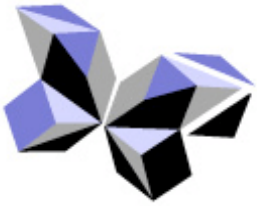

Digital Business Ecosystem

Contract n° 507953

WP N°10: DBE Test Automation

D10.2: DBE Test Automation Framework (Formal Foundations)

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: D10.2

Due date: 31/10/2006

Delivery Date: 16/02/2007

Short Description:

The purpose of this deliverable is to provide a report on the theoretical framework that underpins the Test Automation Framework, D10.2, that is now available as a plug-in for the DBEStudio. The software plug-in constitutes the main part of this deliverable. However, this report is an important complement to it.

Partners owning: UniS

Partners contributed: UniS

Made available to: public

VERSIONING

VERSION	DATE	AUTHOR, ORGANISATION
1.0	**09/02/2007**	**YONGYAN ZHENG**
2.0	**20/09/2006**	**YONGYAN ZHENG, PAUL KRAUSE**

Quality check

Internal Reviewer: John Kennedy, Intel

Internal Reviewer: Paul Malone, Waterford Institute of Technology

Executive Summary

This report describes the theorectical foundation of the WP10.2 softwre deliverable 'test-case-generator' plug-in of the DBEStudio. Some of the work presented in this report has been published in the following papers:

- Yongyan Zheng, Jiong Zhou, Paul Krause. A Model Checking based Test Generation Framework for Web Services. In proc of IEEE ITNG2007.
- Yongyan Zheng, Paul Krause. Automata Semantics and Analysis of BPEL. In proc of IEEE DEST2007.
- Yongyan Zheng, Paul Krause. Asynchronous Semantics Anti-patterns for Interacting Web Services. In proc of IEEE QSIC 2006.

Contents

CONTENTS

List of Figures

LIST OF FIGURES

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 standardized 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. A SOC offers three main functions: communication protocols, service descriptions, and service discovery.

In this work we look at the service descriptions, with 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 report provides the theoretical background to a test-case generation and execution plug-in that has been developed for the DBEStudio. The plug-inmay be downloaded from the Source-Forge

DBE Project.

1.1 Background

SOC is the emerging paradigm for the realization of heterogeneous, distributed systems, obtained from the dynamic combination of remote applications owned and operated by distinct organizations. SOC characterizes 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.

1.1.1 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.

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

- UDDI (Universal Description, Discovery and Integration) (UDD) provides a stan-

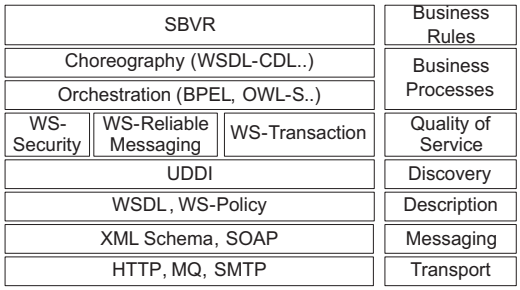

Figure 1.1: Web Service Stack

dard way for publishing and discovering information about web service.

- WSDL (Web service Definition Language) (CCMW01) defines a public interface of a web 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) (SOA03) 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 serialized 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:

- Business rule layer: SBVR (Semantics of Business Vocabulary and Business Rules Specification) (OMG06) describes the business rules using the pre-defined business vocabulary.

- Business process layer: WS-CDL (Web Services Choreography Description Language) (KBR04) provides a conversation protocol to describe the global interac-

tions of web services. BPEL (Business Process Execution Language) (ACD+03) and OWL-S (Ontology Web Language for Services) (OWL04) specify service compositions.

- 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 has a similar role as UML use case diagrams to capture the business requirement; WS-CDL is similar to UML collaboration or sequence diagrams to model the global web service interactions; BPEL has a similar role as UML statecharts to model local web service interactions and the stateful behaviour of individual web services; finally, WSDL has similarities with UML class diagrams to describe web service interfaces.

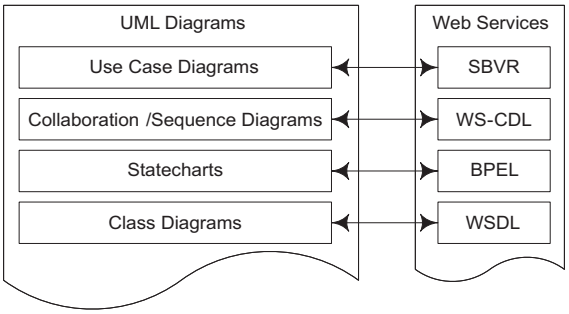

Figure 1.2: UML diagrams and web service specifications

1.1.2 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 .

1.1.2.1 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 to provide a conversation protocol for the global interaction of web services.

1.1.2.2 BPEL and OWL-S

Both BPEL and OWL-S are workflow languages for modeling 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, 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 focus on modelling the preconditions and post-conditions of the process so that the evolution of the domain can be logically inferred. It relies on ontologies to formalize 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.

1.1.2.3 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 some 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. These two relationships are defined in the *partnerRole* and *myRole* attributes of the *partnerLinkType*.

For a composite service modelled by process workflow languages such as BPEL or OWL-S. WSDL defines the data types of the service which are used directly in the process. 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

The recent years has 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 are not yet 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 categorized 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 exists two kinds of interactions of BPEL: internal and external. The external interactions between BPEL models are by message passing. The internal interactions between activities of a BPEL model are by data sharing. Most existing work ignores the internal interactions of BPEL activities.

In the literature, there is less effort 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 will be:

- 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.
- To demonstrate that the test it is feasible to apply model checking as the test generation engine for BPEL based web services.

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 categorized 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 propositional 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 the 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 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 implementation of web services conforms to the BPEL behaviour and 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 formalize BPEL semantics. This formalization enables us to reason about properties of BPEL processes such as their incorrectness 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 propositional input events of our formal model can capture most features of BPEL language (including exception handling and compensation handling), 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.

5) Demonstration that model checking is an effective means to automatically generate test cases in the web service domain.

1.6 Report 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 compatibility. Chapter 4 provides detailed analysis of BPEL features in our formal model. Chapter 5 outlines our automatic verification and test case generation framework. Chapter 6 summarizes the thesis with conclusions and potential future work.

Chapter 2

Literature Review

An extensive literature of related work on the formal semantics and verification of process models. In order to maintain the focus of this report, 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 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 based web services

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 livelocks which would cause the composition to not 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 (MM04; HS04) provide good reviews for the current verification approaches.

2.1.1 Petri Nets based approaches

A **Petri net** (Pet62) $N = (P, T, F)$ 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 is 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 (OvdAB+05).

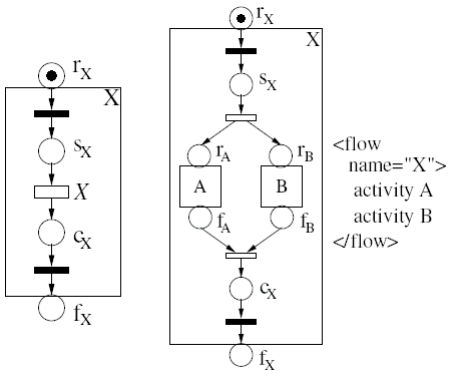

Figure 2.1: An example of Petri nets for BPEL activities

Each BPEL activity is associated with a Petri net with an input place r_i and an output place f_i. The place r_i,s_i,c_i, and f_i denotes 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 to model internal actions for checking pre-conditions or evaluating post-conditions for activities; and the other bars are used to implement composite constructs. The other types are denoted as labelled boxes. The second type is substitution transition, which is abstract representation of subnets for the enclosed BPEL activities. The third

type is activity transition to model 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 labelled 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 a **Colored Petri nets** (CP-net) (Jen97), every token has a value. CP-nets extend Petri nets with the primitives for defining data types (color) 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 coloured token in all the input places of this transition. Likewise, the output of a transition is not just a token but a specifically coloured token.

Web service algebra is proposed in (HB03) 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 (OvdAB+05; Sta05) 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 (Sta05), the tool BPEL2PN is developed to map BPEL code to Petri nets, and model checker LoLA (Sch00) is used to verify CTL temporal logic. The author of (LMSW06) extends (Sta05) to use Petri net to capture the global interactions between BPEL processes. In (OvdAB+05), 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 (YTYL05), they claim to capture both BPEL control and data dependencies in CP-nets, and CPN tools (CPN) 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 summarize 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 (YK04), they use CP-nets to model 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 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 provide data manipulation and interaction with other service partners. Instead of verifying the BPEL process, their work focus on designing a correct CP-net based model and generate BPEL skeleton process.

Petri net provide the constructions for specifying synchronization of concurrent processes. Petri net adopts interleaving semantics for concurrency, and synchronous communication. As a distributed computing paradigm, web services communicate by message passing, so it is more suitable to adopt asynchronous communication by message passing for inter web service interactions. Also, in Petri-nets, every transition is restricted to model 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 (Mil89), CSP (Hoa85), and LOTOS (BB87), and Pi-calculus (MPW92).

Pi-calculus extends CCS with mobile-ability, in which the basic entity is a process - it can be an empty process a choice between several I/O operations and their continuations; a parallel composition; a recursive definition, or a recursive invocation. I/O

operations can be input (receive) or output (send). 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 lets us reason about the composition's correctness. One can treat relevant properties by assigning behavioural types to processes. There are at least two possible ways to type processes:

- In the first case, one can type only port subsets. We can proscribe the type or shape of data that can be exchanged via two ports. This would create additional message limitations. For a two processes that can exchange any kind of message, if we type the messages (ports), we could limit the exchange.
- In the second case, one can type the entire process. When the entire process is typed, the type notion becomes a homomorphic image of the process. In many such systems, process and type are synonyms.

With process algebras semantics for web services, the general question is what information to type. Typing too little can make it impossible to verify some properties. On the other hand, typing too much creates a complexity that renders verification unusable or impractical.

In (Fer04), a two-way mapping is defined between BPEL and LOTOS, and model checking toolbox CADP (FGK+96) 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 (FUMK03), the process algebra FSP (Finite State Processe) (MK99) is used for the BPEL semantics and the model checker LTSA (MK99) as the verification engine. The web service composition specification is modelled in 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 (FUMK04; FUMK06) 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 (XLP06), 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 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 analyze Pi-calculus processes based on 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 (WPSW04), they propose a language named CDL to extend WSDL to model the behavioural of individual web services, and propose a composition language which can support both centralized or distributed orchestration. 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 . They point out that the current tools support for verifying of Pi-calculus is inmature. 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 to map to those input languages. There are other works use Pi-calculus as the formal semantics for the orchestration language (GLSC06; ML06; DMCS05).

In the above Process Algebra approaches, they consider both the core BPEL ac-

tivities and the advanced BPEL features with fault handling, compensation handling, and event handling.

2.1.3 Automata based approaches

Automata or labelled transition systems are a well-known model underlying formal specifications of systems. An automaton is a mathematical model for a finite state machine FSM. An FSM is a machine that, given an input symbols, transfers through a series of states according to a transition function. A Mealy machine is a FSM that generates an output based on its current state and an input. The transitions will include both an input and output signal. In contrast, a Moore machine is a FSM that only depends on the machine's current state, transitions having no input signal. For each Mealy machine there is an equivalent Moore machine, and vice versa. Mealy machine is commonly used, and there exist a variety of models extends Mealy machine with data, hierarchy, or time.

In (WFN04), BPEL models are mapped into deterministic finite state automata for the matchmaking of web service composition. The STSs(State Transition Systems) are used in (PTBM05) to be the BPEL formal semantics, and a tool is developed as a part of the ASTRO toolset (AST). Both of these formal models are abstracted from data.

In (FBS05), they propose guarded automata (GA) to be the formal models for both BPEL and the conversation protocol. GA extends Mealy machine 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 representing the BPEL activities communicate. 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 symbolic transition guards can reach the same verification result. We will discuss it in section .

Similar as (FBS05), the author of (Nak05) transform BPEL into EFA (Extended Finite State Automata). A tool is developed to automate the mappings from BPEL to EFA, and from EFA to Promela. EFA extends Mealy machine with data, and it adopts asynchronous interleaving semantics of concurrency. They also has no concern of the interaction among BPEL activities due to data dependencies. It is not clear which communication mechanism is used when modelling read and write data by BPEL activities.

In the above automata based approaches, they only cover core subset of BPEL activities and do not consider fault handling, compensation handling, or event handling.

2.1.4 Summary

For a large system of concurrently executing services, a crucial aspect of the correctness of such system is their temporal behaviour.

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, one can establish whether two processes have equivalent behaviours. Such analyses are 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 analysis, which are useful for model substitution and refinement. However, 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.

The problem of the promising pi-calculus based approaches is that it offers typing for order and format of messages. The limitation of Petri-net based approaches is the single event restriction on every transition, which will easily cause large model state space. Most Petri bet 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 need to fill this gap.

In summary, 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.

2.2 Testing of BPEL based web services

As we can see from the previous section, there are intensive research on providing precise semantics for BPEL and verification of correctness of BPEL models. However, there is less effort in using BPEL as the test models for deriving test cases. A framework is proposed in (BP05a) to augment WSDL with a UML2.0 PSM (Process State Machine)

for modelling the web service interactions. After transforming PSM to a Symbolic Transition System, existing *ioco-conformance* testing tools can be applied. In (HM05), they use Graph Transformation Rules along with WSDL to generate test cases. WSDL-S is proposed in (SP06) 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 so that the existing test techniques for EFSM can be applied.

For the purpose of providing an 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 categorized as test purpose oriented and coverage oriented. 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. So we target at coverage oriented test generation. It is not new to apply model checking to achieve test coverage (HCL+03; HGW04; RH01), but it is new to apply such a technique in the domain of web services.

2.3 Analysis of Data Dependencies in Orchestration

Referring to the works discussed in section , many works abstract from data and directed the attention to the control flow. When data is omitted, the transition guards and variables were left out, so selecting one of two control pathes, solved by the evaluation of data, needs to be modelled by a nondeterministic choice. Even for those works considering BPEL data, they do not consider modelling data dependencies in an explicitly way. In this section, we review the works with consideration of modelling data dependencies in the orchestration models such as BPEL models. The purpose of analyzing data dependencies is to ensure data is always defined before being used.

In (PA03), 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 the consistency, they provide a set of constraints when constructing a data flow graph. 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 data can be connected to multiple input parameters, but an input parameters bound to a constant data cannot have any other incoming data flow edge. A toolset is developed to support the visualization. Even though their focus is not rigorous verification of design models, they shows the importance of considering control dependencies and data dependencies in separation.

In the composition language proposed by (WPSW04), each task (equally to a BPEL activity) has inputDependencies 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 is the source who sends x to it. The task who 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 web service task, channels represent takes 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 proposed formal model should also be able to capture the data dependencies between BPEL activities in an explicit way.

A grid workflow language is proposed in (FQH05), where each activity may have 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 the data exchanges in conditional activities (e.g. BPEL switch activities and pick activities), in sequential loop activities (e.g. BPEL while activities), and in parallel loop activities

(e.g. BPEL flow activities).

The authors in (APS05) 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 is limited to within a composite process. The processes in a composite process can exchange data among themselves or with the parent process. Here an OWL-S process has the same role as a BPEL activity. The control flow features of OWL-S are a lot simpler than BPEL. We will add similar *level-based* constraint on the data exchanges within a BPEL process, and the rationale of adding this constraint will be covered in section 4.

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 need to analyze the data exchanges to get the data dependencies. The work of (BP05b) discusses how to analyze data exchanges between YAWL workflow models, so that the resulted data dependencies between web services can be used for service matching. In (DMCS05), 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 *compositttion axioms* to add constrains 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 to guide the communications between BPEL models, so data dependencies between BPEL models need to be analyzed.

In (MPT06), 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 a orchestration model is modelled by STS (State Transition System) (PTBM05). STS with data is the synchnonized product of all the STSs and data nets. A tool is needed to do the experimental evaluation. Since we assume our formal model includes data, there is no need to add a seperate data model. Data flow

can be derived from our formal model based on existing data flow analysis techniques.

In sum up, it is necessary to model data exchange explicitly in our formal model, to capture the data dependencies within a BPEL model and the data dependencies between BPEL models in a correct way.

2.4 Summary

In this section, we review 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 the approaches of analyzing data dependencies internal and external to an orchestration model. We discuss the motivation to propose a formal model, and the required features of the formal model. Thereafter, we present the reason to design and develop an automated test generation framework. Moreover, we point out the importance of analyzing data aspect of BPEL models.

Chapter 3

Web Service Automata

WSA provides a formal description of the legal protocol for web service interaction, and following some translation step it can be used as a reference model for test case derivation, applying we 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 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. We assume that we have available an enumerable infinite set V of variables and sets AX,BX of assignment expressions and Boolean expressions respectively, together with a set D of values. We also assume that we have a set of functions Env where $\epsilon \in Env : V \rightarrow D$ assigns variables of V with values from D. Given an expression exp, we need three functions:

- $def : AX \rightarrow V$, where $def(exp) \in V$ returns the assigned variable, i.e. the

variable on the left hand side of the assignment.

- $cuses : AX \rightarrow \wp(V)$, where $\wp(V)$ is the power set of V and $cuses(exp) \subseteq V$ returns the variables on the right hand side of the assignment.
- $puses : BX \rightarrow \wp(V)$, where $puses(exp) \subseteq V$ returns the variables in the Boolean expression.

If $exp \in AX$, we need a function $apply_A : AX \times Env \rightarrow Env$ that satisfies:

- If $v \neq def(exp)$ then $apply_A(exp, \epsilon(v)) = \epsilon(v)$. This means only the assigned variable may change.
- If for all $v \in cuses(exp)$ we have $\epsilon_1(v) = \epsilon_2(v)$, then $apply_A(exp, \epsilon_1(v)) = apply_A(exp, \epsilon_2(v))$.
- If for all $\epsilon \in Env$ we have $apply_A(exp_1, \epsilon) = apply_A(exp_2, \epsilon)$, then $exp_1 = exp_2$.

If $exp \in BX$, we need a function $apply_G : BX \times Env \rightarrow \{true, false\}$ that satisfies: if for all $v \in puses(exp)$ we have $\epsilon_1(v) = \epsilon_2(v)$, then $apply_G(exp, \epsilon_1(v)) = apply_G(exp, \epsilon_2(v))$.

Each web service automaton M will be associated with certain variables and expressions. These constitute its data structure. A formal definition of a machine's data structure is given in definition 2.

Definition 2. A **Web Service Automaton** (WSA) M is a finite state machine, consisting of $WSA_M = (I^M, S^M, s_0^M, S_f^M, T^M, \delta^M)$. As a convention, we omit the superscript of M such that $M = (I, S, s_0, S_f, T, \delta)$, and the components of M inherit subscripts, superscripts, or dashes.

1) I is the signature of M, denoted as a three tuple $I = (E, L, O)$, where E, L, O are pair-wise disjoint and represent a set of input events, internal events, and output events, respectively. Let $Msg = (L \cup E \cup O)$ to be the set of events, we refer to the elements of $L = L_{in} \cup L_{out}$ as internal input events and internal output events, and to those of $Msg = (E \cup O)$ as 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) $T \subseteq IN \times BX \times (\wp(AX) \cup OUT)$ is a set of transitions, where $IN = (E \cup L_{in} \cup \{\Omega\})$ and $OUT = (O \cup L_{out} \cup \{\Omega\})$. For each $t = (m, g, a) \in T$ (graphically denoted as $m[g]/a$), $m \subseteq IN$ is a set of triggering events, $g \in BX$ is the guard predicate, and $a \subseteq (\wp(AX) \cup OUT)$ is the action set composed of assignments and output events. Ω indicates the omission of an event. We would represent a transition by $\Omega[g]/\Omega$ which simply determines a state change and nothing else. The elements of transition t are denoted as $t.m = m, t.g = g, t.a = a$.

   - The events of the transition input event set $t.m \subseteq IN$ are linked by logical operator conjunction, disjunction, or negation, denoted as $AND : e_1 \wedge e_2, .., \wedge e_n$, $OR : e_1 \vee e_2, .., \vee e_n$, and $NOT : \neg(e_i)$, respectively.
   - The *data structure* of machine M is the form of (V_M, AX_M, BX_M), which can be retrieved from T. Since $T \subseteq IN \times BX \times (\wp(AX) \cup OUT)$, we can retrieve $AX_M = \{exp \in AX | \exists t \in T \wedge exp \in t.a\}$, $BX_M = \{exp \in BX | \exists t \in T \wedge exp \in t.g\}$, and V_M which is the disjoin union of $\bigcup_{exp \in AX}(\{def(exp)\} \cup cuses(exp))$ and $\bigcup_{exp \in BX}\{puses(exp)\}$.
   - We define that a WSA is *T-complete* iff $t \in T$ then $s \xrightarrow{t} s'$, which means every transition will be triggered from some state.

4) $\delta \subseteq S \times T \times S$ is the transition relation.

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 in the composite automaton (definition 10), $!m$ and $?m$ will become $@!m$ and $@?m$ to indicate that output event !m and input event ?m become internal output and internal input events, respectively.

Definition 3. A WSA is **deterministic** iff $\forall s, s', s'' \in S. \forall t \in T,\ s \xrightarrow{t} s' \wedge s \xrightarrow{t} s'' \Rightarrow s' = s''$ holds.

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 $!x \in O_1 \wedge ?x \in E_2$.

3.2 Dynamic Semantics

We have defined the static structure of a WSA. We now explain its dynamic semantics. We assume, first of all, that a WSA is equipped with a means of storing incoming messages. Instead of limiting the buffers to a particular type such as FIFO or multi-set, we consider different buffering schemes (section 3.4). We assume that each WSA is associated with a **finite buffer**. The buffer $\beta(E \cup L_{in})$ stores the external and internal input events. In one **step** of the WSA, either: 1) a message e is received from the environment $e \in E$, and added to the buffer $\beta(E \cup L_{in})$; or 2) an enabled transition fires, possible causing a state change and change to the values of some variables $v \in V_M$. In order to precisely define a step, we need to formalise the notion of a **configuration** η 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 η, when 1) its triggering event set $t.m$ is either empty or belonging to $\beta(E \cup L_{in})$, 2) $t.m$ can be consumed according to the machine's buffering scheme, and 3) the transition guard $t.g$ is evaluated to be true. 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 transition is called **enabled transition**.

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 λ 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_i with $\lambda(o_i) = ?x$. 2) When applying λ to message instances and messages, for a message instance $om = (!om, ?om)$, there exists message $m = (!m, ?m)$ such that $\lambda(!om) = !m$ and $\lambda(?om) = ?m$.

Definition 5. A **configuration** of a machine M is of the form $\eta = (s, B, \epsilon)$, where $s \in S, B \subseteq \beta(E \cup L_{in})$ is the current buffer content, and function $\epsilon : V \rightarrow D$ assigns the variable of V with a value from D. The set of configurations of M is denoted as $confs(M)$ where $\eta \in confs(M)$. Note that we assume the buffer is empty when M is either in the initial state or in a final state (hypothesis 1), so $\eta_0 = (s_0, \emptyset, \epsilon_0)$ and $\eta_n = (s_n, \emptyset, \epsilon_n)$ are the initial and a final configuration, respectively.

Hypothesis 1. The buffer is empty when machine M is in a final state. If M is in a final state, we say a lifecycle of the machine ends. If the buffer is not empty then the remaining messages will be discarded because they cannot be consumed within this lifecycle.

Definition 6. A **step** is a triple $(\eta, x, \eta') \in confs(M) \times (T \cup E \cup L_{in}) \times confs(M)$. A step $\eta \mapsto^x \eta'$ changes machine M from configuration η to configuration η' in the following forms:

1) If $x \in E \cup L_{in}$, representing the case when a message x is arriving and is added to the buffer B, then $(s, B, \epsilon) \mapsto^x (s', B', \epsilon')$ iff $s = s', \epsilon = \epsilon', B' = B + \{x\}$.

2) If $x = t \in T$, representing the case when either the message is already in the buffer $t.m \in B$ or there is no triggering event associated with t, then $(s, B, \epsilon) \mapsto^t (s', B', \epsilon')$ iff t is an enabled transition, i.e. $(s, t, s') \in \delta, B' = (B \setminus \{t.m\} + (t.a \cap L_{in}), apply_G(t.g, \epsilon) = true$.

Definition 7. We define a set of **configuration sequences** of M as $configs(M, \eta_0)$. A configuration sequence of M,$\alpha = \langle \eta_0, x_0, .., \eta_n) \rangle \in configs(M, \eta_0)$, is a sequence of configurations linked by input events,internal input events, or enabled transitions from the initial configuration to a final configuration, where $\eta_i = (s_i, B_i, \epsilon_i)$, $x \in E \cup L_{in} \cup T$.

1) We define a set of **transition sequences** as $trans(M)$, where a transition sequence of M is a sequence of enabled transitions $trans(\alpha) = \langle x_0, .., x_{n-1} \rangle$ such that $trans(M) = \{trans(\alpha)\}$, where $x_i \in T, \alpha = \langle \eta_0, x_0, .., \eta_n \rangle \in configs(M, \eta_0)$, and $trans(\eta_0) = \{\Omega\}$.

2) We define the set of **traces** as $trances(M)$. A trace of machine M is a non-empty sequence of external event occurrences $\langle o_1, .., o_n \rangle$, such that $trace(M) = \{\langle o_1, .., o_n \rangle\}$, where for all enabled transitions $trans(M)$, $\lambda(o_i) \in t.m \subseteq E$ or $\lambda(o_i) \in t.a \cap O \neq \emptyset$ holds.

Definition 8. Let $SReach(M) = \{s \in S | \exists \alpha \in traces(M).s_0 \xrightarrow{\alpha} s\}$ 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 $SReach(M) = S \wedge TReach(M) = T$.

3.3 Composition

In order for our notion of composition to be commutative and associative, we introduce the unordered Cartesian product (definition 9). 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 10). Such composite automaton is not guarantee to be a WSA (definition 11). We then elicit a set of constraints on the individual WSAs as the necessary, sufficient, or necessary and sufficient conditions for a composite automaton to be a WSA or to be a deterministic WSA. For strongly composable WSAs (definition 12), we prove that the buffer content of the composite

automaton is a unique pair, the configurations as well as configuration sequences of individual WSAs can be projected from the composite automaton, and we also prove that the composition operator is associative.

Definition 9. If $(X_i)_{i\in I}$ is an indexed family of sets, we define $\prod_{i\in I} X_i$ 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 $\prod_{i\in I} X_i$ as the **unordered Cartesian product** of a set of X_i. Note that $\prod$ is a commutative and associative binary operator.

Definition 10. If M_1, M_2 are WSAs, then we define their **Composite Automaton** $\hat{M} = M_1 \parallel M_2$ to be the structure $\hat{M} = (\hat{I}, \hat{S}, \hat{s_0}, \hat{S_f}, \hat{T}, \hat{\delta})$, where

1) $\hat{I} = (\hat{E}, \hat{L}, \hat{O})$, we define com_{12} to be the common messages of M_1, M_2 by $com_{12} = (E_1 \cap O_2) \cup (E_2 \cap O_1)$. Now we can define $\hat{I}$ by $\hat{E} = (E_1 \cup E_2) \setminus com_{12}$, $\hat{L} = E_1 \cup E_2 \cup com_{12}$, and $\hat{O} = (O_1 \cup O_2) \setminus com_{12}$.

2) $\hat{S} = S_1 \prod S_2$.

3) $\hat{s_0} = \{s_0^1\} \prod \{s_0^2\}$.

4) $\hat{S_f} = \{S_f^1\} \prod \{S_f^2\}$. A state of $M_1 \parallel M_2, .., \parallel M_n$ 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 $\forall v \in V_i$ is a local variable of machine M_i, where $V_i = \{v \in V | \exists exp \in AX_i \cup BX_i\}$. This means that there are no shared variable between $t \in T_i$ and $t \in T_j$ where $i \neq j$.

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

Definition 11. M_1, M_2 are **composable** iff $Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$.

Definition 12. M_1, M_2 are **strongly composable** iff a) $Msg_1 \cap Msg_2 = com_{12}$, and b) $T_1 \cap T_2 = \emptyset$.

Theorem 1. $\hat{M} = M_1 \parallel M_2$ are WSAs iff $Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$ (1).

Proof. Expanding $Msg_1 \cap Msg_2$, we get $Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap L_2) \cup (L_1 \cap E_2) \cup (O_1 \cap L_2) \cup (O_2 \cap L_1) \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$ (2). Suppose that $\hat{M}$ is a WSA, by lemma 1 $\hat{E} = (E_1 \cap L_2)(E_2 \cap L_1) \setminus com_{12} = \emptyset$, and based on set theory $A \setminus B = \emptyset \Rightarrow A \subseteq B$, we have $(E_1 \cap L_2) \cup (E_2 \cap L_1) \subseteq com_{12}$. Similarly, we can get $(O_1 \cap L_2) \cup (O_2 \cap L_1) \subseteq com_{12}$. This results (1).

Next, we need to prove if (1) holds then $\hat{M}$ is a WSA. If (1) holds, from (2) we can deduce $(E_1 \cap L_2) \cup (E_2 \cap L_1) \cup (O_1 \cap L_2) \cup (O_2 \cap L_1) \subseteq com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$ (3) . If $m \in (E_1 \cap L_2) \cup (E2_{L1})$ then $m \notin com_{12}$ 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 $E \cap L \subseteq (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \setminus com_{12}$.Based on the pair-wise disjoint of E_i, L_i, O_i, if $m \in \hat{E} \cap \hat{L} = (E_1 \cap L_2) \cup (E_2 \cap L_1) \setminus com_{12}$ then $m \notin (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \setminus com_{12}$. 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 $A \cap B = \emptyset \wedge A \subseteq B \Rightarrow A = \emptyset$, we have $\hat{E} \cap \hat{L} = \emptyset$. Similarly, $\hat{O} \cap \hat{L} = \emptyset$ holds.

Thereafter, by (3) of lemma 1, we have $\hat{O} \cap \hat{L} = \emptyset$. This indicates that the internal messages of a machine are pair-wise disjoint with the external messages of the partner machine, i.e. $E_i \cap L_j = \emptyset$ and $O_i \cap L_j = \emptyset$ where $i \neq j$.

Finally, we need to show that $\hat{M} = M_1 \parallel M_2$ is *T-complete*. For a transition $t \in \hat{T}$, without loss of generality, we may suppose that $t \in \hat{T}$, then since M_1 is T-complete, there exists s_1, s_1', s such that $s_1 \overset{t \in T_1}{\rightarrow} s_1'$, which results $(s_1, s_2) \overset{t \in \hat{T}}{\rightarrow} (s_1', s_2)$. Therefore, $\hat{M}$ is T-complete.

Corollary 1. Suppose that M_1, M_2 are WSAs and $\hat{M} = M_1 \parallel M_2$, if $(E_1 \cap E_2) =$

$(L_1 \cap L_2) = (O_1 \cap O_2)$ (1), then $\hat{M}$ is a WSA.

Proof. From the proof of theorem 1 we have $E \cap L \subseteq (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) \setminus com_{12}$, if(1) holds then $\hat{E} \cap \hat{L} = \emptyset$. Similarly, $\hat{O} \cap \hat{L} = \emptyset$ holds, and by lemma 1 $\hat{E} \cap \hat{O} = \emptyset$ holds.

Lemma 2. If $\hat{M} = M_1 \parallel M_2$ is deterministic, then M_1 and M_2 are deterministic.

Proof. We use $\rightarrow_i$(resp. $\hat{\rightarrow}$) to denote a transition of M_i (resp. $\hat{M}$. For $t \in \hat{T}$, suppse $t \in T_1$ we have $s_1 \rightarrow_1 s_1' \wedge s_1 \rightarrow_1 s_1''$, then $(s_1, s_2)\hat{\rightarrow}(s_1', s_2) \wedge (s_1, s_2)\hat{\rightarrow}(s_1'', s_2)$. If $\hat{M}$ is deterministic, we must have $s_1' = s_1''$. Hence M_1 is deterministic; likewise M_2 is deterministic.

Lemma 3. Suppose M_1, M_2 are WSAs and $\hat{M} = M_1 \parallel M_2$, if (1)$M_1, M_2$ are deterministic and (2) $T_1 \cap T_2 = \emptyset$, then $\hat{M}$ is deterministic.

Proof. For $t \in \hat{T}$,$(s_1, s_2)\hat{\rightarrow}(s_1', s_2') \wedge (s_1, s_2)\hat{\rightarrow}(s_1'', s_2'')$. Suppose that $T_1 \cap T_2 = \emptyset$ (2), in the case of $t \in T_1$, we must have $s_1 \rightarrow_1 s_1' \wedge s_1 \rightarrow_1 s_1''$ and $s_2 = s_2' = s_2''$, but then $s'1 = s''1$ by deterministic of M_1 (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 and $T_1 \cap T_2 = \emptyset$, then $\hat{M}$ is deterministic.

The case $T_1 \cap T_2 \neq \emptyset$ is slightly trickier and we merely include it for completeness. We define a transition t to be self-loop in machine M, written as $t \in self(M)$, iff for all $s, s' \in S_M$, if $s \xrightarrow{t} s'$ then $s = s$;.

Proposition 1. Suppose M_1, M_2 are WSAs, $\hat{M} = M_1 \parallel M_2$ is deterministic iff (1) M_1, M_2 are deterministic, (2) $T_1 \cap T_2 \subseteq self(M_1) \cap self(M_2)$.

Proof. Suppose (2) holds and $(s_1, s_2)\hat{\rightarrow}(s_1', s_2') \wedge (s_1, s_2)\hat{\rightarrow}(s_1'', s_2'')$. If $t \in T_1 \setminus T_2$ or $t \in T_2 \setminus T_1$, then $(s_1', s_2') = (s_1'', s_2'')$ by argument perceived of lemma 3. So we only need to consider the case in which $t \in T_1 \cap T_2$, by (2) we get $t \in self(M_1) \cap self(M_2)$. Hence $(s1, s2) = (s_1', s_2') = (s_1'', s_2'')$, $\hat{M}$ is deterministic.

Next we consider the converse. Suppose that $\hat{M}$ is deterministic and $t \in T_1 \cap T_2$, by *T-completeness* there exists $s1, s_1', s2, s_2'$ such that $s_i \rightarrow_i s'i$ where $i = 1, 2$, we have $(s_1, s_2) \hat{\rightarrow} (s_1', s_2) \wedge (s_1, s_2) \hat{\rightarrow} (s_1, s_2')$. Since $\hat{M}$ is deterministic, we have $(s_1', s_2) = (s_1, s_2')$. Because s_1, s_1' are arbitrary, we can conclude that $t \in self(M_1)$. We can get $t \in self(M_2)$ similarly. Hence, we prove that $t \in self(M_1) \cap self(M_2)$.

Theorem 2. Suppose M_1, M_2 are WSAs, $\hat{M} = M_1 \parallel M_2$ is a deterministic WSA iff

(1) M_1 and M_2 are deterministic;

(2) $Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$;

(3) $t \in self(M_1) \cap self(M_2)$.

Proof. If $\hat{M} = M_1 \parallel M_2$ is a deterministic WSA, then (1) is proved by lemma 2, (1) is proved by theorem 1, and (3) is proved by proposition 1. Conversely, the condition (2) proves that $\hat{M}$ is a WSA following theorem 1; and the condition (1) and (3) proves that $\hat{M}$ is deterministic following proposition 1.

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

(1) $(E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) = \emptyset$;

(2) M_1, M_2 are composable;

(3) If M_1, M_2 are deterministic then $\hat{M} = M_1 \parallel M_2$ is also deterministic.

Proof. From definition 12, if M_1, M_2 are strongly composable then $Msg_1 \cap Msg_2 = com_{12}$. From theorem 1 we have $Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$ and $com_{12} \cap ((E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)) \neq \emptyset$ because of pair-wise disjoint property of E_i, L_i, O_i. Hence we can prove that (1) holds. (2) holds following corollary 1. By definition 12 and lemma 3, (3) can be proved.

Lemma 4. 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 $B_1 \in \beta(E_1 \cup L_1)$ and $B_2 \in \beta(E_2 \cup L_2)$ such that $B_1 \cap B_2 = B$ and $B_1 \cap B_2 = \emptyset$, i.e.$B = B_1 + B_2$.

Proof. Let $B_i(m) = \begin{cases} B_i(m) & m \in E_i \cup L_i \\ 0 & otherwise \end{cases}$, we shall show that $(E_1 \cup L_1) \cap (E_2 \cup L_2)$ (1), from this it follows $B_1 \cap B_2 = \emptyset$ so that a message m_1 implies $m \notin B_2$. For (1), we have $com_{12} \cap ((E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)) = \emptyset$. Based on the pair-wise disjoint property of E_i, L_i, O_i, the proof of theorem 1 shows that $E_i \cap L_j = \emptyset$ and $O_i \cap L_j = \emptyset$ where $i \neq j$. From proposition 2, if M_1, M_2 are strongly composable then $(E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2) = \emptyset$, so we can prove that (1) holds as claimed.

Lemma 5. Suppose M_1, M_2 are strongly composable, let $\hat{M} = M_1 \parallel M_2$ and $\eta \in Conf(\hat{M})$, then $project_i(\eta) \in Conf(M_i)$ where $project_i(\hat{s}, \hat{B}, \hat{\epsilon}) = (s_i, project_i(\hat{B}, \epsilon\,|\hat{V}_i)$, , i.e. $project_i(\eta) = \eta_i$.

Proof. By lemma 4 we have $project_i(\hat{B}) \in \beta(E_i \cup L_i)$, it follows that the configuration of individual automaton η_i can be projected from the configuration of composite automaton $\eta \in Conf(\hat{M})$ by $project_i(\eta)$.

Lemma 6. Suppose that M_1 and M_2 are strongly composable, let $\hat{M} = M_1 \parallel M_2$, then $\alpha \in configs(\hat{M} \Rightarrow project_i(\alpha) \in configs(M_i))$ (1).

Proof. Since both $trans(\hat{M})$ and $traces(\hat{M})$ can be retrieved from $configs(\hat{M})$, it is sufficient to prove (1) holds. Based on definition 8, we need $\eta \mapsto^x \eta'$ to represent a step between configurations.

First, when $x \in \hat{E}$, we have $\eta \mapsto^x \eta'$ iff 1) In the case of $x \in E_1 \setminus \hat{L}$,$project_1(\eta) \mapsto^x project_1(\eta')$ and $project_2(\eta) = project_2(\eta')$. By lemma 4 it follows $x \in \hat{E} \Rightarrow \eta_1 \mapsto^x \eta_1' \wedge \eta_2 = \eta_2$. From definition 8 it follows that the conditions $s_1 = s_1'$,$B_1' = B_1 + x$, and $\epsilon_1 = \epsilon_1'$ holds for $\eta_1 \mapsto^x \eta_1'$. 2) In the case of $x \in E_2 \setminus \hat{L}$,$project_2(\eta) \mapsto^x project_2(\eta')$ and $project_1(\eta) = project_1(\eta')$. Similar to 1), we can get $x \in \hat{E} \Rightarrow \eta_2 \mapsto^x \eta_2' \wedge \eta_1 = \eta_1$, and the conditions $s_2 = s_2'$,$B_2' = B_2 + x$, and $\epsilon_2 = \epsilon_2'$ holds for $\eta_2 \mapsto^x \eta_2'$.

Second, when $t \in \hat{T}$, we have $\eta \mapsto^{x} \eta'$ iff 1) In the case of $x \in T_1$, $project_1(\eta) \mapsto^{x} project_1(\eta')$ and $project_2(\eta) \mapsto^{x.a \cap E_2} project_2(\eta')$. By lemma 5, it follows $x \in T_1 \Rightarrow \eta_1 \mapsto^{x} \eta_1'$ and $\eta_2 \mapsto^{x.a \cap E_2} \eta_2'$. 2) Similarly, in the case of $x \in T_2$, $project_2(\eta) \mapsto^{x} project_2(\eta')$ and $project_1(\eta) \mapsto^{x.a \cap E_1} project_1(\eta')$. By lemma 5, it follows $x \in T_2 \Rightarrow \eta_2 \mapsto^{x} \eta_2'$ and $\eta_1 \mapsto^{x.a \cap E_1} \eta_1'$.

As a result, for projection on transition and messages, we have $project_i(x_0, .., x_{n-1}) = project_i(x0), .., project_i(x_{n-1})$. If $x_i \in T_1 \cup E_1 \setminus \hat{L}$ then $project_1(xi) = x_i$. If $x_i \in T_2$ then $project_1(x_i) = t.a \cap E_1$.

Lemma 7. The composition operator is *communtative*. Suppose M_1, M_2 are WSAs, $M_1 \parallel M_2 = M_2 \parallel M_2$.

Proof. Given $M_A = M_1 \parallel M_2 = (I^A, S^A, s_0^A, S_f^A, T^A, \delta^A)$ and $M_B = M_2 \parallel M_1 = (I^B, S^B, s_0^B, S_f^B, T^B, \delta^A)$, following the definition of web service automaton, the elements of M_1 are symmetrical with the elements of M_2. For instance, $E_A = (E_1 \cup E_2) \setminus com_{12} = E_B$. Following the commutative nature of the set operators $\cup \cap \prod$, we can prove $I_A = I_B, s_0^A = s_0^B, s_f^A = s_f^B, T_A = T_B$ and $\delta^A = \delta^B$. Hence $M_1 \parallel M_2 = M_2 \parallel M_2$ is proved as claimed.

Lemma 8. The composition operator is *associative*. Suppose M_1, M_2, M_3 are WSAs, $(M_1 \parallel M_2) \parallel M_3 = M_1 \parallel (M_2 \parallel M_3)$.

Proof. Suppose $M_A = (M_1 \parallel M_2) \parallel M_3 = (I^A, S^A, s_0^A, S_f^A, T^A, \delta^A)$ and $M_B = M_1 \parallel (M_2 \parallel M_3) = (I^B, S^B, s_0^B, S_f^B, T^B, \delta^B)$, following the commutative nature of the set operators $\cup \cap \prod$, we can prove $s_0^A = s_0^B, s_f^A = s_f^B, T_A = T_B$ 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 $E = E_A \cup E_B \cup E_C$, $L = L_A \cup L_B \cup L_C$, $O = O_A \cup O_B \cup O_C$, and $X = X_{12} \cup X_{13} \cup X_{23}$ where $X_{12} = com_{12}$,$X_{13} = com_{13}$, and $X_{23} = com_{23}$.

First, we need to prove $E_A = E_B$.

- Let $E_a = E_{1\|2} \cup E_3$ and $X_a = (E_{1\|2} \cup O_3) \cup (O_{1\|2} \cup E_3)$, where $E_{1\|2} = (E_1 \cup E_2) \cap \bar{X_{12}}$ and $O_{1\|2} = (O_1 \cup O_2) \cap \bar{X_{12}}$. 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 $E_3 \cap X_{12}$ (5). From (4)(5) it results $E_A = E \cap (\bar{X} \cup (E_3 \cap \bar{X})) = E \cap \bar{X}$.

- Similarly, let $E_b = E_{2\|3} \cup E_1$ and $X_b = (E_{2\|3} \cup O_1) \cup (O_{2\|3} \cup E_1)$, where $E_{2\|3} = (E_2 \cup E_3) \cap \bar{X_{23}}$ and $O_{2\|3} = (O_2 \cup O_3) \cap \bar{X_{23}}$. 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 $L_A = (L_{1\|2} \cup L_3) \cup (E_{1\|2} \cap O_3) \cup (O_{1\|2} \cap E_3)$ and $L_{1\|2} = L_1 \cup L_2 \cup X_{1\|2}$, we have $L_A = L \cup X_{1\|2} \cup X_a$ where $X_a = (E_{1\|2} \cap O_3) \cup (O_{1\|2} \cap E_3)$ (9). By (3)(9), $L_A = L \cup X_{1\|2} \cup X_{1\|3} \cup X_{2\|3} = L \cup X$ can be derived.

Similarly, by $L_B = (L_{2\|3} \cup L_1) \cup (E_{2\|3} \cap O_1) \cup (O_{2\|3} \cap E_1)$ and $L_{2\|3} = L_2 \cup L_3 \cup X_{2\|3}$, we have $L_B = L \cup X_{2\|3} \cup X_b$ (10), where $Xb = (E_{2\|31}) \cup (O_{2\|3} \cap E_1)$. By (8)(10), $L_A = L \cup \cup X_{1\|3} \cup X_{2\|3} = L \cup X$ can be derived. Therefore, $L_A = L_B$ is proved as claimed.

3.4 Buffering Schemes

Since there is more than one reasonable buffering mechanism, the buffering scheme of WSA should be flexible and configurable. Inspired by Alur (AHP96), we identify the commonly used buffering schemes:

1) FIFO buffering scheme: A machine has one FIFO buffer. The CFSM (BZ83) and the guarded automata (FBS05) apply this scheme. No message overtaking is allowed. A message can be consumed only when it is at the head of the FIFO buffer.

2) Multi-set (definition 13) buffering schemes: a) Multi-set buffer with FIFO sub-buffers: A machine has one multi-set buffer that 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. b) 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. c) 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 of b).

Definition 13. A **finite multi-set** is formally defined as a pair (X, n), where X is some set and $n : X \rightarrow N$ 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 generalized for multisets.

Note that without loss of generality, we assume the communication channels are lossless, so that every message is always received after it is sent. Message overtaking may occur during communication, Figure 3.1 shows two types of message overtaking in MSCs.

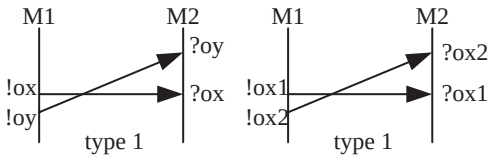

Figure 3.1: Message overtaking

For type 1, 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 from the order of send event occurrences $!ox, !oy$. For type 2, suppose message $x = (!x, ?x)$, there exist message instances $ox_1 = (!ox_1, ?ox_1)$ and $ox_2 = (!ox_2, ?ox_2)$ where $\lambda(ox_1) = \lambda(ox_2) = x$. Message overtaking happens when the order of receive event occurrences $?ox_1, ?ox_2$ is different from the order of send event occurrences $!ox1, !ox2$.

We assume that WSA with both FIFO and multi-set buffering schemes only allow type 1) but not type 2) of message overtaking.

3.5 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.

3.5.1 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 $x = (!x, ?x)$ such that $!x \in O_1$ and $?x \in E_2$.

Definition 15. Two WSAs M_1, M_2 are **semantically compatible** if a) they are strongly composable, and b) $trans(M_1 \parallel M_2) \neq \emptyset$.

Now we discuss when the condition $trans(M_1 \parallel M_2) \neq \emptyset$ holds. This condition holds iff for any $\hat{\eta} = ((s_0^1, s_0^2), B, (\epsilon_0^1, \epsilon_0^2))$ there is no transition $t \in \hat{T}$ such that $\hat{\eta} \mapsto^t \hat{\eta'}$, where $(\epsilon_0^1, \epsilon_0^2)$ denotes the initial values of variables. First, suppose the initial con-

figuration is $\hat{\eta}_0 = ((s_0^1, s_0^2), B, (\epsilon_0^1, \epsilon_0^2))$, if $B \in \beta(\hat{E})$ then there exists α such that $\hat{\eta}_0 \mapsto^{\alpha} \hat{\eta_B} = ((s_0^1, s_0^2), B + \alpha, (\epsilon_0^1, \epsilon_0^2))$, where α consists solely of external input events, i.e. $\alpha \in \hat{E}$. Second, from definition 10, it shows $(s_i^1, s_i^2) \xrightarrow{t} (s_j^1, s_j^2)$ iff a) if $t \in T_1$ then $s_i^2 = s_j^2 \wedge s_i^1 \xrightarrow{t} s_j^1$, or b) if $t \in T_2$ then $s_i^1 = s_j^1 \wedge s_i^2 \xrightarrow{t} s_j^2$. $\hat{\eta_B} \mapsto^{t \in T_1} \Leftrightarrow \eta_B^1 \mapsto^{t \in T_1}$ or $\eta_B^2 \mapsto^{t \in T_2}$. Here $\eta_B^i = (s_0^i, proj_i(B), \epsilon_0^i)$, and $proj$ is the projection operator. Without loss of generality suppose $\hat{\eta} \mapsto^{t \in T_2} \hat{\eta'}$, we have $s_0^1 \xrightarrow{t \in T_1} s_1^1$, $t.m \in B$, and $apply_G(t.g, \epsilon_1) = true$. Conversely, we have $\neg(\hat{\eta_B} \mapsto^{t \in T_1})$ iff for all B either $\neg(s_0^1 \xrightarrow{t \in T_1})$, $t.m \notin \hat{E}$, or $apply_G(t.g, \epsilon_1) \neq true$.

As a result, condition $trans(M_1 \parallel M_2) \neq \emptyset$ holds iff for $\forall B \in \beta(\hat{E})$, there does not exist $t \in \hat{T}$ such that $\hat{\eta_B} \mapsto^t$, which indicates that no transition is possible after receiving as many as external inputs.

3.5.2 Anti-patterns

According to definition 15, the condition $trans(M_1 \parallel M_2) \neq \emptyset$ 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.

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. (Wik07)).

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

Corollary 1. 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 $!om < ?om$ enforces that a message instance must be received after it is sent.

Definition 17. In machine M, suppose $e, e' \in Msg_M$, the mutually exclusive relation on event occurrences $o_1 \sharp o_2$ 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 machine M_i is a structure $l_i = (C_i, <_i, \sharp_i)$, where C_i is the event occurrence set of $E_i \cup O_i$, $<_i$ and $\sharp_i$ are the strict partial order and the mutual exclusion relations on C_i, respectively.

Definition 19. For machines $\{M_1, .., M_n\}$, we define **message orderings** to be the structure $<_X = \bigcup_{\lambda(!om),\lambda(?om)\in X}(!om < ?om)$, where $om = (!om, ?om)$, $X \subseteq \bigcup_{1\leq i,j\leq n} com_{ij}$ is the set of messages sent between machines $\{M_1, .., M_n\}$, and $<$ is the strict partial order on event occurrence pairs.

Definition 20. The **causal ordering** for a set of machines $\{M_1, .., M_n\}$ is the structure $\prec_C = (\bigcup_{1\leq i\leq n} l_i) \cup <_X$, 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 $\alpha \in traces(M)$ such that $s_0 \xrightarrow{\alpha} s$ and $\neg(s \xrightarrow{t})$ for $\forall t \in T$. Referring to definition

8, let S_d be the set of all blocking states,$S_d = \{s \in SReach(M) \setminus S_f : \forall t \in T.\neg(s \xrightarrow{t})\}$, so M is blocking iff $S_d \neq \emptyset$.

In state machine diagrams, an initial state is pointed by an arrow started 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 $ox = (!ox, ?ox)$ and $oy = (!oy, ?oy)$ of x and y, respectively, such that $\lambda(!ox) = !x$ and $\lambda(!oy) = !y$. 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, 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 $?oy <_1 !ox$ and $?ox <_2 !oy$ (1).

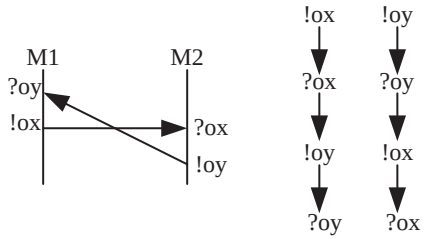

Figure 3.2: Unspecified reception

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 $!ox <?ox <!oy <?oy$ and $!oy <?oy <!ox <?ox$. 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.

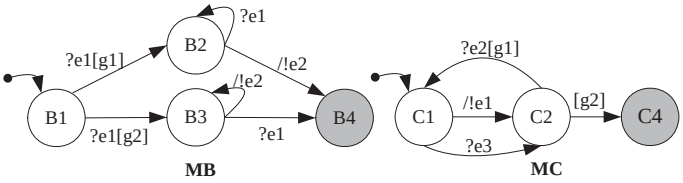

Figure 3.3: An example of anti-pattern 1

Figure 3.3 illustrates an example. $traces(M_B)$ is composed of $\langle ?oe_1[1], ?oe_1[2], !oe_2\rangle$ and $\langle ?oe_1[1], !oe_2, ?oe_1[2]\rangle$. $traces(M_C)$ is composed of $\langle !oe_1[1], ?oe_2, !oe_1[2]\rangle$, $\langle !oe_1[1], ?oe_2, ?oe_3\rangle$, $\langle !oe_1[1]\rangle$, and $\langle ?oe_3\rangle$. 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-patten1, $M_B \parallel M_C$ 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 $S_d^B = \{B_2\}$ and $S_d^C = \{C_2\}$.

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 $!ox \sharp_1 !oy$ (2).

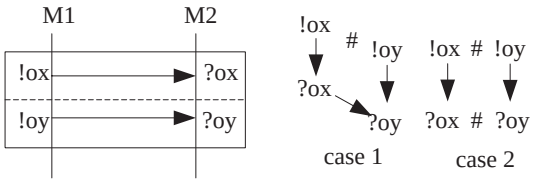

Figure 3.4: Non-local branching 1

Figure 3.4 shows the MSC and casual 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 $\prec_c$:$!ox \sharp !oy \wedge !ox <?ox \wedge !oy <?oy \wedge ?ox <?oy$. By $!ox \sharp_1 !oy$ and message ordering, we have $ox \Rightarrow \neg(!oy) \Rightarrow \neg(?oy)$ and $oy \Rightarrow \neg(!ox) \Rightarrow \neg(?ox)$, so $?ox <_2 ?oy$ does not

hold.

2) If $?ox\sharp_2?oy$, we have casual ordering $\prec_c$:$!ox\sharp!oy \wedge !ox <?ox \wedge !oy <?oy \wedge ?ox\sharp?oy$. If $!ox$ (resp. $!oy$) happens in M_1 and $?oy$ (resp. $?ox$) 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, we have $!om_2\sharp_A!om_3$. In case 1), we have $?om_2 <_B ?om_3$, so $M_A \parallel M_B$ has non-local branching choice with $S_d^B = \{B_4\}$ or $S_d^B = \{B_5\}$. In case 2), we have $?om_2\sharp_B?om_3$, so $M_A \parallel M_B$ has non-local choice with $S_d^B = \{B_2\}$ or $S_d^B = \{B_4\}$.

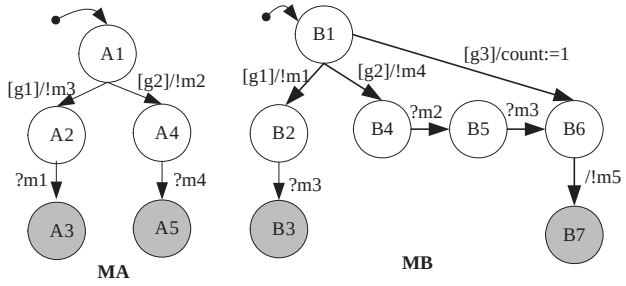

Figure 3.5: An example of anti-patterns 2 and 3

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 $!ox\sharp_1?oy$ and $?ox\sharp_2!oy$ (3).

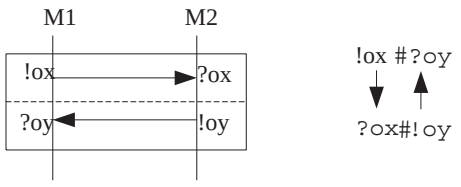

Figure 3.6: Non-local branching 2

Figure 3.6 shows the MSC and casual ordering. M_1 sends message instance ox to M_2, and M_2 sends message instance oy to M_1. The causal ordering is $\prec_c$:$!ox <?ox$, $!oy <?oy$, $!ox\sharp?oy$, and $?ox\sharp!oy$. 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, $!om_3 \sharp_A ?om_4$ and $?om_3 \sharp_B !om_4$ 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, $!om_2 \sharp_A ?om_1$ and $?om_2 \sharp_A !om_1$ will cause non-local choice. In this example, $om_2 \sharp_A om_3$ will also cause non-local choice with the type of anti-pattern2.

The above anti-patterns can be encoded into temporal logics, so that model checking tools (e.g. (Hol03; CCGR99)) can be used to automate the pre-checking.

3.6 Summary

The formal static semantics and dynamic semantics of Web Service Automaton are given in this section. The web service automaton is more general than the existing automata-based semantics because we include multiple-event transitions, data, and internal events. We use message-passing as the uniform mechanism for machine communications. Aslo, anti-patterns are identified for web service interactions. The anti-pattern pre-checking has the advantage that it needs less computation effort to identify incompatible behaviours, and the corresponding properties can be reused.

Chapter 4

Analysis of BPEL Features in Web Service Automata

BPEL consists of two categories of activities: basic and structured activities. Basic activities are atomic actions. Structured activities impose control flow dependency constrains on the executions of either the basic or structured activities within them. A structured activity can contain arbitrary depth of sub-activities. Like many programming languages, BPEL has control structures including *pick, switch, while, scope, eventHandlers, faultHandlers* constructs to express control dependencies between activities. Also, BPEL has *sequence, flow* constructs to support sequential and concurrent relationship between activities, and *compensationHandler* construct to reverse the completed activities. For data handling, BPEL uses 'blackboard approach', where a set of variables is shared by all activities.

In this section, we analyse BPEL main features and describe how to capture these features in WSA. The detailed mapping from BPEL to WSA will be covered in (Zhe07). Each BPEL activity is associated with a machine. We use *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 by an arrow started with a filled black circle, and a final state is shaded.

BPEL consists of basic and structured activities. Basic activities are atomic actions. Structured activities impose control dependencies on the executions of either the basic or structured activities within them. A structured activity can contain an arbitrary depth of sub-activities. BPEL has structured activities including the *pick, switch, while, sequence, flow, scope, eventHandlers, faultHandlers*, and a *compensationHandler* structured activity to reverse completed activities. BPEL handles data using a *blackboard* approach, where a set of variables is shared by the enclosed activities.

In this section, we analyse the main features of BPEL and describe how to capture these features in WSA. We use a loan approval process example (**?**) to illustrate the data flow model and the control flow hierarchy. We use 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 by an arrow starting with a filled black circle, and a final state is shaded.

4.1 BPEL Control Flow

4.1.1 Hierarchy

A WSA has no hierarchy. We simulate the hierarchical relationships of BPEL activities by adding start message and done message as common administration messages between machines. A machine can play the role of parent or child. For a machine M, its parent machine is the one who sends a start message to M, while its child machine is the one who receives a start message from M. A child machine will send a done message to its parent machine when reaching one of its final states. For a machine M_j, if M_i sends a start message to M_j, then M_i is the parent machine of M_j and M_i is the child machine of M_i. A child machine will send a done message to its parent machine when reaching one of its final states. Each machine has zero or one parent machine, and zero or many children 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 the machine

hierarchy of the loanapproval example. The machine without an incoming dark arrow (start message) is the *process* machine. The machine without an incoming hollow arrow (done message) is a *basic* machine. The *process* is the parent of *flow*, and the *flow* is the parent of *receivelinkWrapper*, which in turn is the parent of *receive*.

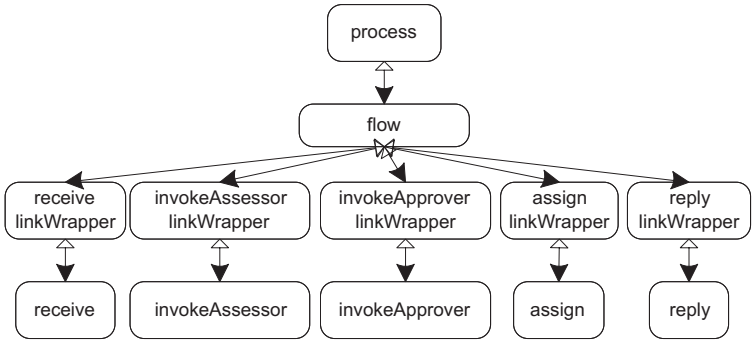

Figure 4.1: An example of machine hierarchy

4.1.2 Concurrency, Fault Propagation, and Interruption

We use logical operators AND, OR, NOT to capture BPEL concurrency, fault propagation and interruption features. For 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 4.2. 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.

For fault message 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 4.2. Instead of using a queue for each fault,

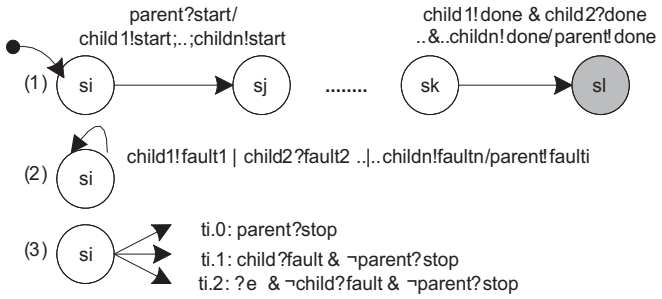

Figure 4.2: Propositional input events and common machine structure

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.

BPEL has two kinds of Interruptions. First, when a termination message is thrown when a *terminate* is reached, the process machine ends abnormally, and a stop message is propagaged downstream. Second, when a fault is thrown by a throw activity, the normal activity needs to stop before the fault handler starts. The fault is propagated upstream until it can be caught by a *scope* or *process* activity that has the fault handler for this fault. 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 4.2, transition ti.0 is triggered when either a stop or a term message arrives. The ti.1 will be triggered when it does not receive a fault message from its child, and only when neither stop nor term message arrives. It indicates that a fault will not be propagated when the machine is asked to stop or terminate. The ti.2 indicates the normal event e cannot trigger the transition when either a fault arises or an interruption message arrives.

With consideration to fault propagation and machine interruption, BPEL structured activities have a common machine structure, shown in (4) of Fig 4.2. Each structured activity machine has a *stopStatus* as a local variable. When the machine receives the children's done message, based on the value of stopStatus, the machine

enters a normal or abnormal final state. Suppose M is a structured machine, we can derive three scenarios from the common machine structure: 1) when M receives the children's done messages and the stopStaus is false (t1.2), it ends at normal final state s_{i+1}; 2) when M receives a stop message from its parent, it propagates the message to its children (ti.0) and update the stopStatus to true. Given the true value of stopStatus, M enters the abnormal final state s_1 after receiving its children's done messages (ti.1); 3) when M receives a fault from its children, it and forwards the message to its parent (ti.3), and follows the 2) scenario.

4.1.3 Synchronization of Activities and Dead-Path-Elimination

A set of *links* can be declared in the flow construct to express the synchronization 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 A and B are source and target activities of link1, respectively, link1 is A's outgoing link with *source* tag, and B's incoming link with *target* tag.

The synchronization between source and target activities is realized 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 either an AND or OR logical constraint on link values. 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 setLink messages to the target activity machines.

The *target* tag and *source* tag are standard elements of BPEL constructs, indicating each BPEL activity may or may not have incoming links and outgoing links. To capture this feature, we use a separate 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 4.3 shows the two types of linkWrapper machine structures, which covers the cases when an activity has target links and no target link. We use the type1

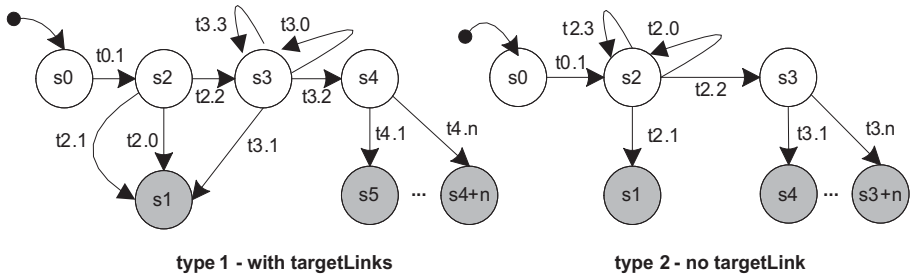

Figure 4.3: linkWrapper machine

linkWrapper structure as an illustration. Suppose B is the core machine and A is the linkWrapper machine for an activity, several scenarios can be derived from the machine structure.

First, a normal scenario follows the path $\langle t0.1, t2.2, t3.2, t4.x \rangle$. Machine A starts by receiving a start message from its parent (t0.1). It receives target link values from the link's source machine, and if the links' values satisfy the join-condition, it starts B (t2.2). A sends its done message to its parent after receiving B's done message (t3.2). If x and y are B's input and output, respectively, then A will send x to B and receive y from B. Thereafter, A sets the corresponding source link to true and the other source links to false, and sends the setlink messages to the links' target machines (4.x). Each outgoing transition of state s_4 corresponds to a source link.

Second, alternative scenarios can be: 1) A is interrupted by a stop message from its parent (t2.0); 2) when the join-condition is false, A updates all the source links to false and sends them to the links' target machines, and A ends abnormally (t2.1); 3) when A is interrupted by a stop message from its parent after B started, it stops B(t3.0). After receiving B's done message, A sends a done message to its parent (t3.1); 4) when receiving a fault from B, A forwards the fault to its parent (t3.3) and follows scenario

3).

4.1.4 Scoping, Compensation and Fault Handling

A scope has a primary activity that defines its normal behaviour, and it can optionally enclose *eventHandlers* (EHS), *faultHandlers* (FH), and *compensationHandler* (CH) activities. In a scope, EHS runs concurrently with a primary activity. The right of Fig 4.4 shows the hierarchy of invoking a CH. Only CH and FH are allowed to send a *compensate* message, and the target machine of this message must be a scope (scope2) immediately enclosed in the current scope (scope1). When receiving a compensation message, scope2 starts its CH to do the compensation activity.

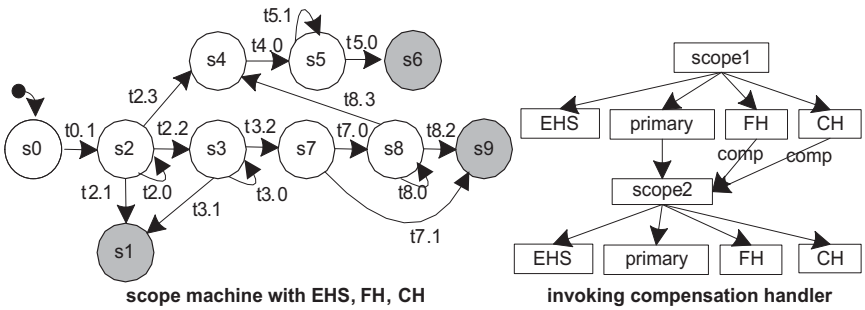

Figure 4.4: Scope and Compensation

The left of Fig 4.4 shows a scope machine with EHS, FH, and CH. When the scope is started, it starts the primary machine and EHS (t0.1), it waits for the primary machine to finish at state s3. When it receives primary machine's done message, it disables EHS (t2.2). The machine enters s7 when the EHS is done message (t3.2). If the scope does not receive a compensate message when its EHS ends at s7, the execution path is $\langle t0.1, t2.2, t3.2, t7.1\rangle$. Otherwise, the execution path is $\langle t0.1, t2.2, t3.2, t7.0, t8.2\rangle$. A scope's CH is available for invocation only when the scope completes normally. We model this by t7.0 and t8.2. When the scope receives a compensate message, it starts its CH (t7.0). The scope is done after the CH is done (t8.2).

Alternatively, when the scope receives a fault, it stops its children and starts the FH (t2.3). When the primary machine and the EHS finish, it enables the FH (t4.0). The

scope may receive a fault re-throw from the FH, and it forwards the fault to its parent (t5.1). The scope ends when the FH finishes (t5.0). When the scope is interrupted to stop, the implicit bpws:forceTermination faultHandler will run. For simplicity, we do not model this implicit faultHandler and instead we use the stop message to stop its children, and the common machine structure for BPEL structured activities is also used in scope.

4.1.5 Multiple Threads of Message 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 model this by associating the eventHandlers activity with EHS machine, each onMessage activity with a message event handler machine (MEH), each onAlarm activity with an alarm event handler machine (AEH), and each thread of onMessage activity with a MEH thread machine (MEHT). Each thread takes care of one message instance.

Fig 4.5 shows the EHS, MEH, and MEHT machines. For the EHS machine, we only show the normal scenario for simplicity. The parent of EHS is a scope machine and the children of EHS are MEHs and AEHs. When receiving a start message from the scope, EHS starts its children (t0.1). 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 children (t2.2). When EHS receives its children's done messages, it sends a done message to its parent (3.2). For 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 local variables *count* for the current thread number.

The normal scenario of MEH is $\langle t0.1, t2.1, t2.3, t3.0, t3.2\rangle$. When it receives a start message from the parent, the *count* is initiated to zero (t0.1). When it receives an external message, it increases the *count* by 1 and starts a new thread as a child machine (t2.1). The machines waits for an external message event to arrive at s2, a new thread

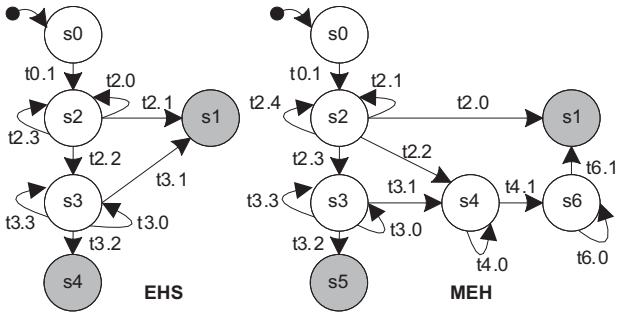

Figure 4.5: EHS and MEH machines

is created for each message instance (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 child machine to finish (t2.3). When one of its children finishes, it decreases the *count* by 1, until all the children finish (t3.0). When the *count* is zero, MEH ends at a normal final state (t3.2).

An alternative scenario is $\langle t0.1, t2.0\rangle$ where the machine is disabled before starting any thread. The other two scenarios contain transition sequence $\langle t4.0, t4.1, t6.0, t6.1\rangle$, when the machine is interrupted by a stop message (t2.2, t3.1), it stops its child machine one by one until all the children have been stopped (t4.0), then it enters state s6 to wait for its children to finish (t4.1). Similar to t3.0 and t3.2, when one of its children finishes, it decreases the *count* by 1 until all the children finish (t6.0). MEH ends at a normal final state (t6.1).

4.2 BPEL Data Flow

Data flow captures the relations between inputs and outputs of BPEL activities. 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 condition to fire particular links in the source element. So taking into account variables is essential in the formal model. There are two types of variables in BPEL: BPEL variables and links. BPEL variables are declared in the variables tag of either process or scope activity. Links are Boolean variables declared in the links tag of the flow activity. The

output link of an activity is defined as *true* if the activity completes normally, otherwise the link is defined *false*. The link synchronization feature will be covered in section 3.2. BPEL variables and links can be used and defined by the process or scope enclosed activities, and the flow enclosed activities, respectively.

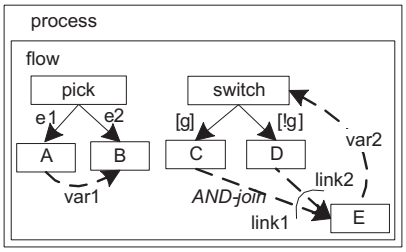

Figure 4.6: Unreachable and deadlock activities

Fig 4.6 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 and E are unreachable. B is unreachable 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 can never be executed. E is unreachable due to the fault design of links. In the *switch*, C and D are mutually exclusive in control flow, so the 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 *swtich* and E, which is caused by the cyclic data flow between them. On the 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 flow and data flow.

BPEL handles data by a *blackboard* approach, where data is shared by BPEL activities. By message passing, there are two possible ways to construct data flow from a BPEL model. One approach is to simulate the shared data access by adding data

writing to and reading from the *blackboard*. The other approach is to analyse the BPEL model to discover data dependencies among activities. The data flows identified from these two approaches are called centralized and decentralized data flow, respectively. Fig 4.7 shows the difference between them.

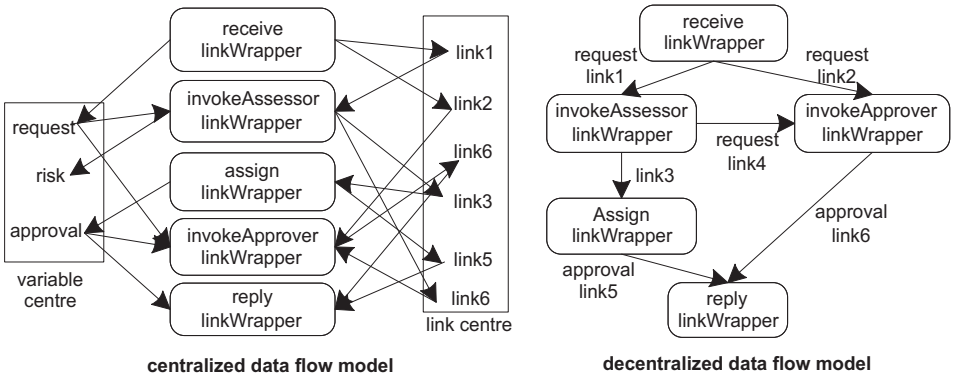

Figure 4.7: High level data flow models

In the centralized data-flow model, all the data exchanges between activities need to be via a variable or link centre. In the decentralized data-flow model, activities can exchange data directly. We choose the decentralized approach because it is more efficient in terms of data communication.

4.2.1 Data flow annotation in WSA

In WSA, a transition attribute named *status* indicates whether the transition is with $def(x)$ or $use(x)$. Let a message $msg(x)$ with msg as message name and x as message parameter, for a machine M, if $msg(x) \in E_M$, x is called an *input parameter* of M; if $m(x) \in O_M$, x is called an *output parameter* of M.

According to the definition of 3.1, in a machine M, a transition t is with $def(x)$ if 1) x is an input parameter of M; 2) x is assigned a value in an assignment. A transition t is with $use(x)$ if 1) x is an output parameter of M; 2) x is on the right of an assignment; 3) x is in a Boolean expression.

When more than one machine is considered, the boundary of system-under-test(SUT)

needs to be included. A component out of system boudary is called *tester* (or *environment*). Suppose machine M_1 sends a message $msg(x)$ to M_2:

1) If M_2 is chosen as the SUT, then M_1 is the tester of M_2 and x is the input parameter of M_2. In M_2, there exists a transition $\{t \in T_2 \,|msg(x) \in t.m\}$ with $def(x)$.

2) If M_1 is chosen as the SUT, then M_2 is the tester of M_1 and x is the output parameter of M_1. In M_1, there exists a transition $\{t \in T_1 \,|msg(x) \in (t.a \cap O_M)\}$ with $use(x)$.

3) If both M_1, M_2 are chosen as the SUT, then in the composite machine $M = M_1 \parallel M_2$, $msg(x) \in (E_1 \cap O_2)$ becomes an internal message of M. As a result, there is neither a transition $\{t \in T_2 \,|msg(x) \in t.m\}$ with $def(x)$ nor a transition $\{t \in T_1 \,|msg(x) \in (t.a \cap O_M)\}$ with $use(x)$.

In summary, for a transition $t \in T_M$, when t receives a message $msg(x)$ from a tester, t is annotated as $def(x)$; when t sends a message $msg'(x)$ to a tester, t is annotated as $use(x)$.

The BPEL data flows for a variable x can be constructed by identifying du-pairs (i.e. the transitions with $def(x)$ and $use(x)$), and checking whether there is a def-clear-path between the du-pairs.

Definition 4.1. 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) x is the output parameter of M_1 and the input parameter of M_2.

2) t_1 is with $def(x)$, i.e. either t receives a message with x as parameter from a tester, or x is on the left of an assignment in t.

3) t_2 is with $use(x)$, i.e. either t sends a message with x as parameter to a tester, x is on the right of an assignment in t, or x is in the Boolean expression in t.

4) There exists a def-clear path from t_1 to t_2 with respect to x.

Let P, Q be BPEL processes, the data flow is called BPEL internal data flow when $M_1, M_2 \in P$, while it is called BPEL external data flow when $M_1 \in P, M_2 \in Q$. Note that we do not construct the BPEL internal and external data flows explicitly, instead, we annotate the transitions with *status* to identify whether a variable is defined or used in it. The workload of finding def-clear-path is leaved to the model checkers. Let a data flow for variable x be $\langle M_i, .., M_j \rangle$, there exists a transition $t \in M_j$ annotated with $use(x)$, which indicates that x needs to be asserted as being defined previously. Details of how to use model checkers to find def-dlear-path and how to add these assertions for different model checkers are covered in (Zhe07).

4.2.2 Detailed Analysis of BPEL data handling

A BPEL process can be seen as a component. Each partnerLink declared in the BPEL process corresponds an external BPEL component. In Fig 4.8, the BPEL process of *loanapproval* example has three partner links, so the system has four components: loanapproval, customer, assessor, and approver.

```
<partnerLinks>
  <partnerLink name="customer" ..."/>
  <partnerLink name="approver" .../>
  <partnerLink name="assessor" ..."/>
</partnerLinks>
```

Figure 4.8: The loanapproval's partnerLinks

When more than a BPEL component are selected as SUT, an *external data-flow model* is used to capture how messages are transfered from one BPEL component to other BPEL components. When a single BPEL component is selected as SUT, an *internal data-flow model* is used to specify the relation between inputs and outputs of internal BPEL activities.

4.2.2.1 BPEL internal data exchange model

In a BPEL process, the BPEL variables and flow links are explictly declared. Fig 4.9 shows an example.

```
<variables>
  <variable name="request" messageType="loandef:creditInformationMessage"/>
  <variable name="riskAssessment" messageType="asns:riskAssessmentMessage"/>
  <variable name="approvalInfo" messageType="apns:approvalMessage"/>
</variables>  ...
<flow>
  <links>
    <link name="receive-to-assess"/> ...
  </links>  ...
</flow>
```

Figure 4.9: The BPEL variables

In a BPEL component P, let x be either BPEL variable or flow link explicitly declared in P, M_a be a BPEL activity and M be the associated WSA machine. When M_a receives a message $msg(x)$ from another BPEL component, or reads the x value, x will become an input parameter of machine M. When M_a sends a message $msg(x)$ to another BPEL component, or writes a value to x, x will become an output parameter of machine M. We analyse the data features of BPEL activities as the following.

- M_a receives a message $msg(x)$ from an external BPEL component. Each machine of activities *receive*, *invoke* with outVariable, *pick*, and *eventHandler* will contain a transition with $def(x)$ of BPEL variable x.

- M_a writes a value to x. The machine of the *assign* activity with $x$on the left of assigment expressions will contain a transition with $def(x)$ of BPEL variable x. The linkWrapper machine of an activity with source links has a transition with $def(x)$ of flow link x.

- M_a sends a message $msg(x)$ to an external BPEL component. Each machine of activities *invoke* and *reply* will contain a transition with $use(x)$ of BPEL variable x.

- M_a reads the x value. The machine of *assign* activity with x on the right of assigment expressions, and the machines of activities *while* and *switch* will each contain a transition with $use(x)$ of BPEL variable x. Also, the linkWrapper machine corresponding to an activity with target links has a transition with $use(x)$ of flow link x.

In a BPEL process P, the data can only exchange between two machines $M_1, M_2 \in P$ if the relation between the machine is either: 1) M_1, M_2 have a same parent machine, i.e. they are the same level machines; or 2) M_1, M_2 are parent and child.

An example of data exchange modelling in WSA is shown in Fig 4.10. The example also illustrates the reason to add the data exchange constraints.

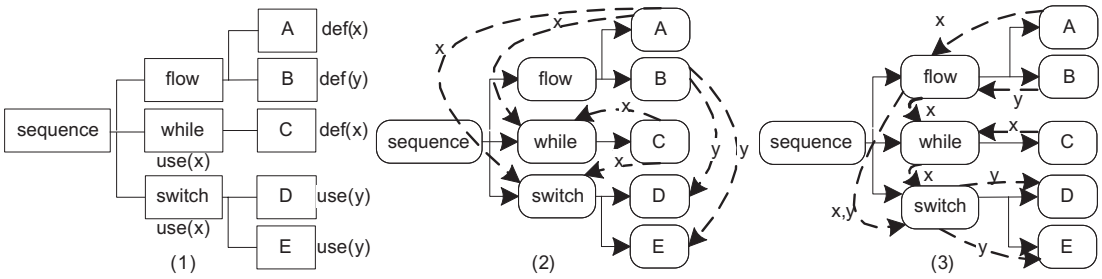

Figure 4.10: An example of BPEL internal data exchanges for variable x and y

The (1) of Fig 4.10 shows a BPEL process fragment, where the solid lines denote the enclosement relations between activities. The fragment 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. By analysis, the variable x is defined in A and C, and is used in *while* and *switch*; the variable y is defined in B, and is used in D and E.

The (2) and (3) of Fig 4.10 shows two ways of capturing data semantics of the BPEL fragment in (1). The dashed and solid lines denote the data exchanges and the control flows of machines, respectively. When a machine defines or uses a variable z, it will send or receive a message $setData(z)$, respectively. In (2), there is no constraint on data exchanges, instead the machine defining z will directly send message $setData(z)$ to the machine using z. Based on this approach, there exist two problems in the example.

- First, the *while* machine receives $setData(x)$ from A and uses x in its predicate. If the predicate is true, it starts the child C. C re-defines x and send $setData(x)$ to *while* to re-evaluate the predicate, also C sends $setData(x)$ to *swtich*. The loop continue until the predicate is false. Because C is in a while loop, everytime

C is executed, a message $setData(x)$ will send to *switch*. If the *while* loop n times, then the *swith* will receive $setData(x)$ message for n times.

- Second, in *switch* machine, either D or E will be executed but not both, so only one of the D, E needs to receive $setData(y)$. However, since B cannot decide the choice of *switch*, B will send two $setData(y)$ messages to D and E, respectively. If E is not execute, then B sends a message to a unreachable machine.

These problems can be solved by adding constraints on the data exchanges to flow between machines at the same levels or to flow between parent and child machines, shown in (3). First, C only sends $setData(x)$ to its parent *while*, and *while* will forward $setData(x)$ to the same level machine *switch*. Second, B sends the message $setData(y)$ to the parent *flow*, and *flow* fowards the message to the same level machine *switch*, which in turn fowards the message to either D or E. Comparing to the approach shown in (2), the approach of (3) to model data exchanges is clearer and more precise, but it requires additional message transfer.

When the loanapproval BPEL process is the SUT, the interal data exchange model of the loanapproval process is shown in Fig 4.11. The control flows are not shown in order to simplify the figure. As we discusssed in section xx, a BPEL activity with source or target links will associate with a linkWrapper machine and a core machine, where the linkWrapper will be the parent of the core. There are three scenarios in

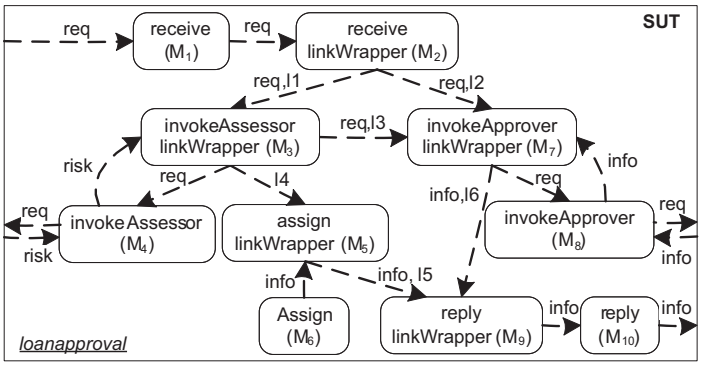

Figure 4.11: The internal data exchange model of the loanapproval process

Fig 4.11, we use one scenario as an illustration. For simplicity, a short machine name M_i is used. The M_1 machine receives a request req from the tester *customer*, it fowards $setData(req)$ to the parent machine M_2. The M_2 machine sets the values of links $l1, l2$ based on the req value. If link $l1$ is true, machine M_2 will send messages $setData(req)$ and $setLink(l1)$ to machine M_3, and send message $setLink(l2)$ to machine M_7. Since $l1$ is true, machine M_3 sends *start* message and fowards $setData(req)$ to machine M_4. The M_4 machine interacts with the tester *Assessor* and fowards $setData(risk)$ to machine M_3. The links $l3, l4$ are set based on the $risk$ value. If $l4$ is true, a message $setLink(l3)$ is sent to machine M_7, and a message $setLink(l4)$ is sent to machine M_5. After receiving $setData(info)$ from machine M_6, machine M_5 sets link $l5$ to true and sends $setData(info)$ and $setLink(l5)$ to machine M_9, which in turn sends *start* message and forwards $setData(info)$ to machine M_{10}. Finally, machine M_{10} returns a message with $info$ as parameter to the tester *customer*.

By model checking, we can derive the internal data flows for each BPEL variable and for each flow link.

- For BPEL variable req, $\langle M_1, M_2, M_3, M_4\rangle$,$\langle M_1, M_2, M_3, M_7, M_8\rangle$,$\langle M_1, M_2, M_7, M_8\rangle$.
- For BPEL variable $risk$, $\langle M_4, M_3\rangle$.
- For BPEL variable $info$, $\langle M_6, M_5, M_9, M_{10}\rangle$ and $\langle M_8, M_7, M_9, M_{10}\rangle$.
- For flow links $l1, l2, l3, l4, l5, l6$, their data flows are $\langle M_2, M_3\rangle$, $\langle M_2, M_7\rangle$, $\langle M_3, M_7\rangle$, $\langle M_3, M_5\rangle$, $\langle M_5, M_9\rangle$, and $\langle M_7, M9\rangle$,respectively.

Note that for a variable x, the $setData(x)$ or $setLink(x)$ messages are only added to capture the BPEL internal data exchanges. These additional messages are not used for the BPEL external data exchanges.

4.2.2.2 BPEL external data exchange model

Since web services communicate by exchanging messages, the BPEL external data exchange model is also a conversation model to show how BPEL models interact. If a

conversation protocol (e.g. WS-CDL [ ]) is available, such message flows can be can be explictly captured. For BPEL interactions, a top-down approach is to firstly design the conversation protocol to capture the iteractions between BPEL models, and then secondly design the BPEL models to fullfil the interactions. Thereafter, the BPEL models can be verified against the conersation protocol. A bottom-up approach is to design BPEL models, and verify the interactinos between them. [ ] has point out the advantage of the top-down approach over the bottom-up approach. However, when a conversation protocol is missing, it is especially important to verify the correctness of the BPEL model interactions. In our framework, we assume that there is no conversation protocol to guide the global interactions.

For the loan approval example, three partner links are included in the loanapproval BPEL process: *customer, assessor*, and *approver*. If the *customer* is selected as a tester, then the SUT contains three components: *loanapproval*, *assessor*, and *approver*. The BPEL external data exchange model can be easily constructed by identifying which partnerLink the message is sent to or received from, in a BPEL activity.

```
<invoke name="invokeAssessor" partnerLink="assessor"   portType="asns:riskAssessmentPT"
operation="check" inputVariable="request" outputVariable="riskAssessment"/>
```

Figure 4.12: The loanapproval's invoke activity

In the *invoke* activity example of Fig 4.12, the *check(request)* message is sent to the *assessor* component, and the message *check(riskAssessment)* is received from the *assessor* component. Correspondingly, there is a data exchange from for variable *req* from *loanapproval* to *assessor*, and a data exchange from for variable *risk* from *assessor* to *loanapproval*. Fig 4.13 shows the external data exchange model of the loan approval example.

In order to derive data flows for each data exchange among BPEL processes, we need to include both internal and external data exchange models. Fig 4.14 shows the itneral data exchange models of *assessor*, *loanapproval*, and *approver*, as well as the external data exchange model among them.

According to the model in Fig 4.14, model checkers can derive external data flows for

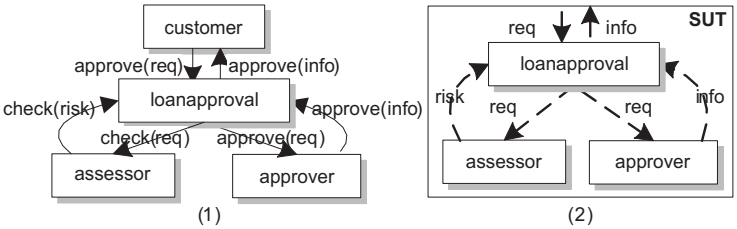

Figure 4.13: BPEL external data exchange model

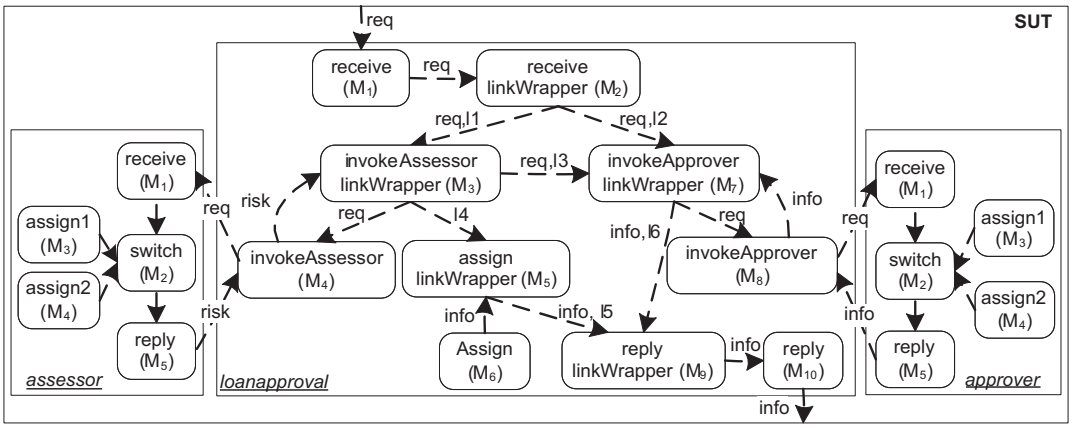

Figure 4.14: BPEL internal and external data exchange model

each data exchange among BPEL processes. $P.M_i$ to denote machine M_i of BPEL process P. We use S, L, A as shorthands for the BPEL processes *assessor,loanapproval,approver*, respectively.

For variable req, there exist three data flows:

- $\langle L.M_1, L.M_2, L.M_3, L.M_4, S.M_1, S_{M2}\rangle$,
- $\langle L.M_1, L.M_2, L.M_3, L.M_7, L.M_8, A.M_1, A.M_2\rangle$,
- $\langle L.M_1, L.M_2, L.M_7, L.M_8, A.M_1, A.M_2\rangle$.

For variable $risk$, there exist two data flows:

- $\langle S.M_3, S.M_2, S.M_5, L.M_4, L.M_3\rangle$ and $\langle S.M_4, S.M_2, S.M_5, L.M_4, L.M_3\rangle$.

For variable $info$, there exist two data flows:

- $\langle A.M_3, A.M_2, A.M_5, L.M_8, L.M_7, L.M_9, L.M_{10}\rangle$,
- $\langle A.M_4, A.M_2, A.M_5, L.M_8, L.M_7, L.M_9, L.M_{10}\rangle$.

4.3 Summary

We present Web Service Automata as a formal semantics for web services, and analyse various features of BPEL. The web service automaton is more general than the existing automata-based semantics in that it can model most features of BPEL and it allows verification of BPEL control and data flows.

Chapter 5

Verification and Test Case Generation of BPEL

As is well known, it is tedious, time-consuming and error prone to create test cases manually, especially for large and complex models. BPEL is a semi-formal flow-based language with complex features like activity hierarchy, concurrency, and dead-path-elimination. Hence, it is desirable to introduce automatic test case generation tools for BPEL. In order to verify BPEL rigorously, there are a number of proposals for applying model checking to verify BPEL, by transforming BPEL models into formal models such as process algebras, Petri nets, and automata (HS04). From the testing point of view, we use BPEL as the test model to derive test cases.

Based on our previously proposed web service automata (WSA) in section , this paper presents an automatic test case generation framework for BPEL. It is based on model checking and the test criteria are coverage oriented. Since NuSMV (CCGR99) and SPIN (Hol03) model checkers are already used on a regular basis for the verification of real-world applications, they are used as two alternative test generation engines in our framework. 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 links declared in BPEL models will be considered in data flow testing. Two-levels

of test cases will be generated in our framework. WSDL test cases are for unit testing to check the interface conformance between the implementation and the WSDL of individual services. BPEL test cases are for integration testing to check the behavioural conformance between the web service interactions and the various behavioural scenarios in BPEL models.

5.1 Model Checking in Testing

Model checking is a formal verification technique for determining whether a system model satisfies certain properties. Proposals of applying model checking in coverage-based testing were made in (HCL+03; HGW04; RH01). The idea is to use a model checker to find test cases by formulating test criteria as a *trap properties* to be verified. A trap property is the negation of the original property. The process consists of four steps. First, the design models are mapped to finite state automata suitable for model checkers. Second, the test criteria are encoded into temporal logic, such as CTL or LTL formulae. Third, a counterexample is generated if the model does not satisfy the temporal logic formula. A counterexample is an execution trace that will take the finite state model from its initial state to a state where the violation occurs. Finally, a test case can be retrieved from the counterexample. To achieve test coverage, the test criterion will be encoded into a set of trap properties, so that test cases satisfying the test criterion can be retrieved from a set of counterexamples.

5.2 Test Coverage Criteria in Temporal Logic

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. 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, each criteria includes $s \in S_{fMproc}$, which forces the model checkers to execute a BPEL model to its end.

Definition 3. Suppose a BPEL model is associated with a set of WSAs $\{M_1, .., M_n\}$, a **test case** of the BPEL model starts from the initial state of M_{proc}, and ends at one of the final states of M_{proc}.

Definition 4. A test suite is said to achieve **state coverage** of a set of WSAs $\{M_1, .., M_n\}$, if each state $s \in S_{Mi}$ and one of the final states $s_f \in S_{fMproc}$ can be executed at least once.

Definition 5. A test suite is said to achieve **transition coverage** of a set of WSAs $\{M_1, .., M_n\}$, if each transition $t \in T_{Mi}$ and one of the final states $s_f \in S_{fMproc}$ can be executed at least once.

Data flow testing is interesting because stimulating the sequences of operations which define and subsequently use variable values is an effective systematic method for exposing faults. For the *du-path coverage*, we adopt the definition from (RW85). According to definitions 1 and 2, given a variable v and a transition t, v is *def* in t if $v = def(exp)$ where $exp \in t.a$, v is *computation-use* in t if $v = cuses(exp)$ where $exp \in t.a$, and v is predicate-use in t if $v = puses(exp)$ where $exp \in t.g$. We use $d(v)$ and $u(v)$ to denote the sets of transitions where v is defined and computation-used (or predicate-used), respectively. A transition sequence $\langle t_1, t_2, .., t_n \rangle$ is a *def-clear-path* with respect to variable v if v is not defined in ti where $1 \leq i \leq n$. A *du-pair* of v is the transition pair (t_i, t_j) where v is defined in t_i and is computation-used or predicate-used in t_j. A *du-path* 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.

Note that we are only interested in the variables explicitly declared in BPEL models (denoted as V_{bpel}), and the data dependency between BPEL activities. So, 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 6. A test suite is said to achieve **all-du-path coverage** of a set of WSAs $\{M_1, .., M_n\}$, if for each $v \in V_{bpel}$, every du-path with respect to each $v \in V_{bpel}$ can be executed at least once, and one of the final states $s_f \in S_{fMproc}$ can be reached from each du-path.

Now we can encode the test coverage criteria into CTL and LTL temporal logic. CTL (Computation Tree Logic) views time as branching, so from a given branch alternative states may be reached. In the following, we use the temporal operators E (there exists some path), F (finally), X (next), and U (until). (HCL+03) gives a detailed study of encoding various structural test coverage criteria into CTL. Based on this work, the negation of state, transition, and all-du-path coverage criteria are encoded into the CTL of 1), 2), and 3) as follows. Here M is a web service automaton.

1) $\{\neg EF(s_i \wedge EF\ s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.

2) $\{\neg EF(t_i \wedge EF\ s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.

3) $\{\neg EF(t_i \wedge EX\ E\,[\neg d(v) U (t_j \wedge EF\ s_f)])\}$ where $v \in V_M, t_i \in d(v), t_j \in u(v), s_f \in S_{fMproc}$.

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. We use temporal operators [] (always), $\langle\rangle$ (eventually), X (next), and U (until). Suppose M is a web service automaton. The negation of state, transition, and all-du-path coverage criteria are encoded into the LTL of 1), 2), and 3) as follows.

1) $\{\neg \langle\rangle (s_i \wedge s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.

2) $\{\neg \langle\rangle (t_i \wedge s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.

3) $\{\neg \langle\rangle (t_i \wedge X(\neg d(v) U t_j) \wedge \langle\rangle s_f)\}$ where $v \in V_M, t_i \in d(v), t_j \in u(v), s_f \in S_{fMproc}$.

5.3 Test Generation Framework

In this section, we will elaborate the proposed BPEL test case generation framework, shown in Fig 5.1.

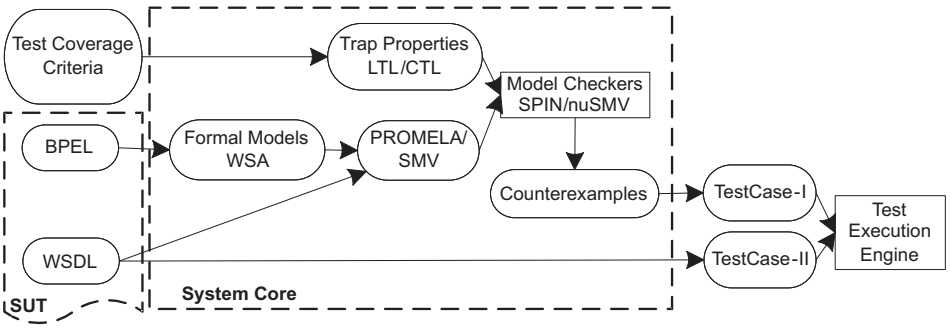

Figure 5.1: Test framework architecture

- From BPEL to formal model WSA

BPEL has complex features such as hierarchy, interruption, concurrency, synchronization, scoping, compensation, fault handling, and multi-threads handling. Each BPEL activity is mapped to a WSA, or a core WSA and a WSA for link handling. A BPEL structural machine can start and stop its enclosed machines. The propositional input events of WSA can capture various BPEL features. The logical-AND operator can capture the synchronization of *end*. For instance, a BPEL flow activity will not end until all its enclosed activities finished. In a flow machine, this feature can be captured by adding the logical-AND to the incoming *done* messages from its children. The logical-OR operator can model the fault propagation. In an activity, a fault is propagated as long as one of its enclosed activities raises a fault. The logical-AND together with logical-NOT can model priority BPEL messages such as termination or stop messages. BPEL data flow is analyzed explicitly, so that interactions of BPEL activities can be modelled by message passing. It is essential to provide an intermediate model between BPEL and model checkers. Without such layer, every model checker needs to consider how to model BPEL features in its input language, which compli-

cates the process. Instead, since BPEL features have been modelled in WSA, which is a Mealy-machine based model without hierarchy, WSA can be easily transformed to the automata-based input models of most model checkers. Fig 5.2 shows an example of the WSA code.

```
<WSA name="loanapproval" status="process">
  <data>
    <partner name="loanapproval_flow"/>
  </data>
  <state id="s0" status="initial">
    <transition id="t0.0" nextState="s1">
      <receiveEventList>
        <receiveEvent name="start" from_queue="adminQ"/>
      </receiveEventList>
      <guardList>
        <guard name="true"/>
      </guardList>
      <sendEvent name="start" partner="loanapproval_flow" to_queue="adminQ"/>
    </transition>
  </state>
  <state id="s1">
    <transition id="t1.1" nextState="s3">
      <receiveEventList>
        <receiveEvent name="done" partner="loanapproval_flow" from_queue="loanapproval_flow_doneQ"/>
      </receiveEventList>
      <guardList>
        <guard name="true"/>
      </guardList>
    </transition>
  </state>
  <state id="s3" status="final"/>
</WSA>
```

Figure 5.2: An example of WSA code

- From WSA to Promela or SMV models

Promela is the input language of SPIN model checker. It adapts the computational model of channel communicating finite-state automata with variables. A Promela model consists of a set of processes with keyword *proctype*, the channels with keyword *chan* can be declared globally or local to a process. Each process consists of a variable declaration part and a behavioural modelling part. The states, transition names (e.g. *ti*), and local variables of a WSA are captured in the variable declaration part. The transition relations are captured in the behavioural modelling part, which enclosed within a *do* loop. Since Promela supports message communication via channels, it is straightforward to transform WSA to Promela.

SMV is the input language of NuSMV. A SMV model is composed of a set of modules with keyword MODULE. Each module consists of two sections VAR and ASSIGN. The states, transition names, transition input events, transition output events, and local variables of a WSA are declared in the VAR section. The transition relations are captured in ASSIGN section. Since SMV language has no support for channels, the input queues of WSA need to be modelled explicitly. We model a queue for each message type as a SMV module, and the actual input queues are instantiated in the SMV module corresponding to a web service automaton. We implemented a queue structure that supports FIFO manipulation, however, the state space increases dramatically with such models. To reduce the state space, we implement a simple version of queue model holding only one message. Note that both Promela and SMV language does not support *String* type, so the values of string variable need to be enumerated in a user-define type section. The user-define type section is *mtype* in Promela, and *VAR* in SMV models.

- From WSDL to PROMELA or SMV models

Since the message type of BPEL variables are declared in associated WSDLs, we do not model such message types in WSA, rather, the required message types will be extracted from the corresponding WSDL and mapped to Promela and SMV models. In Promela, a message type in WSDL is declared as a complex type with keyword *typedef*. In SMV models, a message type in WSDL is declared as a complex type with keyword *MODULE*.

- Encoding Test Coverage Criteria into Trap Properties

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 *#define* to declare the elements required in the never claim. Fig 5.3 (a) shows

a never claim for covering a state and a final state of the process machine. $\langle\rangle\,(p \wedge q)$ 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 LTL state coverage definition in section 4, a set of the negation such LTL formula can provide state coverage of the BPEL model. Since SPIN can only verify one property in a run, SPIN need to run n times for n pairs of (p, q).

NuSMV model checker accepts either LTL or CTL formulas, a formula starts with the keyword *SPEC* in SMV models. Fig 5.3 (b) shows a CTL formula declaration for covering a state and a final state of the process machine. According to the CTL state coverage definition in section 4, a set of such CTL formula can provide state coverage of the BPEL model. Since NuSMV can verify more than a property in a run, SMV only needs to run once for a set of CTL formula.

(a)
```
#define p (loanapproval_flow_receive1:state == s2)
#define q (loanapproval:state == s3 || loanapprovalstate == s1)
never {    /* <>(p && q) */
T0_init:
        if
        :: ((p) && (q)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

(b)
```
SPEC !EF(loanapproval_flow_receive1.state = s2
 & EF loanapprovalstate = s3 | loanapprovalstate=s1)
```

Figure 5.3: An example of state coverage

- 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. This kind of test case checks whether the web service interactions conform to the communication protocols modelled in BPEL. A test case consists of a set of execution paths of the BPEL. For instance; a path representing the customer's request is less than 10000, the assessor service is invoked, and the returned risk is low. This execution path involves two services; loanapproval and assessor. If the loanapproval is selected as the service-under-test, the customer and the assessor will be two testers. In this case, the test case will remind users to input the right range values of request and risk.

- From WSDL to Test Cases

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.

- Execution of Test Cases

BPEL test cases will call the methods of WSDL test cases, so that the both dynamic interaction behaviour and static individual operation of remote web services can be tested. The two levels of test cases that are generated can run using the common JUnit test execution engine. A GUI is provide for users to input test data manually.

5.4 Symbolic Predicates

In a state machine with guards, the predicates of different paths need to be true alternatively, so that a model checker can be forced to explore alternative paths. For instance, $path_1$ and $path_2$ will be executed when $request.amount < 10000$ and $request.amount \geq 10000$ are true, respectively. Instead of inputing the actual parameter $request.amount$ with value less than 10000 (resp. greater than), we use a symbol

$pred_i$ to represent each predicate. *Gray code* (e.g. (Gra07)) on the predicates can be derived, where two successive values differ in only one digit, such that a model checker can explore all the paths. For instance, we use $pred_1$ and $pred_2$ to represent the above two predicates, the two-bit gray code matrix $(pred_1, pred2) : (0,0), (0,1), (1,1), (1,0)$. The values 1 and 0 denote boolean *true* and *false* respectively. For a state s_i, if t_{i0} and t_{i1} are the only two active transitions of s_i, where $pred_{i0} = t_{i0}.g, pred_{i1} = t_{i1}.g$, then the relationship $pred_{i0} \neq pred_{i1}$ can be derived. In this case, the combination $(0,0)$ and $(1,1)$ can be removed.

The corresponding Promela code is shown in Fig 5.4(a). First, each symbolic $pred_i$ is declared as a global boolean variable. A gray code matrix is constructed by inputting the predicate symbols. Second, list the predicate relationship, if there is any. These relationships will be read by an application to reduce the predicate combinations. Third, two processes will be inserted into the Promela model, a *runner* process assigns values of predicates based on the matrix from step 2), and a *chooser* process choose the current values of predicates. The *chooser* process will be started from the top level *init* process.

(a)

```
bool pred1 /* request.amount<10000 */
bool pred2 /* request.amount>=10000 */
/* Predicate Relationships
pred1 != pred2  */
proctype runner(byte type) {
 if
 :: type == 1-> atomic {pred1 = 1 ; pred2 = 0 ; }
 :: type == 2-> atomic {pred1 = 0 ; pred2 = 1 ; }
}
proctype chooser() {
  run runner(1) ;
  run runner(2) ;
}
```

(b)

```
--Predicate Relationships
--pred1 != pred2
MODULE runner
VAR
 pred1 : boolean;
 pred2 : boolean;
 i : 1..2;
ASSIGN
 next(pred1) := case
          i = 1      : 1;
          i = 2      : 0;     esac;
 next(pred2) := case
          i = 1      : 0;
          i = 2      : 1;     esac;
 next(i) := case
          i  = 2     : 1;
          1          : i + 1; esac;
FAIRNESS running
```

Figure 5.4: An example of predicate handling

Similarly, the SMV code is shown in Fig 5.4(b). A *chooser* module declares the predicates, shows the predicate relationship as comments. An application will read the predicates and generate gray code matrix, and reduce the matrix based on predicate relationship. Thereafter, *ASSIGN* section will be inserted into the SMV model, so that *chooser* can choose the current values of predicates. The *chooser* process will be started from the top level *main* module.

5.5 Summary

In this chapter, we presented an automatic test generation framework for BPEL, based on model checking and the state, transition, and du-path test coverage criteria. The generated BPEL based and WSDL based test cases can check the conformance of web service behaviour and interface, respectively. The proposed web service automata have been implemented in XML, and the transformation engines are implemented in XSLT. An Eclipse plug-in was developed in Java to invoke various transformation engines, to enable a user to choose the service under test (i.e. BPEL) and the test coverage criteria, and to interact with model checkers. The test framework is part of the *DBEStudio* deliverable for the EU project (DBE07). An extension of this work is to define additional test coverage criteria which are suitable for BPEL, i.e. criteria for integration testing, so that model checking can be used on scalable models.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We present Web Service Automata as a formal semantics for web services, and analyse various features of BPEL. The web service automaton is more general than the existing automata-based semantics in that it can model most features of BPEL and in particular it allows verification of both BPEL control and data flows. The proposed web service automata has been implemented in XML. Also, we implemented a front end mapping from BPEL to web service automata, and back end mappings from web service automata to the input languages of the NuSMV and SPIN model checkers. The transformation engines are implemented in XSLT.

Based on test coverage criteria such as state coverage and transition coverage for control flow, and du-path coverage for data flow, model checkers can generate counter-examples. The generated BPEL based and WSDL based test cases can check the conformance of web service behaviour and interfaces, respectively. An Eclipse plug-in was developed in Java to invoke various transformation engines, to enable a user to choose the service under test (i.e. BPEL) and the test coverage criteria, and to interact with model checkers. The test framework is part of the DBEStudio deliverable for the EU project (DBE07).

Case studies are ongoing to to evaluate the efficiency and salability of our test

generation framework. The results of theses evaluations will be included in an extended version of this report, which should be available in April 2007.

6.2 Future Work

An open issue is to ensure the correctness of formal models, i.e. preservation of the BPEL semantics. We can partially verify the formal models, by defining some BPEL features as system properties in temporal logic and model check the formal models against these properties. An extension of this work is to apply our formal model to a choreography language such as WS-CDL. We believe the same approach can be used due to the similar features of WS-CDL and BPEL. With the conversation protocol like WS-CDL, BPEL can be verified against WS-CDL to check whether the local behaviours of BPEL models conform to the global behaviours of WS-CDL. Another extension of this work is to define additional test coverage criteria which are suitabe for BPEL, i.e. criteria for integration testing, so that model checking can be used on scalable models.

References

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services version 1.1. May 2003. 4

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyser for mesage sequence charts. In *Proc. of TACAs*, pages 35–48. Springer-Verlag, 1996. 35

[APS05] Anupriya Ankolekar, Massimo Paolucci, and Katia P. Sycara. Towards a formal verification of owl-s process models. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 37–51. Springer, 2005. 21

[AST] Astro toolset. http://www.astroproject.org/. 16

[BB87] Ed Brinksma and Tommaso Bolognesi. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1), 1987. 13

[BP05a] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *Pro. of EUROMICRO*, pages 134–142. IEEE Computer Society, 2005. 18

[BP05b] Antonio Brogi and Razvan Popescu. Towards semi-automated workflow-

based aggregation of web services. In *ICSOC*, Lecture Notes in Computer Science, pages 214–227. Springer, 2005. 21

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983. 35

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *Pro. of CAV*, pages 495–499. Springer-Verlag, 1999. 43, 64

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. March 2001. 3

[CPN] Cpntools. http://wiki.daimi.au.dk/cpntools/. 12

[DBE07] Digital business ecosystem. http://www.digital-ecosystem.org, 2007. 74, 75

[DMCS05] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Trans. Softw. Eng.*, 31(12):1015–1027, 2005. 15, 21

[FBS05] Xiang Fu, Tevfik Bultan, and Jianwen Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005. 16, 17, 35

[Fer04] Andrea Ferrara. Web services:a process algebra approach. In *Proc. of ICSOC*, page 242. ACM Press, 2004. 14

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp - a protocol validation and verification toolbox. In *Proc. of CAV*, pages 437–440. Springer-Verlag, 1996. 14

REFERENCES

[FQH05] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl: an abstract grid workflow language. In *Proc. of CCGRID*, pages 676–685. IEEE Computer Society, 2005. 20

[FUMK03] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *Proc. of ASE*, page 152. IEEE Computer Society, 2003. 14

[FUMK04] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *Proc. of ICWS*, page 738. IEEE Computer Society, 2004. 15

[FUMK06] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based analysis of obligations in web service choreography. In *Proc. of AICT-ICIW*, page 149. IEEE Computer Society, 2006. 15

[GLSC06] Chunming Gao, Rongsheng Liu, Yan Song, and Huowang Chen. A model checking tool embedded into services composition environment. In *Proc. of GCC*, pages 355–362. IEEE Computer Society, 2006. 15

[Gra07] Gray code. http://en.wikipedia.org/wiki/Gray_code, 2007. 73

[HB03] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proc. of ADC*, pages 191–200. Australian Computer Society, Inc., 2003. 12

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *Proc. of ICSE*, pages 232–242. IEEE Computer Society, 2003. 19, 65, 67

[HGW04] Mats P. E. Heimdahl, Devaraj George, and Robert Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proc. of HASE*, pages 178–186. IEEE Computer Society, 2004. 19, 65

REFERENCES

[HM05] Reiko Heckel and Leonardo Mariani. Automatic conformance testing of web services. In *Proc. of FASE*, pages 34–48. Springer, 2005. 19

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 13

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. 43, 64

[HS04] Richard Hull and Jianwen Su. Tools for design of composite web services. In *Proc. of SIGMOD*, pages 958–961. ACM Press, 2004. 10, 64

[Jen97] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, volume 3*. Springer-Verlag New York, Inc., 1997. 12

[KBR04] Nickolaos Kavantzas, David Burdett, and Greg Ritzinger. Web services choreography description language version 1.0. April 2004. 3

[LMSW06] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting bpel processes. In *Proc. of BPM*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2006. 12

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. 13

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: state models and Java programs*. John Wiley and Sons, Inc., New York, NY, USA, 1999. 14

[ML06] Manuel Mazzara and Roberto Lucchi. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 2006. To appear. 15

[MM04] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 08(6):51–59, 2004. 10

REFERENCES

[MPT06] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Specifying data-flow requirements for the automated composition of web services. In *Proc. of SEFM*, pages 147–156. IEEE Computer Society, 2006. 21

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992. 13

[Nak05] Shin Nakajima. Lightweight formal analysis of web service flows. Technical report, National Institute of Informatics, 2005. Progress in Informatics. 17

[OMG06] OMG. Semantics of business vocabulary and business rules specification. October 2006. 3

[OvdAB+05] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in ws-bpel. In *BPM Center Report*. BPMcenter.org, 2005. 11, 12

[OWL04] Semantic markup for web services. `http://www.w3.org/Submission/OWL-S/`, 2004. 4

[PA03] C. Pautasso and G. Alonso. Visual composition of web services. In *Proc. of HCC*, pages 92–99. IEEE Computer Society, 2003. 19

[Pet62] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, 1962. 11

[PTBM05] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *Proc. of ICWS*, pages 293–301. IEEE Computer Society, 2005. 16, 21

[RH01] Sanjai Rayadurgam and Mats P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of ECBS*, page 0083. IEEE Computer Society, 2001. 19, 65

REFERENCES

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985. 66

[Sch00] Karsten Schmidt. Lola: A low level analyser. In *Proc. of ICATPN*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, 2000. 12

[SOA03] Simple object access protocol. http://www.w3.org/TR/soap12-part0/, 2003. 3

[SP06] Avik Sinha and Amit Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proc. of TAV-WEB*, pages 17–22. ACM Press, 2006. 19

[Sta05] Christian Stahl. A petri net semantics for bpel. Informatik-Berichte 188, Humboldt-Universitt zu Berlin, July 2005. 12

[UDD] Universal description, discovery and integration. http://www.uddi.org/. 2

[WFN04] Andreas Wombacher, Peter Fankhauser, and Erich Neuhold. Transforming bpel into annotated deterministic finite state automata for service discovery. In *Proc. of ICWS*, page 316. IEEE Computer Society, 2004. 16

[Wik07] Strict partial order, mutually exclusive. http://en.wikipedia.org/wiki, 2007. 38

[WPSW04] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. Notations for the specification and verification of composite web services. In *Proc. of EDOC*, pages 35–46. IEEE Computer Society, 2004. 15, 20

REFERENCES

[XLP06] Ke Xu, Ying Liu, and Geguang Pu. Formalization, verification and restructuring of bpel models with pi calculus and model checking. Technical report, IBM, 2006. 15

[YK04] Xiaochuan Yi and Krys J. Kochut. A cp-nets-based design and verification framework for web services composition. In *Proc. of ICWS*, pages 756–760. IEEE Computer Society, 2004. 13

[YTYL05] Y. Yang, Q. Tan, J. Yu, and F. Liu. Transformation bpel to cp-nets for verifying web services composition. In *Proc. of NWeSP*. IEEE Computer Society, 2005. 12

[Zhe07] Yongyan Zheng. *An Automatic Test Generation Framework for BPEL Web Services*. PhD thesis, Feb 2007. 44, 56