techniques in the domain of web service orchestration; also, we studied the existing works with consideration of data dependencies in orchestration models; finally, we highlight the reasons to propose web service automaton as a conclusion.

In chapter 3, we provided the formal semantics of Web Service Automaton, and
identified several anti-patterns for the interaction compatibility of BPEL processes. The propositional input events of WSA can capture various BPEL features. For instance, the logical-AND operator can capture the synchronisation feature. The logical-OR operator can model the fault propagation. The logical-AND together with logical-NOT can model the higher priority events such as interruptions.

In chapter 4, we analysed the BPEL control dependencies and data dependencies. By analysing the data dependencies, the hidden internal interactions between BPEL activities are modelled as data message exchanges between machines. The data dependency analysis allows us to model the internal interactions of BPEL activities and the external interactions between BPEL processes, by means of the message passing mechanism. Thereafter, we investigated the behavioural semantics of BPEL activities in details, based on the analysis result of the control and data dependencies. In the literature, we are not aware of any automata approaches for BPEL with consideration of scoping, fault handling, event handling, and compensation handling features. Even though many Process-algebra approaches and Petri-net approaches include these features in their formalisms, due to the complication of these features, there are only a few works specifically analysing these features in depth. We demonstrated how to model these advanced features in web service automata.

In chapter 5, we introduced our test framework, which is based on model checking techniques. Since NuSMV and SPIN model checkers are already used on a regular basis for the verification of real-world applications, they are used as two alternative verification and test generation engines in our framework. The advantage of using model checking for test case generation is that only feasible test cases will be generated. However, the state space explosion problem is known to be inherent to model checking techniques. In order to alleviate the state explosion problem, we abstracted certain part of the BPEL processes in WSA. The abstraction will be applied to the complex data types and concrete data values of the BPEL variables, and to the BPEL predicates. The abstraction will not hinder the model checking but it helps to speed up the model checking. To further simplify the model, those redundant transitions and
7.1 Conclusions

states related to faults and interruptions will be removed. In our framework, the general properties of the BPEL process is verified in the first phase, and the functionality of the BPEL process can be checked in the second phased. In this two-phase testing process, the model checking tool acts as the verification and test case generation engine, respectively. In the property checking of BPEL processes, our analysis result of data dependencies can not only easily identify illegal cross-boundary links before running the model checker, but also allow the model checker to evaluate the non-instantiate data. As pointed out by Ankolekar [11], detecting non-instantiate data is useful to indicate harmful interaction between control and data dependencies. In test case generation for BPEL processes, structural test coverage criteria are applied to multiple machines. The state and transition coverages are used for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. Note that in the state and transition coverages, we do not include the supporting machines (i.e. linkWrapper and compensateManager machines) in the test case generation. This reduces the state space exploration in model checking. Because we abstract the BPEL predicates into symbolic predicates, the predicate combinations will be constructed to force the model checker to explore all the alternative paths. Since the test cases can be generated without inputting actual data, we call it symbolic test case generation. Our test framework supports two kinds of test cases. The BPEL-based test cases at the level of integration testing can check the behaviour conformance of web service interactions and various business scenarios. The WSDL-based test cases at the level of unit testing can check the interface conformance between the implementation and the WSDL of individual services.

In chapter 6, we outlined the implementation of our framework. The web service automaton has been encoded in XML format. The transformation engines have been implemented in XSLT templates, which cover the front-end mapping from BPEL to web service automata, the back-end mappings from web service automata to the Promela and SMV for the SPIN and NuSMV model checkers respectively, the mappings from transition traces to JUnit test cases, and the mapping from WSDL interface to JUnit.
7.2 Discussion

test cases. The modules developed in Java include: the GUI allowing the user to select the single or multiple BPEL SUT (service under test), the model checker, the test coverage criterion, the BPEL internal data exchange machine paths, the predicate combination and reduction, the invocation of the model checker, and the transition traces retrieval from the generated counterexamples. We used the case studies from BPEL standards [6, 10] to evaluate the effectiveness of the test framework.

7.2 Discussion

An open issue is to prove the correctness of the model transformation, i.e. the preservation of the BPEL semantics. One solution is to transforming the two kinds of models into a third common formalism, and checks the equivalence between the models in the common formalism. This solution relies on the assumption that the further transformations are correct. An example can be found in [23], Forster uses BPEL and Message Sequence Charts (MSCs) as the implementation and specification, respectively. Both MSC and BPEL are transformed into a common formalism called Finite State Processes (FSPs). By trace equivalence checking, it is possible to determine whether the BPEL processes preserve the behaviour of MSC model. In this example, the mappings from MSCs and BPEL to FSPs needs to be proved corrected. An alternative solution is to partially verify the formal model against the BPEL features. Some BPEL features are encoded into temporal logic properties, so that the model checker can be used to check the formal model against these BPEL features. Xu et al. [59] adopt this solution. Nakajima [44] suggests that the correctness of the translation can be verified via the inspection, which is aided by the modularity of the translation. The modularity comes from the multi-layer mappings from a model to another. Currently, we adopt the simplest inspection approach. This is a trade-off between the soundness of a formal approach and the simplicity of an informal approach. Also, given a set of BPEL processes that are known as correct, we can partially check the transformation correctness, by running the model checker to verify the general properties, such as deadlock-freeness.
and absence of non-instantiate data, of the target models.

7.3 Future Work

Currently, we applied the conventional state and transition test coverage criteria to multiple machines. An extension of this work is to define additional test coverage criteria which are suitable for integration testing, so that the functionality checked by less model checking time and effort. Also, currently the framework provides a GUI for the user to input test data. The question of how to automatically generate appropriate input data as parameters in the test case remains a topic for further research.

Also, our test framework enables the user to choose test coverage criteria, but does not allow the user to define specific test purposes. An interesting future work is to support on-the-fly test criteria without the need of writing temporal logic manually by the user. In [65], Zhao et al. apply SPIN model checker to verify the UML statechart model against Message Sequence Chart (MSC) model. The event sequence derived from the MSC model are encoded into a *Buchi Automaton* and listed in the *never claim* block in Promela. When checking a statechart against the event sequence, since the event sequence is negated in the never claim, if there exists a counterexample, then the statechart matches the event sequence. In the domain of web services, we can apply the similar approach in generating test cases. For a single BPEL process, the event sequence can be inputted by the user, so a test case can be generated to check the conformance of the BPEL process against the event sequence. For multiple BPEL processes, WS-CDL [36] as a choreography language provides a conversation protocol between BPEL processes. The desirable event sequence can be derived from the WS-CDL protocol, so that the conformance of the interactions of BPEL processes can be checked against the global conversation protocol.
References


REFERENCES


REFERENCES


REFERENCES


