University of Alberta

A Dynamic Verification Platform for BPEL Environments

by

Qin Li 

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2007

Your file Votre référence
ISBN: 978-0-494-33296-2
Our file Notre référence
ISBN: 978-0-494-33296-2

Abstract

The Business Process Execution Language for Web Services (BPEL4WS or simply BPEL) is an xml based language, which is used to design business processes based on the interactions between Web services. Most previous work for BPEL testing is model checking related, and it focuses on the static behavior of BPEL. In this thesis, we present a runtime monitoring system for BPEL, which is composed of the trace specification language, the instrumentation system and the trace verification system. It targets the dynamic behavior of BPEL.

The discussion of the runtime monitoring system includes the following topics:

- The design methodology for the trace specification language.
- The implementation of the instrumentation system and the trace verification system.
- The generic representation of workflow patterns and fault models within workflow patterns.
- Two experiments designed to prove the coverage of the runtime monitoring system to workflow fault models.

Acknowledgements

I would like to especially thank my supervisor Dr. J. Miller for his tremendous contributions to my thesis, for his constant guidance, support and encouragement. I am grateful for his many useful comments on this work and for the many things that I have learned from him. I would also like to thank the schoolmates in the Department of Electrical and Computer Engineering, University of Alberta. When talking to them, I always obtained the good ideas and help.

I am also very thankful to my parents and my wife. Every time in my life, they are always the source of my power and courage. I would further like to thank all my friends in China, Canada and all around the world. Their friendship gives me so much happiness and I am deeply missing them.

Table of Contents

List of Tables

List of Figures

1 Web Service and BPEL4WS

1.1 Web Service

Browser-based, Web-accessible software applications have been playing a more and more important role in people's daily life. Using these applications, people can find almost all the information they want, and they can look for and buy almost anything online, from a book to a car. However, Web technology's success in business-to-consumer applications has not transferred to business-to-business applications. There remain several challenges regarding this application-to-application interaction. For example, applications are implemented in incompatible platforms and restrictions exist in access across organizational boundaries. Nowadays Web services have emerged to address these challenges. They provide a systematic and extensible framework for application-to-application interaction [GUN02]. Web services are Web applications that are published on to the web so that other Web applications can find and use them[JAI01]. They take the Web to its next stage of evolution, in which software components can find other suitable software components and bind with them to implement complete business transactions.

A typical Web Services framework consists of three components. They are service provider, service broker and service requester [JAI01]. Service providers create services and publish them to the outside world by registering them in the service broker. Service brokers act as a lookup service between the service provider and the service requester. Service requesters search in the service broker to find the location of desired services and then bind their application to the service provider to utilize these services.

Web services are essentially formed upon three major technologies: Simple Object Access Protocol [BOX00] (SOAP); Web Services Description Language [CHR01] (WSDL); Universal Description, Discovery and Integration (UDDI). SOAP provides a mechanism to communicate between Web services and client applications. It is an XML-based protocol and works on the existing transport protocols, such as HTTP, FTP and SMTP. WSDL is an XML language to describe the interfaces of Web services. Client applications rely on WSDL to bind themselves with the required Web services. UDDI specifies a mechanism to register and locate Web services. It represents the service broker in the Web services framework.

1.2 BPEL4WS

In many situations, one single web service is not sufficient to provide an effective service. For example, a travel agent provides the service of booking air tickets. Actually, the travel agent doesn't have its own information about the air ticket. It has to get that information from airline companies. So it has to combine its web service with web services from the airline companies. From this example, we find that there is strong requirement to build composite multiple web services as a single business process to provide more powerful services. To accomplish that goal, BEA, IBM, Microsoft and some other major software vendors introduced the Business Process Execution Language for Web Services (BPEL4WS) in July 2002. The most recent public release of its specification can be found in [IBM03].

BPEL4WS is a XML based language.It defines a model and grammar for describing the behaviour of a business process based on interactions between Web services [IBM03].

BPEL4WS includes several types of basic activities: invoking Web service operations, receiving and replying to requests, and assigning data to messages. These basic activities can be combined into structured activities using sequencing, paralleling, switch constructs, while loops and selective communication. Following paragraphs will discuss these basic and structured activities in more detail.

We present an example of BPEL4WS process for handling a purchase order [IBM03]. It starts with some definitions:

```
<process name="purchaseOrderProcess"
targetNamespace="http://acme.com/ws-bp/purchase"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:lns="http://manufacturing.org/wsdl/purchase">
<partnerLinks>
    <partnerLink name="purchasing"
        partnerLinkType="lns:purchasingLT"
        myRole="purchaseService"/>
    <partnerLink name="invoicing"
        partnerLinkType="lns:invoicingLT"
        myRole="invoiceRequester"
        partnerRole="invoiceService"/>
    <partnerLink name="shipping"
        partnerLinkType="lns:shippingLT"
        myRole="shippingRequester"
        partnerRole="shippingService"/>
    <partnerLink name="scheduling"
        partnerLinkType="lns:schedulingLT"
        partnerRole="schedulingService"/>
    </partnerLinks>
<variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice"
        messageType="lns:InvMessage"/>
    <variable name="POFault"
        messageType="lns:orderFaultType"/>
```

```
    <variable name="shippingRequest"
        messageType="lns:shippingRequestMessage"/>
    <variable name="shippingInfo"
        messageType="lns:shippingInfoMessage"/>
    <variable name="shippingSchedule"
        messageType="lns:scheduleMessage"/>
</variables>
```

A BPEL4WS specification normally contains Partner Link and Variable definitions. A partner link type characterizes each partner link and characterizes the conversational relationship between two partners by defining their "roles" in the conversation. Further, it specifies the portType provided by each partner for receiving messages within the context of the conversation [IBM03]. Variables provide the means for holding data that is exchanged between partners or data that is required for holding state related to the process. The type of each variable may be a WSDL message type, an XML Schema simple type or an XML Schema element [IBM03].

The most "meaningful" part of a BPEL4WS specification is the description of the process control logic. In this example, the following takes place. First, the purchase order is received. Then three tasks are launched concurrently: Initiating Price Calculation, Deciding on Shipper, and Initiating Production Scheduling. At last, when the three concurrent tasks are completed, the invoice processing is performed and the invoice is sent back to the customer.

```
<flow>
    <links>
        <link name="ship-to-invoice"/>
        <link name="ship-to-scheduling"/>
    </links>
    <sequence>
```

```
    <assign>
        <copy>
            <from variable="PO" part="customerInfo"/>
            <to variable="shippingRequest" part="customerInfo"/>
        </copy>
    </assign>
    <invoke partnerLink="shipping" portType="lns:shippingPT"
       operation="requestShipping" InputVariable="shippingRequest"
       outputVariable="shippingInfo">
        <source linkName="ship-to-invoice"/>
    </invoke>
   <receive partnerLink="shipping"
    portType="lns:shippingCallbackPT"
    operation="sendSchedule" variable="shippingSchedule">
        <source linkName="ship-to-scheduling"/>
   </receive>
</sequence>
<sequence>
    <invoke partnerLink="invoicing" portType="lns:computePricePT"
     operation="initiatePriceCalculation" inputVariable="PO">
    </invoke>
    <invoke partnerLink="invoicing" portType="lns:computePricePT"
     operation="sendShippingPrice" inputVariable="shippingInfo">
         <target linkName="ship-to-invoice"/>
    </invoke>
    <receive partnerLink="invoicing"
     portType="lns:invoiceCallbackPT"
     operation="sendInvoice" variable="Invoice"/>
</sequence>
<sequence>
    <invoke partnerLink="scheduling" portType="lns:schedulingPT"
    operation="requestProductionScheduling" inputVariable="PO">
    </invoke>
    <invoke partnerLink="scheduling" portType="lns:schedulingPT"
     operation="sendShippingSchedule"
    inputVariable="shippingSchedule">
        <target linkName="ship-to-scheduling"/>
    </invoke>
```

```
    </sequence>
</flow>
```

In BPEL4WS, an activity is the kernel, or core, element. The other elements can be considered as dependent upon the activity elements; they principally implement setting up an environment for the execution of the activities. As described earlier, BPEL4WS includes basic and structured activities. Basic activities include <receive>, <replay>, <invoke>, <assign>, <terminate>, <throw>, <wait> and <empty> [IBM03].

In these basic activities, there are three activities a business process can use for interacting with the outside world: <invoke>, <reply>, and <receive>.

The <receive> activity allows the business process to do a blocking wait for a specific incoming message. In this example, task Receiving Purchase Order is a <receive> activity, and this activity waits specifically for the incoming purchase order message. The <reply> activity allows the business process to send a message back to the partner from which it has already received a message through a <receive> activity. In this example, task Sending Invoice is a <reply> activity. The <invoke> activity enables the business process to make invocations to operations provided by partners. In this example, there are several <invoke> activities. These <invoke> activities are used to invoke operations provided by the external Web services, such as invoiceService, shippingSerivce and schedulingService.

The <assign> activity is used to copy data between variables. In this example, the value is copied from the "customInfo" part of variable PO to the "customInfo" part of variable shippingRequest through an <assign> activity. The <throw> activity is used to signal a certain kind of fault in the business process. Normally, there will also be a fault handler

element in the process to catch the thrown fault and use a <reply> activity to send partner information about the fault. The <terminate> activity is used to immediately abandon all the execution within the business process. The <wait> activity allows the business process to wait for a specific time period or until a certain deadline is reached. The <empty> activity does nothing. This activity is useful when we want to synchronize concurrent activities. In this example, <terminate>, <throw>, <wait> and <empty>activities are not applied.

As to the structured activities, they include <sequence>, <flow>, <switch>, <while>, <pick> and <scope> [IBM03]. These structured activities prescribe the order in which a collection of activities will take place. The <sequence> activity specifies a collection of activities to be performed sequentially in a lexical order. In this example, the three groups of activities, receiving, processing and responding, are contained inside a <sequence> activity, so they will be executed one after another. The <flow> activity provides the ability for activities to run in parallel. It also provides links for the synchronization of parallel activities. Every link has exactly one activity as its source and one activity as its target. If two activities are linked together then the target of the link can be executed only after the source activity has completed. The source activity may also have a transition condition through the trainsitionCondition attribute of its source element. The transition condition is a Boolean expression and it is evaluated when the source activity is completed. The status of the outgoing link of the source activity is determined by the evaluation result of the transition condition. In this example, there is a link between a requestShipping activity and a sendShippingPrice activity. The former activity is the

source of the link, and the later activity is the target of the link. There is no transition condition specified in the source activity, so it is deemed to be present with a default value of "true". Once a requestShipping activity is completed, the status of its outgoing link becomes true and it enables the sendShippingPrice activity to start.

The <switch> activity allows one branch of activity to be selected from a set of activities. This feature is much like the switch construct that occurs in many traditional programming languages. The <while> activity allows one activity to repeat until the specified Boolean condition no longer evaluates to true. The <pick> activity allows the business process to be blocked and wait for suitable events. An event could be a piece of message or a time-out alarm. Each event has an associated activity. When an event occurs, the associated activity will be performed. The <scope> activity allows the definition of a group of activities with their own associated variables and fault handlers.

1.3 Challenges on BPEL4WS Testing

With the popularity of Web services, organizations are increasingly using BPEL4WS for modelling business process within Web service architectures. On the other hand, BPEL4WS applications have been found to have extremely high quality requirements. They should exhibit "very high" quantities of reliability, flexibility, and security but also the "high" quantities of availability, maintainability and scalability. In order to achieve such demanding requirements, BPEL4WS applications need comprehensive testing.

However, for testers, testing BPEL4WS can be a huge challenge because a number of its special characteristics cause most traditional testing strategies, testing evaluation

standards and practical testing application techniques not to be suitable for BPEL4WS testing. Below we summarize some major reasons for this problem.

Challenges inherited from the testing of Web service

BPEL4WS processes are packaged as Web services, so all the characteristics of Web services are inherited by the BPEL4WS processes. Testing Web services is very difficult because of their characteristics. Firstly, Web services are distributed applications with numerous runtime behaviours. For distributed systems, many issues need to be verified, such as interoperability, timing, fault tolerance, availability, reconfiguration, reliability, security and performance [OFF02]. These issues make the testing of distributed applications highly complex.

Secondly, there are no user interfaces for Web services, and they are only accessible to software application through SOAP messages. So it is very difficult to observe the test result. It means that it is impossible for a tester to execute and test Web services in the same manner as traditional stand-alone software applications.

Thirdly, for a Web service user, normally only the WSDL specifications are available, hence only black-box testing is feasible because the design and implementation details of Web services are not available. Unfortunately, currently WSDL doesn't contain sufficient information for an application engineer to test Web services. For a standard WSDL specification, we can only derive information with regard to inputs, outputs, the types of inputs and outputs, the order of the inputs and outputs, and how the Web service should

be invoked. Information such as input/output dependency between Web services, and the invocation sequence cannot be obtained from the WSDL file [TSA02].

Finally, Web services are based on multiple standard protocols such as UDDI, WSDL, and SOAP. These protocols are used to publish, find and bind Web services. Therefore, in the lifecycle of a Web service, the error may occur in the Web service itself or in those supporting facilities.

Extremely large state space

There are two major factors that cause the huge state space of BPEL4WS processes. Firstly, BPEL4WS supports concurrent tasks through structured activity <flow>. These concurrent tasks can be combined into a large number of different sequences contributing to the large state space of the BPEL4WS process. Secondly, a BPEL4WS process can invoke an operation provided by other Web services. That invocation can be a synchronous request/response or an asynchronous one-way operation. If the invocation is an asynchronous operation, then it means that the response can return to the caller at anytime. The various returning times can place the BPEL4WS process into different states. Any increase in the number of asynchronous operations, will dramatically increase the size of the state space. For example, in [NAK02] there is a small size ticket order process. That process only involves three Web services, Traveller, Agent and Airline, and the transactions in each Web service are quite simple. However, the overall BPEL4WS process has 280 thousands states and approximately 470 thousands of transactions.

Multiple process instances

BEPL4WS supports a mechanism called "correlation" [IBM03]. This mechanism is BPEL4WS- unique and provides an application level mechanism to match the message and conversation with BPEL4WS process instances for which they are intended. For a BPEL4WS process, which involves various external Web services, implementing this mechanism is not trivial work. Consequently, in order to test the correctness of this mechanism, we not only need to simulate two or more clients and external Web services, but also we need to test the combination of messages from those clients and Web services.

No quality guarantee of external Web services

A BPEL4WS process can involve many external Web services. These Web services are implemented by different organizations or individuals, and they have different reliability characteristics. Therefore, in conversations between BPEL4WS processes and external Web services, various exceptions can happen, such as communication errors, missing messages, and baleful attacks. Implementing a comprehensive error handling mechanism for those exceptions is a challenge. Consequently, a thorough testing of that mechanism will be difficult to achieve.

Long running BPEL4WS processes

Many BPEL4WS processes are long running. During the long period of running, BPEL4WS processes and their context can heavily change. For instance, BPEL4WS

processes can bind to different services according to the context in which they are executed. The internal content of services may be modified by their providers. Furthermore, there may be new versions of selected services and new services provided by other entities. These changes might affect the correctness and quality level of BPEL4WS processes. However, traditional testing activities applied before the deployment of BPEL4WS processes can not foresee these changes.

1.4 Related Research on BPEL4WS Testing

Because BPEL4WS is a newly drafted standard, there has been limited research conducted for testing of BPEL4WS programs. Fortunately, research on testing workflow and business process has been conducted for several decades. BPEL4WS is a specific workflow language, which is to model business processes. So many other methods used to test workflow and business process can be inherited to test BPEL4WS.

UML activity diagrams are often used to model business processes. In [DUM01], they investigate the expressiveness and adequacy of the activity diagrams notation for workflow specification. Using a set of control-flow workflow patterns as their evaluation criteria, they demonstrated that activity diagrams support the majority of workflow patterns. Although UML is a widely used modelling language and UML activity diagrams have the ability to model business processes, they have a major drawback that they don't provide formal semantics necessary to perform analysis and verification.

Another approach is to model business processes using Petri nets. Van der Aalst [AAL00] used Workflow Petri nets (WF-Nets) to model business process definitions. In

his WF-Nets, activities are modeled as transitions, and dependencies are modeled by places with arcs. He developed a Petri-net-based workflow analyzer called Woflan. Woflan can partially verify the correctness of business processes definition by analyzing their soundness property. The soundness property includes three aspects. They are (1) It is possible to always terminate; (2) When a business process is terminated, there is no tokens left; (3) There is no dead activities. Soundness implies the absence of live-locks and deadlocks. However, soundness is only the minimal property that any workflow definition should satisfy. Application specific progress properties cannot be verified through WF-Nets. Another major deficiency is that there are three kinds of workflow patterns that WF-Nets have many limitations when representing. These three workflow patterns are (1) patterns involving multiple instances, (2) advanced synchronization patterns, and (3) cancellation patterns [AAL02].

Besides research conducted on the verification of general business processes, there are also researches specifically focused on verifying BPEL4WS. Foster et al [FOS03] proposed a formal approach to model and verify BPEL4WS using Finite State Processes (FSP). This approach has three major steps. In step1, they model the business process in the form of Message Sequence Charts (MSCs) with message sequence chart extensions of the LTSA tool. Then they use the LTSA tool to verify the behaviour of MSCs and translate the MSCs to FSPs. In step2, they implement the business process in BPEL4WS, and then translate the BPEL4WS implementation into FSP. In Step3, they check the business process by checking the trace equivalence of MSC FSP and BPEL4WS FSP. The major benefit of this method is its ability to verify the business process at the design

stage, allowing many problems to be found before the implementation. Another benefit is that it can guarantee the consistency between the design and implementation. However, there are also some serious limitations in this approach. Firstly, the ability to verify the MSCs is fairly basic. It only checks the reach-ability of the business process. Secondly, in order to check the trace equivalence of the MSC FSP and the BPEL4WS FSP, the activities specified in the BPEL4WS FSP must be manually re-labelled to match those specified in the MSC FSP. Furthermore, actions supported in BPEL4WS, but not supported in MSC, such as activities of assignments, switch conditions and initiators, must be manually hidden. It is obvious that these two manual tasks can very likely introduce faults.

Another notable work is verifying BPEL4WS, is using model checker SPIN. Nakajima [NAK02] is the first researcher that proposed to use the SPIN model checker to verify Web service flows. In his approach, processes are directly translated from WSFL (Web Service Flow Language), which is one of the BPEL4WS's predecessors, to the verification language Promela (a Process Meta Language). The Promela specification is then translated by SPIN to Buchi automaton to check the safety property of the business process. Further, SPIN allows the expression of application specific progress properties in terms of LTL (Linear Temporal Logic). The LTL formula is also translated into BÜchi automaton by SPIN, and it is checked against the target system automaton. Xiang Fu [FUX04] introduces an advanced method to verify business processes with SPIN. In his method, BPEL4WS specification is firstly translated into an intermediate representation Guarded Automata. Then these automata are translated into Promela. Because the guards

in the guarded automata are expressed as XPath expressions, it enables his approach to verify properties about XML data manipulation, while other model check approaches don't have this capability. Furthermore, it proposes the concept of synchronizability to tackle one inherent limitation of SPIN, which is we can only achieve partial verification by fixing the size of communication channels in Promela.

Comparing to model checking using WF-Nets, model checking using SPIN has a major advantage. SPIN enables us to verify the application specific progress properties, while WF-Nets can only be used to verify safety properties. Although model checking using SPIN can verify broader properties of business process, it still has a major shortage. It requires that the service links (channels) between Web services are predefined and established before the interaction starts. Therefore, some advanced features in BPEL4WS, for example, dynamically determining the target Web service, cannot be supported in current model.

Clearly, all of the approaches discussed above are model checking related. Because model checking is a very formal method, they all share one characteristic that they are not easy to understand and use by the average practitioner. Moreover, the large size of the explicit representation of the state space of most systems severely limits the size of systems that can be model checked. This is a problem referred to as the state space explosion problem. Although state reduction techniques have been proposed to reduce the state space explosion, fully describing and verifying a system is still extremely difficult. In addition, model checking conducts static checks of a business process, which may not

fully considered the value and timing characteristics of the process. It is generally agreed that the massive number of runtime interactions that connect various components is what makes web application reliability a challenging task. Similarly, for business processes composed of various Web services, the runtime interactions between those Web services are critical to the reliability of the business process. Hence we cannot ignore those significant factors and it is clearly of valuable to investigate verifying these business processes from the perspective of run time monitoring.

Recently, two approaches other than model checking, [MAY06] and [LIZ05], have been proposed to test BPEL4WS processes. [LIZ05] proposed a test framework to do unit testing on BPEL4WS processes. In that framework, there are a group of BPEL4WS test processes, which are BPEL4WS processes and are created solely for testing purpose. For each partner of the BPEL4WS process under test, one BPEL4WS test process is created to simulate behavior of this partner. Further, a central BPEL4WS test process is created to coordinate the other testing processes. Every BPEL4WS test process is aware of the correct data exchange between itself and the process under test, so that it can catch any failure on the interface between itself and the processes under test. However [LIZ05] doesn't describe how to define test cases and how test cases are implemented in BPEL4WS test processes. That missed information is critical to a BPEL4WS unit testing framework, so that it casts doubt on feasibility of this approach.

The major idea in [MAY06] is that based on the interface of a BPEL4WS process, a mock web service is created for each partner of the BPEl4WS process and mock Web

services are enabled to receive data from the BPEL4WS process and to feed data back. The test framework, which is used to implement that idea, consists of four layers. The structure of the test framework is outlined in figure1.1

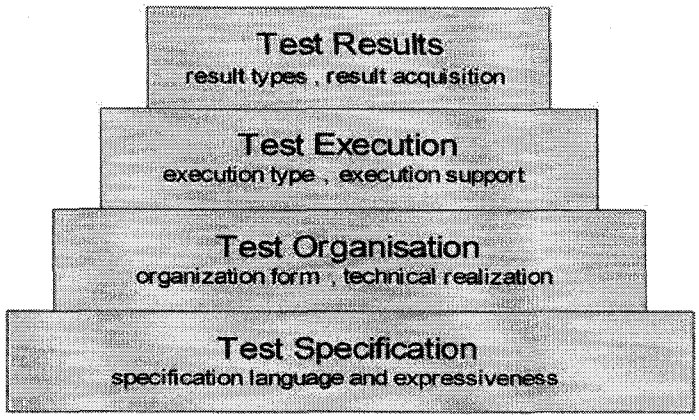


Figure 1.1 Test Framework of BPEL4WS

The Test Specification layer specifies what data should be received and be sent out from the perspective of mock Web services. The test organisation layer manages the test specifications and the deployment of BPEL4WS processes. The Test execution layer manages the execution of BPEL4WS process and partners. The Test Result layer deals with the test result and statistics.

[MAY06] is an efficient approach for unit testing of a BPEL4WS process since it can test all the interfaces of a BPEL4WS process. Additionally, it presents a complete structure for managing test cases, the deployment of BPEL4WS process and the lifecycle of partners. Those features are very useful in the real implementation. However, there are a couple of significant limitations in this approach. Firstly, the approach focuses on the interface of a BPEL4WS process and its internal content is ignored. Therefore, errors

which don't lead to failures on interface cannot be detected. In addition, a test case is specified from the perspective of the partner. Partner's conditional sending data can be described by logical expressions, but its conditional receiving data can not be described in this approach. The reason for this limitation is that the corresponding conditional sending data from the BPEL4WS process is performed in the process itself and there is no communication channel in this approach to allow the BPEL4WS process to pass the condition expression to its partners. Besides those limitations, [MAY06] doesn't provide information regarding how test cases are performed in partners. For instance, there is no illustration about how the time, when the partner sends out the data, is determined, although it is a very critical issue in implementing the test framework. Finally, there are so many features designed in the test framework, it makes the implementation of the test framework a challenging task. In addition, the test framework described in [MAY06] has not been implemented yet. All these factors make the approach far from convincing.

The methodology proposed in this thesis is developing a run time monitoring system for BPEL4WS processes. It can be viewed as an alternative of the early research; it intends to be simple and easy to apply in practice. It targets the dynamic behavior of a BPEL4WS program at runtime. A couple of sample applications demonstrate its effectiveness in finding bugs and its reasonable cost of application. In the following sections, we will discuss the details of our approach. Right before the end of our research, an approach for runtime monitoring of BPEL4WS processes is proposed in [BAR05]. The approach in it has some similarity with ours. For example, both do runtime monitoring of BPEL4WS processes. Both define a language to describe the monitoring rule and both instrument

code into initial BPEL4WS program according to monitoring rules. On the other hand, our approach is different from [BAR05] in some aspects. Firstly, it builds an external Web service as a monitoring manager. The extensive communication between the monitoring manager and the running BPEL4WS process heavily affects the performance of the running BPEL4WS process. In our approach, the trace verification system, corresponds to the monitoring manager in [BAR05], is embedded within BPEL4WS engine. Therefore, its effect on performance is much smaller. Secondly, it focuses on pre- and post-Conditions of external web services, while our approach focuses on running trace of BPEL4WS processes. Comparatively speaking, pre- and post-Conditions are easier to monitor. Our approach doesn't monitor those since we think trace behavior is more challenging and more important for a runtime monitoring system. With our approach, it is easy to add features to monitor pre- and post- conditions. Alternatively, the two systems could be combined to provide more defect coverage.

1.5 Overview and Contribution

1.5.1 Overview

In the following chapters, chapter 2 will introduce the general structure of a runtime monitoring system and what properties can be monitored by a runtime monitoring system. Then it will analyze what properties are critical to BPEL4WS systems and specify the properties of BPEL4WS systems that will be monitored in the proposed runtime monitoring system. Chapter 3 will describe the three subsystems which compose the proposed runtime monitoring system in details. The three subsystems are the Trace Verification Language, the Instrumentation system and the Trace Verification system.

Chapter 4 will analyze the eleven workflow patterns that directly supported in BPEL4WS systems and then propose the fault models for each workflow pattern. For each fault model, chapter 4 will also describe the mechanism of how the fault model is detected by the proposed runtime monitoring system. Chapter 5 will analyze the mutation process for each fault model from two perspectives. One is the perspective of trace specification and the other one is the perspective of implementation in BPEL4WS BNF. Chapter 6 there will be two experiments to evaluate the effectiveness of the proposed approach. Chapter 7 will give the conclusion of the research and point out the possible future research. In the appendix, there will be the BNF of the Trace Specification Language and the implementation details of the Instrumentation System and the Trace Verification System.

1.5.2 Contribution

This thesis presents a runtime monitoring system for dynamically monitoring the chronological behavior of BPEL4WS systems. The value of runtime monitoring system mainly comes from the long- running characteristic of BPEL4WS systems. Because of this characteristic, it is very valuable for the runtime monitoring system to quickly catch the problem when the problem happens during the execution of BPEL4WS systems. The earlier the problem is caught, the earlier the actions can be taken to minimize the lost.

The first subsystem of the proposed runtime monitoring system is the Trace Specification Language. This language is used to specify the chronological behavior of BEPL4WS systems. It inherits the basic concepts and grammar from the CSP [HOA04]. However, in order to make this language easy to use, the formal semantics from the CSP is hided. As a

result, this language is easy to apply in practice even for the business analyst who doesn't have extensive background on programming. On the other hand, there is no previous work on defining a language to specify the chronological behavior of BPEL4WS systems, so this research is the first work conducted in this area.

The second subsystem of the proposed runtime monitoring system is the Instrumentation System. This subsystem is used to automatically insert monitoring code into the original BPEL4WS processes. Implemented based on the most widely used parser generator JavaCC [JCC01], the instrumentation system demonstrated an efficient pattern which can be used to insert monitoring code based on specification automatically. Further, the design of monitoring code is creative. It doesn't have serious bad affect on the execution of original BPEL4WS processes since the dependencies in original BPEL4WS processes are not modified. Further, the monitoring code is designed to precisely collect the chronological information of BPEL4WS activities.

The third subsystem of the proposed runtime monitoring system is the Trace Verification System. This subsystem is implemented as a custom function of the BPEL4WS engine. Comparing to the other type of Trace Verification System implemented as an external Web service, this Trace Verification System takes much less time to exchange information between itself and the monitoring code. Therefore, the side effect of this Trace Verification System is much smaller. In addition, the algorithm, which is used in the Trace Verification System to determine the expected activity event based on the trace specification, is implemented based on the parse tree generator JJTree [JJT01]. This

algorithm is not complex but it is efficient and effective, so it provides a good example on applying JJTree to verify dynamic behavior against the static specification.

In order to prove the efficiency of the proposed runtime monitoring system, this thesis systematically analyzed the possible fault models within BPEL4WS systems. Since there is no previous work on this topic, we creatively analyzed the fault models from the perspective of workflow patterns. Petia Wohed [WOH02] stated in his work that workflow patterns precisely represent the control dependencies in the workflow modelling. In addition, Petia Wohed investigated whether a workflow pattern is supported in BPEL4WS and how the workflow pattern is implemented in BPEL4WS. Based on Petia Wohed's work, this thesis proposed a few fault models for each workflow pattern supported in BPEL4WS. Since there is no practical BPEL4WS system used in the industry field, we couldn't get any practical experience on the fault models within BPEL4WS systems. Further, without practical BPEL4WS systems, we cannot evaluate our proposed fault models against them. Although the fault models proposed in the thesis might be incomplete or inaccurate, its contribution lies in that it is the first attempt in this field, and it can be used as the initial empirical data.

After we analyzed the fault models for each workflow pattern, we defined mutation process for each fault model in a generic format. A mutation processes describes how a fault model is generated, and it is analyzed from two perspectives. One is the perspective of trace specification, and the other one is the perspective of implementation in BPEL4WS BNF. The generic format of the mutation processes can prove the feasibility

of the fault models. In addition, the mutation processes can be used to help the automatically generating the mutants of BPEL4WS processes. Before defining the mutation processes, we discussed the size of fault. A mutant is achieved by injecting a small fault into the original BPEL4WS process. However no previous work provides precise definition on the size of fault, therefore there was no specification on how small a fault should be. In this research we proposed a definition for the size of fault within the filed of BPEL4WS for the first time. This definition can be used as an initial reference to the definitions in other fields.

2 Runtime software monitoring system

2.1 Over view

A software runtime monitoring system is a software system that observes the behaviour of other software systems and determines if it is consistent with a given specification. Monitoring is concerned with actual transitions between states, not possible transitions, so it takes an executing software system and a specification of the software properties and checks that the execution meets the properties.

Generally, there are two types of properties: safety and temporal. Safety properties include, for example, invariants, properties that define a sequence of events, properties that check values of variables, and properties that deal with resource allocation. The other temporal category includes properties such as progress and bounded liveness as well as timing properties [DEL04]. There is no distinct border between these two categories. For

example, property "Event A occurs within 10 seconds" can be classified as a safety property, whilst it also can be classified as a timing property. Generally, a property often has the form μ->α, where μ is some condition on Σ that identifies the states in which α must hold. This means that in any state where μ is true inΣ, then α must also be true. If α evaluates to false, then the current execution has reached a disallowed state [DEL04].

A runtime monitoring system is composed of three parts: a specification language, a monitor and an event-handler. A specification language is a language used to specify requirements of the problem domain and other properties associated with behaviour of the system. The specification language may be any form of algebra, automata, logic or even a high-level language. The high level language can be a functional, object-oriented, or imperative language. Algebra, automata and logic are formal languages, so they have advantages of being easy to verify. On the other hand, it is easy to construct a high-level language to define properties, but it cannot be verified easily. Normally, we need a mapping or translation mechanism to map the high-level property specifications into the low-level system activities.

A monitor consists of two parts: an observer and an analyzer. The observer is a set of program fragments that are inserted into the target program to instrument the system. It keeps track of changes of monitored objects. When the observer finds the relevant state information according to the specification of properties, it passes that information to the analyzer. The recognized state information is often supplied with a timestamp that can be used in checking timing properties. When implementing an observer, this first step is to

determine the monitoring points. Monitoring points are points in the target program where the execution of monitoring code will be initiated. After determining the monitoring points, we utilize pre-existed tools or develop our own tools to insert monitoring code automatically. Normally, the instrumentation is performed statically. This means that it is performed before executing the target program. In special cases, this process can be conducted dynamically. For example, a monitoring system uses pre and post conditions to specify properties upon entry and exit of a module [DEL04]. The analyzer checks the correctness of the state by comparing the actual state with the expected state of the system.

Monitors observe the behaviour of target programs. When a violation against the specified property is found, a monitor can take appropriate action. The event handler refers to how the monitor reacts. The response action can modify the state space of the executing application, report application behaviour or initiate another process of application.

2.2 Run time monitoring of a BPEL4WS process

As stated in section 2.1, a run time monitoring system can check various safety and temporal properties. Hence, before we design our specification language, we should investigate what properties we are going to specify with that language.

2.2.1 Property of deadlock free

Firstly, we analyze what security properties should be considered. Obviously, the basic security property of a BPEL4WS process is deadlock free. The reason is that if a deadlock happens in a BPEL4WS process, then the process cannot proceed so that it

cannot perform any other properties. Currently, various model-checking methods used to verify BPEL4WS processes are mainly used to check the property of deadlock free of BPEL4WS processes. That trend also indicates that deadlock free is the first property a BPEL4WS system should possess. Although all the BPEL4WS processes require this property, we don't need to specify it through the specification language explicitly. The reason for this decision is that deadlock free is a generic property for all the processes rather than a property for a specific process and it is not valuable to repeatedly specify this property for every process. We treat it as an implicit property within a specification. It means every specification of properties specifies a deadlock free property, although it is not declared explicitly. If the running of a BPEL4WS process satisfies the property specification, then it has the property of deadlock free. We achieve this implicit property through another security property, sequence of events.

2.2.2 Other security properties

Besides the deadlock free property, according to [DEL04] we need to consider other security properties such as invariants, properties that define a sequence of events, properties that check values of variables and properties that deal with source allocation.

2.2.2.1 Properties that check values of variables

The property of invariant is a condition that does not change or should not change if the system works correctly. In general object oriented programming there are two types of invariants, class invariants and loop invariants. A class invariant is an invariant used to constrain objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object [INV01]. In BPEL4WS, although there is no concept of class, but we can easily map this concept to process invariant.

Correspondingly, process invariants are used to constrain states in an instance of business process. In general object oriented programming, a loop invariant is an invariant used to prove properties of loops. In BPEL4WS, there is also loop structure, so that loop invariant can be used directly to specify the property of loops in BPEL4WS. When we take a close look at both invariants, we find that they both are Boolean expressions built from the values of the variables. In [DEL04], it mentioned the property that checks the value of variables. Since the property of invariants actually is also a specification about the value of variables, therefore we decide to combine these two properties into one property. We still call it property that checks the value of variables.

2.2.2.2 **Properties defining a sequence of events**

When we talk about the properties that define a sequence of events in the field of BPEL4WS, we mean the properties that define a sequence of activity events. We use activity events to represent all the events in BPEL4WS, since activity is the kernel, or core element in BPEL4WS. The importance of activity in BPEL4WS was discussed in section 1.2. We think that the sequence of activity events is a critical security property of BPEL4WS processes. In this section, we will explain why it is so important. Firstly, the importance of the sequence of events in Java application was emphasized in [BRU02] and [MUO02]. Secondly, we can analyze the importance form the purpose of BPEL4WS. BPEL4WS emerged as a standard of Web service orchestration [PEL03] [THA03]. Web service orchestration is about providing an open, standard-based approach for connecting Web services together to create high-level business processes. Being a standard of Web service orchestration, BPEL4WS aims to describe how web services can interact with each other at the message level, including the business logic and execution order of the

interactions [PEL03]. From this description, we can see that BPEL4WS mainly defines two things. One is business logic, which can be various business activities, such as sending a purchasing order or receiving a receipt. The other one is the execution order of the interactions. In the field of BPEL4WS, the interactions between Web services are implemented as activities. Therefore, the execution order of the interactions is the execution order of the activities. When we analyze the relationship between basic and structured activities provided by BPEL4WS, it is easy to find out that most structured activities are used to regulate the execution order of basic activities. For example, <sequence> activity is to ensure the activities included in it execute in a sequence order; <flow> activity is to put the execution of its sub activities into a parallel order. Obviously, that relationship proves the description of BPEL4WS in [PEL03], which says that one of the two major goals of BPEL4WS is to describe the execution order of the interactions. From the previous statements, we can conclude that the execution order of the activities is critical in BPEL4WS. From the perspective of run time monitoring, the execution of an activity can be characterized by the start and end event of the activity. Therefore the concept of execution order of the activities can be translated into order of activity events, and the order or sequence of the activity events is critical to the run time monitoring of BPEL4WS.

A real business process example can further prove the importance of the sequence of activities in a business process. The example is an oversimplified business process for employee travel arrangements: The client invokes the business process, specifying the name of the employee, the destination, the departure date, and the return date. The BPEL

business process first checks the employee travel status, assuming that a Web service exists through which such checks can be made. Then the BPEL process will check the price for the flight ticket with two airlines: American Airlines and Delta Airlines. Again assume that both airline companies provide a Web service through which such checks can be made. Finally, the BPEL process will select the lower price and return the travel plan to the client [ORA01]. The business processes itself and the interactions between the business process and the external environment are depicted in figure 2.1. In the figure 2.1, the middle part describes the business process itself, and the left part describes the interactions between the client and the business process, and the right part describes the interactions between the business process and the external Web services. If we focus on the business process itself, we can find that it is composed of several ovals, each of which represents an activity in the business process. Those ovals are connected by the arrowed lines. The direction of the arrows defines the sequence of the activities. For example, the arrow between the activity "Retrieve employee travel status" and activity "Get plane ticket offer from American airline" points to the latter activity from the former activity. In this case, the direction of arrow indicates that the latter activity cannot execute until the former activity is completed. From an overall view, it is these arrowed lines that connect activities to achieve the business process. If the order that is indicated by the arrowed lines is modified or broken, then the business process cannot perform correctly. Even the result might be a disaster. In this sample process, for instance, if the relationship between activity "Get plane ticket offer from Delta Airlines" and activity "Get plane ticket offer form American Airline" changes from parallel to sequence, which is that the output line of activity "Get plane ticket offer from American Airlines" becomes the input

line of activity "Get plane ticket offer from Delta Airlines" and its original input line is deleted, then the business process will become fragile. It can only perform correctly when both activities work well. If any activity doesn't work, then the whole business process will get into a deadlock state. However, if those two activities are in parallel order, one activity failing won't block the whole process.

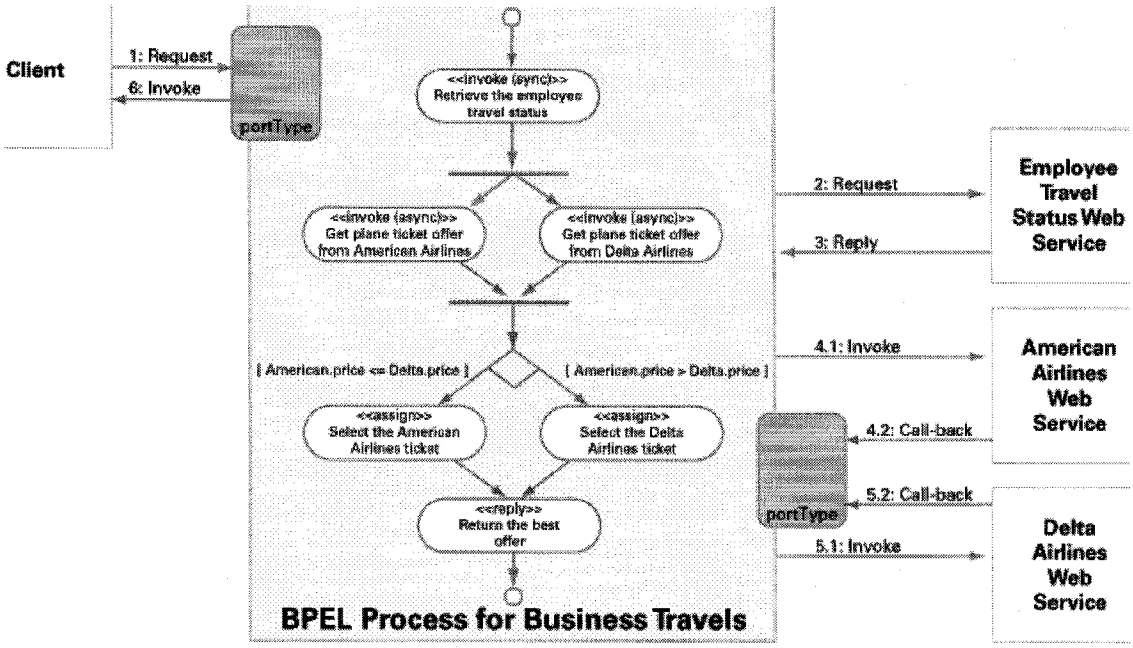


Figure 2.1 Process of employee travel arrangement

2.2.3 Properties mostly studied in this research

Based on the previous analysis of the properties that should be checked in the run time monitoring system of BPEL4WS, we conclude that our runtime monitoring system of BPEL4WS should provide facilities to explicitly define the property that check the value of variables and property that defines a sequence of events. When take a close look at these two properties, we find that in many cases the sequence of events is determined by the value of variables, therefore when we verify the sequence of events, we have to take into account the value of variables. For example, the transitionCondtion attribute in a

source element of link is a Boolean expression composed of variables. The value of the transitionCondition attribute is determined by the values of those variables and it determines whether the target element of the link can be executed. Furthermore, the condition attribute in a <while> activity, which is also a Boolean expression composed of variables, determines whether the <while> activity can repeat itself. Considering the importance of sequence of events to BPELWS as well as the relationship between these two properties, for the sake of simplicity, in this research we only focus on how to verify the sequence of events and the variables related to the sequence of events.

3 Test BPEL4WS

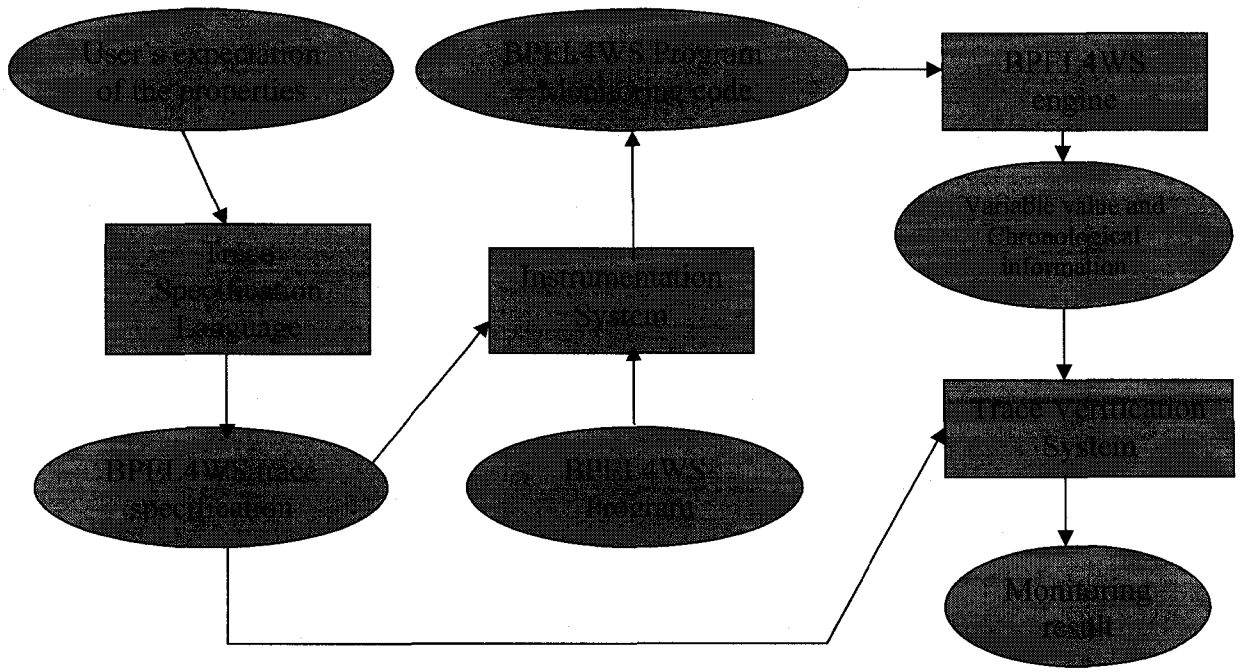
3.1 Overview

Our run time monitoring system has three major components. The first component is the Trace Specification Language. This language is designed to be easy to use. Although, it takes its fundamental model from a CSP[HOA04]]-based representation of the BPEL4WS language, the framework is designed to hide the formal semantics, allowing "ordinary users" without a CSP background to easily use it. This language can be used to specify the dynamic behavior of a BPEL4WS program, where dynamic behavior is explained as the trace of a BPEL4WS program. We define the BPEL4WS trace specification after requirement analysis of the system and before writing the BPEL4WS program. It is not valuable to specify the BPEL4WS trace based on the BPEL4WS program, because in that way the testing facility mimics the BPEL system under test.

The second component is the Instrumentation System. This component uses the BPEL4WS trace specification to insert monitoring code into a BPEL4WS program. The monitoring code satisfies the grammatical requirements of the BPEL4WS language, and it collects variables and chronological information of activities and sends them to the final component of our testing framework.

The third component is the Trace Verification System. It parses the BPEL4WS trace specification and finds out the chronological order between activities. Then it derives the actual executing order of activities from the chronological information of the activities, which is passed by the instrumented monitoring code. Then it compares the actual order of the activities with their expected order. From the comparison of these results, we can achieve the verification of a BPEL4WS system with regard to its chronological properties.

The overall structure of our run time monitoring system is depicted in figure3.1.



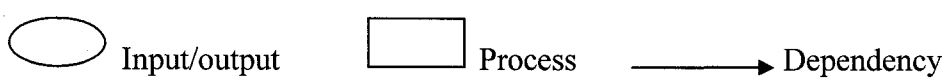


Figure 3.1 Structure of runtime monitoring system

3.2 Specification language

The design of our specification language was inspired by CSP, Communicating Sequential Processes, which is well known process algebra [HOA04]. CSP provides a set of constructs, which can model systems that stretch beyond what is construct-able using the BPEL4WS language. Hence, we have performed some minor modifications to make CSP more precisely fit the descriptive needs of a BPEL4WS program. Since we use our CSP dialect in the context of BPEL4WS, we call it CSP $_{BPEL}$. In the following sections, firstly we will introduce the basic concepts of CSP, and then we will describe how we tailor the CSP to fully align it with theBPEL4WS language.

3.2.1 Introduction to CSP

CSP was one of the first process algebras, which was developed by C. A. R. Hoare at University of Oxford in the1980s. CSP allows the description of systems in terms of component processes that operate independently, and interact with each other through events. There are two primitive concepts in CSP, event and process. Events represent communications or interactions. They may be atomic, or they can have associated data. For example, on and off are atomic events, while out!5, in?x respectively represent events that output the value 5 and input a value represented by x. The syntax of the event can be described by the following grammar:

event = a | m?x | n!y |√

Where 'a' is the atomic event, m and n are the input and output channels respectively, and x and y are values. '√' denotes the successful event [GAN02]

A process describes a behavior of pattern for an object. A process engages events. The collection of events in which a process P has the potential to engage in is called its alphabet, denoted by αP.

3.2.1.1 **STOP**

STOP is the simplest process, which doesn't engage in any events. It is used to describe the behavior of a broken object. A broken object is an object that is equipped with the capabilities to engage in the events of its alphabet, but never exercises those capabilities. Objects with different alphabets are distinguished, even if they never do anything.

3.2.1.2 **SKIP**

SKIP is defined as a process that does nothing but terminates successfully. SKIP is a very useful process in CSP. It is often put after an event to construct a syntactically correct process. This usage will be explained in the section 3.2.1.3.

3.2.1.3 **Prefix**

CSP processes are built up from events and other processes using a number of operators. The first one is prefix, denoted as→. The prefix operator combines an event and a process into a new process. It stipulates that in the new process, the event and the original process should occur in sequence. Let x be an event and let P be a process. Then x→P describes an object which first engages in the event x and then behaves exactly as

described by P. The prefix operator always takes a process on the right and a single event on the left. If P and Q are two processes, then it is syntactically invalid to write P→Q. Similarly, if x and y are two events, it is syntactically invalid to write x→y. In order to describe a process that includes two events x and y occurring in sequence, we can make use of the basic process SKIP to construct a syntactically correct process like: x→(y→SKIP).

3.2.1.4 **Recursion**

There is no recursion operator in the CSP, but recursion is a very important definition in CSP. With a sole prefix definition, we would be able to define processes that engage in a finite sequence of events. However, the sole prefix definition is not able to describe a repeatable or an infinite process. Fortunately, the recursion definition provides this ability, so processes that run forever can be described with the help of a recursive definition. For example, considering the simplest possible everlasting object, a clock which never does anything but tick, can be described by the following process:

CLOCK= tick→CLOCK

The meaning of this process is substituting every occurrence of CLOCK on the right hand side by its definition.

CLOCK= tick→CLOCK
=tick→(tick→CLOCK)
= tick→ (tick→(tick→CLOCK))
=.....

3.2.1.5 Choice

It is necessary for a process to behave conditionally. Such as, under some situations, it behaves in one way, while in other situations it behaves differently.

CSP provides two operators to handle conditional behaviors. One is external or deterministic choice, the other one is internal or nondeterministic choice.

Deterministic choice

Deterministic choice allows the future progress of a process to be defined as a choice among a range of component processes, and allows the environment to determine the choice by communicating an initial event for one of the processes [CSP01]. For example, $(a \rightarrow P) \,\Box (b \rightarrow Q)$ is the process that will first communicate the initial events a and b, and then behaves as either process P or Q depending on which initial event the environment chooses to communicate. There are two special cases that need to be stated. If event a and event b are identical, then the choice between processes P and Q is nondeterministic. Additionally, if the event a and event b were communicated simultaneously, the choice would be resolved nondeterministically.

Nondeterministic choice

Nondeterministic choice allows the future progress of a process to be defined as a choice among a range of component processes, but the environment does not have any ability to influence the selection between alternatives [CSP01]. For example, the process $(a \rightarrow P) \sqcap (b \rightarrow Q)$ can behave like either $a \rightarrow P$ or $b \rightarrow Q$.

3.2.1.6 Sequencing

Sometimes, we would like to define a process where after the component process P finishes, component process Q executes. With the sequencing operator, this process can

be easliy accomplished as P; Q, where ";" is the denotation for the sequencing operator. The sequencing operator is similar to the prefix operator. Both place a list of elements into a sequence. The distinct difference is that the sequencing operator combines two processes, while the prefix operator connects an event and a process.

3.2.1.7 **Concurrency**

The concurrency operator || is used to specify that processes are running in parallel. When two processes are brought together to evolve concurrently, the usual intention is that they will interact with each other. Those interactions are performed through events that they engage in. There is a basic rule regarding the interaction between parallel processes. The rule is that parallel processes can only interact by synchronously engaging in the events that exist in the interaction of the alphabets of the processes. According to the rule, if an event e is in the alphabet of two parallel processes P1 and P2, then P1 can only engage in the event e if P2 can also do so. The following example can appropriately explain how the concurrency operator works.

$P= (a \rightarrow P) \sqcap (b \rightarrow P)$
$Q=b \rightarrow Q$
$R=P||Q$

The Process R is the parallel combination of processes P and Q. Suppose we perform the combination by executing the interaction between P and Q. As the result, we can describe process R as a simple process without a concurrency operator.

In this example, when P internally chooses to do a, since Q can only apply b, this will lead to a deadlock situation. When P chooses b, $R= (b \rightarrow P) || (b \rightarrow Q) = b \rightarrow (P||Q)$. Thus

$R= (b \rightarrow R) \sqcap STOP$

3.2.1.8 **Interleaving**

As stated previously, the concurrency operator requires processes to simultaneously engage with the events in their common alphabets. However, sometimes, we might want to make sure that parallel processes don't interact at all. CSP provides an interleaving operator ||| to achieve this. The interleaving operator represents completely independent concurrent processes. The process P|||Q behaves as both P and Q simultaneously. The events from both processes are arbitrarily interleaved in time

3.2.1.9 **Hiding**

In some cases, we may want to conceal some events of a process from other processes. After the events are hidden, they are not observable or controllable to the other processes. The reason of hiding might be a privacy or security concerns. CSP has a hiding operator to achieve this feature. The notation of this operator is \. For example,

The behavior of a smoker is described by the process

Smoker = (light→smoke→stubout→Smoker)

□(nofuel→refill→light→smoke→stubout→Smoker)

Suppose that smoker is intelligent and he wants to hide the business of refilling the lighter, then using hiding operator to hide that refilling event. So the new behavior of the smoker becomes

Smoker\refill = (light→smoke→stubout→Smoker)

□(nofuel→light→smoke→stubout→Smoker)

3.2.2 Design of CSP_{BPEL}

3.2.2.1 **Process**

In CSP, a process is the behavior of an object. In CSP_{BPEL}, we define the trace of a BPEL4WS program as a process. This process describes the behavior of a BPEL4WS program from the perspective of a running trace. In this dissertation, we introduce a term 'Trace specification'. In the context of run time monitoring system, it is the trace requirements of the whole monitored BPEL4WS program. In the context of CSP_{BPEL}, it is a process, and it specifies the expected trace behavior of the BPEL4WS program. A simple trace specification looks like receiveOrder→processOrder→sendReceipt→STOP. In the context of CSP_{BPEL}, the term 'process' has a special meaning and must not be mixed with the business processes in the context of BPEL4WS.

3.2.2.2 **Events**

The core element in the BPEL4WS program is the activity, including basic and structured variations. The behavior of the BPEL4WS program is mainly the composition behavior of those activities. Therefore, when we consider the events within a BPEL4WS program, what we are interested in are the events of the activities. Since our CSP_{BPEL} language is mainly used to specify the trace of a BPEL4WS program, we focus on when activities start and end. The internal details of any activity are not our concern. In CSP_{BPEL}, an event has the following formula

$$Event:: = Ident_A.b \mid Ident_A.e \mid Ident_A$$

$Ident_A$ is the identifier of the activity. In the formula of event, 'b' represents the beginning of the activity, and 'e' represents the end of the activity. $Ident_A$ is the abbreviation of

IdentA.b→Ident$_A$.e .In order to map the trace specification to a BPEL4WS program, we require that the identifier of activity (Ident$_A$) in the CSP$_{BPEL}$ to be the same as the activity name in BPEL4WS program. With this mapping, we are able to translate the trace specification to the chronological requirement of activities in BPEL4WS programs. For example, if there is a trace specification ReceiveOrder→ProcessOrder→SendRecipt→STOP, then in its corresponding BPEL4WS program, the name of the activity, which corresponds to event ReceiveOrder, has to be ReceiveOrder. The following BPEL4WS code snippet shows how to specify the name of a <receive> activity to "ReceiveOrder".

```
<receive createInstance="yes" name="ReceiveOrder" operation="approve"
 partnerLink="customer" portType="apns:order ApprovalPT"
 variable="request"/>
```

In a BPEL trace specification, we can use the same event in more than one place. Although it looks like that more than one occurrence of the event is specified, actually those events are the same event and only one occurrence of the event is expected. This approach is very useful to specify traces for complex processes. In section 5.3.1, there is a sample trace specification showing that an event of activity is specified twice. If we want to specify multiple occurrences of an activity, we can use the WHILE loop which will be illustrated in section 3.2.2.7.2.

3.2.2.3 **Alphabet**

As we mentioned before, in CSP the collection of events in which a process P has the potential to engage in is called its alphabet. For a trace specification, its alphabet is the collection of activities specified in the specification. The concept of alphabet is not

important in CSP_{BPEL}, since we mostly focus on is the sequence of events rather than the interaction of component processes. Since we are not interested in the interaction of component processes, there would be very little value to consider how the interactions affect the alphabet of processes. Hence, we just need to know the alphabet of the overall process is the collection of defined activities. In the implementation of our CSP_{BPEL}, we don't explicitly specify the alphabet, because it is indicated implicitly by the trace specification itself.

3.2.2.4 **SKIP and STOP**

In CSP_{BPEL}, we inherit basic processes SKIP and STOP from CSP

3.2.2.5 **Sequence**

In CSP, there are two sequence related operators. One is the prefix operator, and the other one is sequencing operator. The former one is used to combine an event and a process, and the later one is used to combine two processes. In CSP_{BPEL}, we combine these two operators into one operator. We still call it the sequencing operator, and it is denoted as→. We choose → as the denotation of the operator because generally in a specification language, the sequence relationship is denoted as→. Such as, Jass [FIS00] and BPE-Calculus [KOS03] .There are two reasons why we combine those two operators. Firstly they both represent sequence relationships. This is the precondition for the combination. If two operators have quite different meanings, it makes no sense to combine them. Secondly, the difference between the meanings of those two original operators is trivial from the perspective of trace monitoring. Our CSP_{BPEL} aims to specify the trace behavior of a BPEL4WS program. Considering two processes A and B,

$A = a \rightarrow P$

$B = (a \rightarrow SKIP); P$

Obviously, events in the processes A and B will execute in the same trace, since the basic process SKIP doesn't do anything but terminate successfully. That is, for a trace monitor, processes A and B behave the same.

Consequently, in CSP_{BPEL} the sequencing operator $\rightarrow$ is used to specify a process that runs immediately after the completion of an event, allowing it can specify a process doesn't executes until the other process finish. Syntactically, $a \rightarrow P$ and $A \rightarrow P$ are both correct. For example, we want to use CSP_{BPEL} to specify the trace of a simple BPEL4WS program, which only has two basic activities ReceiveRequest and ProcessRequest, which are engaged in a <sequence> activity. If Activity ReceiveRequest is put before activity ProcessRequest, the trace specification will be ReceiveRquest→ ProcessRequest→SKIP. In contrast, if the Activity ReceiveRequest is put after activity ProcessRequest, the trace specification will be ProcessRequest→ ReceiveRquest →SKIP.

3.2.2.6 **Concurrency**

Standard CSP provides two concurrency related operators, the concurrency operator || and the interleaving operator |||. They both connect two parallel processes. Operator || concentrates on the interaction between the processes, and requires that processes engage in the same event simultaneously. On the other hand, the operator ||| specifies two processes running in parallel without interaction.

In CSP_{BPEL}, we eliminate the concurrency operator from CSP, since in CSP_{BPEL}, when we monitor the running trace of two parallel processes, only the information telling us that they are really running in parallel is interesting to us. The interaction between the processes cannot expose whether they are running in parallel or not. The concurrency operator focuses on the interaction, so it is not inherited in CSP_{BPEL}.

On the other hand, we have to point out that the interaction between the processes might affect the internal running trace of one process. For example, processes X and Y are running in parallel,

$X= (a \rightarrow P) \Box (b \rightarrow Q)$

$Y=c \rightarrow d \rightarrow STOP$

It might be the case that event c or d in the process Y determines either event a or b is engaged in the process X. Hence, process Y affects the internal running trace of process X. Although the interaction is able to affect the internal trace of a process, for those two parallel processes, their external trace behavior is not affected at all.

In CSP_{BPEL}, we inherit the interleaving operator ||| from CSP. This operator provides the ability to specify that two processes run concurrently, while their interactions are not considered. However, we only inherit the meaning of interleaving operator. We change its name from interleaving operator to concurrency operator and we also change its notation from ||| to ||, because in specification languages, it is more common to use term concurrency and notation ||. Consequently, in CSP_{BPEL} the concurrency operator || is used to specify that a process runs concurrently with another process. In BPEL4WS, only the

structured activity <flow> enables two activities included by it to run in parallel. To describe the trace of two activities in a <flow> activity, we can easily use the trace specification (activityA→SKIP) || (activityB→SKIP)

3.2.2.7 **Condition**

In CSP, the deterministic choice and nondeterministic operators offer the ability to specify the conditional behavior of a process. We have to have an operator to represent conditional behavior in $CSP_{BPEL,}$ since in BPEL4WS conditional behavior is very common. For example, activity <switch>, <pick> and Element < transitionCondition> are used to define conditional behavior. When we analyze the conditional behavior, we find that in CSP, the choice between the component processes is resolved by external or internal events. While in BPEL4WS, the choice is resolved by the value of variables. Therefore, in order to specify a conditional behavior in $CSP_{BPEL,}$ we introduce the mechanism of variables in CSP_{BPEL}. The token to represent a variable is BPEL_VARIABLE. Since in BPEL4WS, a variable is composed of one or more parts, so that the variable token in CSP_{BPEL} has a format of $Ident_{variable}.Ident_{part.}$ The token is composed of two parts, which are separated by a period. The part before the period maps to the variable name, and the part after the period maps to the part name of that variable.

In CSP_{BPEL}, basic mathematical calculations and logical expression evaluation (such as 'add', 'minus', 'time', 'divide', 'equal', 'not equal', 'bigger than', 'smaller than', 'and', 'or') are supported, therefore variables with these mathematical facilities can be used to define simple or complex conditions. For example, a simple condition could be

request.amonout >1000

A complex condition could be

order1.amount+order2.amount+order3.amount<=balance

It is not necessary to declare variables before we use them, however, only those variables that are already declared in the BPEL4WS program or will be declared can be used. This means that if we have implemented the BPEL4WS program before defining the trace specification, then the identifier of variables in CSP $_{BPEL}$ should follow the name in the BPEL4WS program. On the contrast, if a trace specification is defined before implementing the BPEL4WS program, then the identifier of variables in it should follow that in the trace specification. The reason for this rule is that we have to map them so that the CSP_{BPEL} condition can be translated to the condition that can be understood by the BPEL4WS. With the facility of conditions, we are able to describe the behavior of a conditional process. In $CSP_{BPEL,}$ we support two types of conditional processes: branch and loop.

3.2.2.7.1 **Branch**

In order to describe a branch process, we introduce the keywords IF ELSE. The meaning of these keywords is the same as that in many programming languages. These two keywords and conditions, together with the component processes, are able to define a complete condition process. The syntax of the basic branch process is

IF (condition) {ProcessA} ELSE {ProcessB}

This basic branch process describes that if the condition is satisfied, the component process ProcessA executes, and if the condition is not satisfied, then component process ProcessB executes. Variations on the basic branch process are also allowed. For example,

IF (condition) {ProcessA} is syntactically correct in CSP_{BPEL}. It specifies that if the condition is satisfied, then the component process ProcessA executes, and if the condition is not satisfied, no action will be conducted.

A condition process, as a whole, can be combined with an event or other processes through sequence and concurrency operators. For example,

ReceiveRquest→IF (request.amount>1000) {LoanAssess→SKIP} ELSE {LoanApprove→SKIP}

is a valid trace specification. It specifies that firstly activity ReceiveRequest take places, and then a conditional process is performed. In the condition process, the value of the variable request.amonout determines the value of the condition. If the value is greater than 1000, then activity LoanAsses will run otherwise activity LoanApprove will run.

We have another example to illustrate how condition processes efficiently describes the conditional behavior in a BPEL4WS program. The link construct is very important in BPEL4WS, and it is used to express synchronization dependencies. The source element of a link construct has an attribute, transitionCondition. A sample transitionCondition is

transitionConditon="bpws:getVarialbeData('request','amount')<10000".

This attribute is a Boolean expression. The value of the Boolean expression determines whether the target element of the link construct can be executed or not. Hence, the trace behavior of the link construct can de described as

sourceEvent→IF(transitionCondition) {targetEvent→SKIP}

3.2.2.7.2 **Loop**

In $CSP_{BPEL,}$ a loop process aims to describe the repeatable behavior of a component process under a certain condition. We introduce the keyword WHILE to help define a loop process. The syntax of a loop process is

WHILE (conditionExpression) {component process}

The conditionExpression part, a Boolean expression, specifies a precondition for the execution of a component Process. The conditionExpression has the same format as the condition in the branch process. Before the component process starts running, the conditionExpression is validated. If the conditionExpression is validated to be true, then the component Process will execute. After completing the execution of the component process, the conditionExpression is validated again. If the conditionExpression is still validated to be true, then the component process executes again, so on and so forth. Whenever, the conditionExpression is checked to be false, the component process will stop performing.

A loop process is different from a branch process. In a branch process, the maximum running times of a component process is once, and the condition is validated only once. While in a loop process, the component process is able to execute many times, as long as the condition is satisfied, and the condition can also be validated more than once.

In CSP, a recursion definition can be used to define a repeatable or infinite process. However, the recursion definition can only apply to an overall process, and it cannot describe the behavior of a component process. Let's take a look at a recursion definition in CSP,

CLOCK= tick→CLOCK

It is easy to find out that the identifier of the overall process CLOCK exists in the right-hand side of the process equation. It means that process CLOCK is repeating itself. Furthermore, we can find that there is no condition in this definition. Since in a BPEL4WS program, there is no facility for the whole process to repeat itself, a recursion definition is not applicable in CSP_{BPEL}.

In a BPEL4WS program, only activity <while> supports repeatable performance of a specified activity. The trace behavior of a <while> activity can be easily described by a loop process. For example, a <while> activity in a BPEL4WS program is

```
<while condition =" bpws:getVarialbeData('exp,'request') = 'yes'">
<sequence>
    <receive partner="caller"
      name ="receive1"
      operation ="echo"
      variable = "request" />
      <reply partner="caller"
      name="reply1"
      operation="echo"
      variable="request" />
 </sequence>
 </while>
```

Its trace behavior can de described as a CSP_{BPEL} trace specification

WHILE (exp.request='yes') {receive1→reply1→SKIP}

Similarly as branch process, a loop process can also be combined with other processes with an event or other processes through sequence and concurrency operators.

3.2.3 BNF of CSP_{BPEL}

The BNF [BNF01] of CSP_{BPEL} is designed upon the standardized EBNF notation rather than upon a formal BNF. Therefore it has many human readable and non-formal syntax

rules and extensions. For example, in the BNF of CSP_{BPEL}, terminals are not strictly enclosed within quotation marks ("..." or '...'). A terminating character usually a semicolon is not used to mark the end of a rule. In addition, regular expression wild-cards such as * and + are used to represent the repetition of items.

The BNF of CSP_{BPEL} provides adequate tokens to represent all trace related operators, which define the trace relationships between events and processes, and between processes. For example, token "->" represents Sequence Operator, token "||" represents Parallel Operator [JAS01]. With these tokens, we can specify highly complex running trace of BPEL4WS systems. Besides the trace relationship, another primary element in trace specification is event. In the BNF of CSP_{BPEL}, we use a nonterminal token <BPELEVENT> to represent an event in BPEL4WS system. The definition of token <BPELEVNET> is <IDENTIFIER>|<IDENTIFER>".b"|<IDENTIFIER>".e". Nonterminal <IDENTIFER> defines the composition of an identifier, which is the same as the identifier in other programming language. In CSP_{BPEL}, the value of <IDENTIFIER> is the name of a BPEL4WS activity. <IDENTIFIER> means that the event of activity ending following the event of activity beginning, while <IDENTIFIER>.b means the event of activity beginning and <IDENTIFIER>.e means the event of activity ending.

Variables in BPEL4WS program can be represented by nonterminal <BPELVAR>. The definition of < BPELVAR> is <IDENTIFIER>.<IDENTIFIER>. The former <IDENTIFIER> corresponds to the variable name of a variable in BPEL4WS program,

while the latter <IDENTIFIER> corresponds to the part name of a variable in BPEL4WS program.

Like the BNF of other programming language, the BNF of CSP $_{BPEL}$ provides basic features of a programming language. It can skip blank space, and it allows one line and multiple lines of comments. It supports basic mathematical calculations (such as "+", "–", "*", "/") and logical expression evaluation (such as AND, OR, ">", ">=", "<", "==", "<>").Furthermore, it supports basic flow control operations (such as If Else, While). The flow control operations are key elements that enable us to define complex running trace. The complete specification of the BNF of CSP_{BPEL} is listed in appendix A. More details regarding the implementation of CSP_{BPEL} can be found in appendix B

3.3 Instrumentation system

The instrumentation system is used to insert monitoring code into a BPEL4WS program. In order to add monitoring code, first we need to design the monitoring code to be added. In our research, the principle feature of monitoring code is to collect variable information and chronological information of activities, and send that information to the trace verification system. So we would have two major types of monitoring code. One is to collect variable information, and the other one is to collect chronological information. All the details regarding what is monitoring code and how the monitoring code is inserted is described in appendix C.

3.4 Trace verification system

The trace verification system is used to verify if the actual execution order of activities satisfies the expected behaviour specified via the trace specification. The function of this

system is implemented through a two step approach. The first step is parsing the trace specification and creating a parse tree to represent the expected order of activities. The second step is to determine what activity event should happen at which time based upon the parse tree and the variable values received from the running BPEL4WS process. Subsequently, we compare the actual activity event with the expected activity event. In appendix D, we illustrate how the trace verification system is implemented in details.

4 The efficiency of our runtime monitoring system

For our runtime monitoring system, we want to use a mechanism to evaluate its efficiency. Wohed [WOH02] has analyzes the workflow patterns supported in BPEL4WS programs. These patterns capture typical control flow dependencies encountered in workflow modeling. That means workflow patterns cover the behavior patterns of BPEL4WS programs. Therefore, if we can prove that CSP_{BPEL} can describe the trace behavior of each workflow pattern and the trace verification system can catch the failure in the workflow patterns, then we prove that our runtime monitoring system is effective at monitoring BPEL4WS programs. In the following sections, we will discuss how our runtime monitoring system works for each workflow pattern.

4.1 Sequence

The Sequence pattern describes that an activity in a workflow process is enabled after the completion of another activity in the same process [WOH02]. There are two possible programming solutions for this pattern. One is using a structured activity <sequence> (see Listing4.1), the other one is using a link construct (see Listing 4.2).

Listing 4.1
<sequence>

Listing 4.2
<flow>

```
  activityA                          <links>
  activityB                            <link name="L"/>
</sequence>                          </links>
                                     activityA
                                       <source linkName="L"/>...
                                     activityB
                                       <target linkName="L"/>...
                                   </flow>
```

With $CSP_{BPEL,}$ The trace specification of this workflow pattern can be defined as

activityA→activityB→SKIP

This workflow pattern has three major fault models.

- **Fault model 1: (activityA→SKIP)||(activityB→SKIP)**

This failure mode means that two activities that are supposed to execute in sequence, actually execute in parallel. As to listing4.1, this type of failure may be caused by incorrectly using a <flow> activity instead of <sequence> activity to wrap activityA and activityB. As to listing 4.2, this type of failure may be caused by mistakenly specifying a different link for either activityA or activityB. If activityA and activityB are not the source and target of the same link respectively, then they will have a concurrent relationship rather than a sequential relationship. When two activities that is supposed to execute in sequence and actually execute in parallel, our trace verification system can easily catch this fault. The process to catch this failure is shown in figure 4.1.

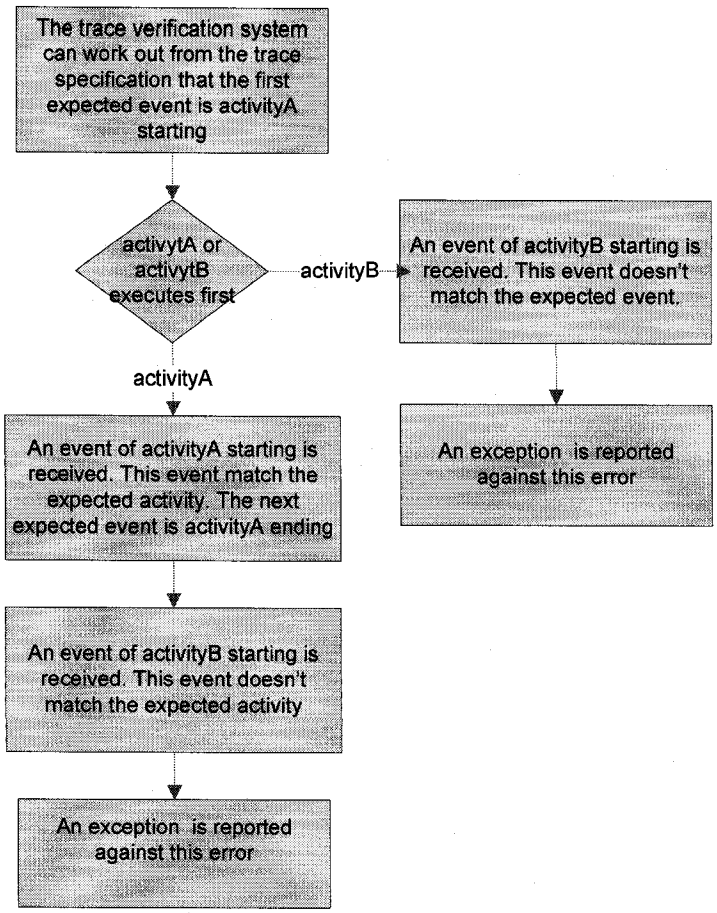


Figure 4.1 How to catch fault model1 in Sequence workflow pattern

- **Fault model 2: activityA→STOP or activityA→activityB→STOP or STOP**

This fault model is for two activities that are supposed to execute in sequence, however at most one activity executes. This type of failure may be caused by the failure of an activity or a<terminate> activity. The failure of an activity means that an activity doesn't terminate successfully. For example, if an activity is invoking an external Web service, then its failure could be "no response" from a Web service or it returning an exception. When an activity fails, the normal execution of the process is stopped and the exception handling is performed. A <terminate> activity explicitly terminates the instance of a BPEL4WS process.

When the sequence workflow pattern performs in this fault model, its actual trace of activities is different from that in fault model1. If an activity fails or a <terminate> activity performs before the execution of any activity, the end event of activityA or activityB won't happen. When the trace verification system expects for the end event of activityA or activityB and the expected event is not received, and then a time out error will happen. Therefore the fault is caught.

This type of fault model may happen in other workflow patterns. Since the reason of this fault model and the mechanism to catch the failure in the trace verification system is identical for all the workflow patterns, we won't analyze this type of fault model for other workflow patterns.

- **Fault model 3: activityB→activityA→SKIP**

This fault model means that activityB supposes to execute after the completion of activityA, however activityB executes ahead of activityA. As in List1, this fault model is caused by misplacing activityA after activityB within a <sequence> activity. With respect to List2, attaching the source element of a link construct to activityA and target element to activityB causes this kind of fault. When a BPEL4WS process runs in this fault mode, the start and end events of activityB arrive at the trace verification system before the start and end events of activityA. However, what the trace verification system expects is the start and end events of activityA to happen first. Therefore, the actual order of events doesn't match the expected one and the fault is caught.

- **Fault model 4: Deadlock**

A deadlock between activityA and activityB means that each activity waits for the other, henece neither completes. In a sequence workflow pattern, if a link construct is incorrectly used, a dead lock can happen. For example, in Listing 4.1, if a link construct is implemented incorrectly with the following code, a deadlock will happen.

```
<sequence>
  activityA
     <target linkName ="BtoA">...
  activityB
     <source linkName ="BtoA">...
</sequence>
```

When the sequence workflow pattern performs in this fault model, since neither activityA nor activityB can complete, the end event of activityA or activityB won't happen. When the trace verification system expects for the end event of activityA or activityB and the expected event is not received, a time out error will happen. Therefore the fault is caught.

4.2 Parallel Split

The Parallel Split pattern describes a point in the process where a single thread of control splits into multiple threads of control that can be executed in parallel, thus allowing activities to be executed simultaneously or in any order [WOH02]. The parallel split can be implemented by wrapping the activities in a <flow> activity.

Suppose, two activities, activityA and activityB, are working in a parallel split pattern. Their trace behavior can be specified as

(activityA→SKIP)|| (activityB→SKIP)

This workflow pattern has two major fault models.

- **Fault model1: activityA→activityB→SKIP or activityB→activityA→SKIP**

This fault model means that two activities suppose to execute in parallel, however they execute in sequence. Normally this type of fault is caused by misusing a <sequence> activity instead of a <flow> activity to wrap activityA and activityB. The expected trace behavior of two parallel activities is that one activity should start before the end of the other activity. If two activities actually execute in a sequence, then one activity can only start after the end of the other activity. Therefore, when the trace verification system receives the end event of the first running activity, it will find that the other activity has not yet started, and then the fault is caught.

- **Fault model 2: activityA→STOP or activityB→STOP or STOP**

This fault model is the same as fault mode 2 for sequence workflow pattern. So we don't repeat the discussion here. Analogously, we won't discuss this type of fault model during analyzing other workflow patterns.

4.3 Synchronization

The Synchronization pattern describes a point in the process where multiple parallel branches converge into one single thread of control thus synchronizing multiple threads [WOH02]. Synchronization can be realized through different solutions. Listings 4.3 and 4.4 are two different solutions to implement this concept. Listing 4.3 applies activities <flow> and <sequence>, while Listing 4.4 uses the activity <flow> and a link construct.

Listing 4.3

```
<sequence>
  <flow>
    activityA1
    activityA2
```

Listing 4.4

```
<flow name="F">
  <links>
    <link name="L1">
    <link name="L2"/>
```

```
  </flow>                                    </links>
  activityB                                  activityA1
</sequence>                                    <source linkName="L1"/>...
                                             activityA2
                                               <source linkName="L2"/>...
                                             activityB
                                               joinCondition ="L1 AND L2"
                                               <target linkName="L1"/>
                                               <target linkName="L2"/>
                                         </flow>
```

With $CSP_{BPEL,}$ The trace behavior of this workflow pattern can be defined as

(activityA1→SKIP)|| (activityA2→SKIP) →activityB

This workflow pattern has two major fault models.

- **Fault model1: activityA1→activityA2→activityB→SKIP or activityA2→activityA1→activityB→SKIP**

This fault model is similar as failure model in the Parallel Split workflow pattern. It also means that two activities supposed to execute in parallel execute in sequence. As to listing 4.3, forgetting wrapping activityA1 and activityA2 with a <flow> activity may cause this fault. As to listing 4.4, the possibility for this fault to happen is quite low since all three activities are tied with link constructs. If there is no <flow> activity, just like the cause of failure for listing 4.3, the implementation won't pass the syntax checking of the BPEL4WS engine. The process to catch this type of fault model is the same as that for failure model in Parallel Split workflow pattern, so we don't repeat the details here.

- **Fault model 2: (activityA1→SKIP) || (activityA2→SKIP) || (activityB→SKIP)**

This fault model is that all three activities execute in parallel. As to listing 4.3, this failure may be caused by putting all the activities in the <flow> activity. The process to catch

this kind of failure is the same as the process to catch fault model 1 of the Sequence workflow pattern since, for both of them, what is expected is a sequential trace while the actual trace is parallel.

- **Fault model 3: (activityA1→SKIP) || (activityA2→SKIP) →IF(C1){activityB→SKIP}**

In this fault model, after a BPEL4WS process reach the synchronization point, an extra condition is applied to the execution of activityB. ActivityB is supposed to execute without precondition. An extra transition condition applied to the source element of acitivytA1 or activityA2 causes this type of fault. When a BPEL4WS process runs in this fault model, after the process reaches the synchronization point, if condition C1 is not satisfied, activityB won't start. Therefore, the trace verification system won't receive the event of activityB, which is expected. When an expected event is not received, the fault is caught.

- **Fault model 4: (activityA1→SKIP) || (activityA2→SKIP) →IF(C1| C2) {activityB→SKIP}**

In this fault model, after a BPEL4WS process reach the synchronization point, there are two extra conditions applied to the execution of activityB. Those two conditions have an 'OR' relationship. The reason causing this fault model is that an extra transition condition is applied to each activity and the relationship between the conditions is set to 'OR'. When a BPEL4WS process runs in this fault model, the mechanism for the trace

verification system to catch this fault model is the same as that in fault model3, so we don't repeat here.

- **Fault model 5: (activityA1→SKIP) || (activityA2→SKIP) →IF(C1& C2) {activityB→SKIP}**

This fault model is highly similar with fault model 4. The only difference is that in this fault model the relationship between conditions is "AND' rather than 'OR' in fault model4. Since the reason causing this fault and the mechanism for the trace verification system to catch this fault are almost the same as those for fault model 4, we don't repeat them here.

4.4 Exclusive Choice

The Exclusive Choice pattern describes a point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen [WOH02]. A solution for exclusive choice is using two link constructs. The source elements of those two link constructs are associated to the decision activity. The value of transition condition in each source element determines which branch activity is chosen. In this pattern, the value of the transition condition in each source element is exclusive to each other, therefore it ensures that at any time only one transition condition can be true and then only one branch is selected. If more than one transition condition can be true, then the solution becomes the workflow pattern Multi-Choice, which we will introduce later. Listing 4.5 is an example to implement the workflow pattern exclusive choice.

Listing 4.5

```
<flow>
  <links>
    <link name="L1"/>
    <link name="L2"/>
```

```
</links>
activityA1
   <source linkName="L1" transitionCondition="C1"/>...
   <source linkName="L2" transitionCondition="C2"/>...
activityA2
   <target linkName="L1"/>...
activityA3
   <target linkName="L2"/>...
</flow>
```

In order to implement exclusive choice, the transition condition C1 and C2 must be exclusive. It means C1 and C2 is not allowed to be true at the same time. For example, C1 = "bpws:getVariableData('request', 'amount')>=10000", and C2="bpws:getVariableData('request', 'amount')<10000"

With CSP_{BPEL}, the trace behavior of the this workflow pattern can be specified as

ActivityA1→ IF (C1) {activityA2→SKIP} ELSE {activityA3→SKIP}

If two transition conditions are not exactly complementary, when we specify the trace behavior, we cannot use IF ELSE. For that sample exclusive choice, if C1 and C2 are not complementary, then the trace behavior has to be defined as

activityA1→((IF (C1){activityA2→SKIP}) || (IF(C2) {activityA3→SKIP}))

This workflow pattern has two major fault models.

- **Fault model 1: activityA1→ IF (C1) {activityA3→SKIP} ELSE {activityA2→SKIP}**

In this fault mode, activityA3 executes when condition C1 is true, while it is expected to run when condition C1 is false. Analogously, activityA2 executes when condition C1 is false, while it is expected to run when condition C1 is true. This fault model may be caused by incorrectly reversing the link names or transition conditions between two source elements of activityA1.

When a business process runs in this fault model, the actual trace of activities will differ from the expected trace. With the trace verification system, this failure will be detected. We use a small example to depict how the failure is caught. The example is a simple loan approval process. On receiving the loan request, the requested amount is compared to an amount (1000). If the requested amount is lower, then an Assessor service is called, otherwise the Approver service is used. The trace specification of the process is:

ReceiveRequest→ IF (request.amount>=1000){invokeApprover→SKIP} ELSE {invokeAssessor→SKIP }

If the process runs in fault model1, the events of activity ReceiveRequest can pass the verification. Suppose the actual request.amount is 2000. When the value 2000 is passed to the trace verification system, it can work out that the next activity is invokeApprover using the algorithm described in figure D.7. Due to the fault of implementation, activity invokeAssessor rather than activity invokeApprover executes. Hence, the expected activity event doesn't match the actual activity event and the fault is caught.

- **Fault model 2: activityA1→(activityA2→SKIP) || (activityA3→SKIP)**

In this fault model, the conditions used to determine the selection of branch activities are missed. After activity1 executes, the other two activities execute in parallel. Regarding Listing 4.5, the reason of this fault model is that the transition conditions in both source elements are removed. When a process executes in this fault model, all three activities executes while the trace specification only expects two activities since within an exclusive choice only one branch can be selected. Therefore the fault can be caught by the trace verification system.

There is a limitation in our system. If the variable value is not assigned to the condition expression by mistake, then the failure cannot be caught by our system. For instance, the condition expression is request.amount >1000. If the initial value of request.mount is 0, and the running process receive a value 2000 from request. That received value supposes to be passed to variable request.amount. However the received value is not passed to the variable by a mistake in implementation. Then the condition expression will be calculated based upon the value 0, and the value 0 will be passed to the trace verification system. Therefore, the failure of not passing variable values correctly is not caught. To tackle this limitation, a mechanism to verify the value of a variable at any point of the process is needed. In our future research, this issue will be addressed.

4.5 Simple Merge

The Simple Merge pattern describes a point in the workflow process where two or more alternative branches come together without synchronization [WOH02]. It is assumed that none of the alternative branches is ever executed in parallel. Simple Merge can only be achieved through a <switch> activity. Listing 4.6 is an example to implement simple merge.

Listing 4.6

```
<switch>
  <case condition ="C1">
     activityA1
  </case>
  <case condition ="C2">
     activityA2
  </case>
 </switch>
 activityC
```

Simple Merge pattern is quite similar as Exclusive Choice pattern. For Listing 4.6, if condition C1 and C2 are exclusive, then the implementation is an exclusive choice. The major difference between these two patterns is that in simple merge, multiple conditions can be true at the same time. However, although more than one condition can be true simultaneously, the <switch> activity determines that the first activity, for which the specified condition evaluates to true, can be triggered.

With $CSP_{BPEL,}$ The trace behavior of the sample simple merge can be specified as

IF (C1) {activityA1→SKIP} ELSE {IF (C2) {activityA2→SKIP}}

Simple merge pattern has two major fault models.

- **Fault model 1: IF (C1) {activityA2→SKIP} ELSE {IF (C2) {activityA1→SKIP}}**

In this fault mode, activityA2 executes as long as conditions C1 is true, while it is activityA1 that supposes to run when condition C1 is true. Reversing activities A1 and A2 causes this fault model. When a process executes in this fault model, the trace verification system can easily catch the failure. If both condition C2 and C1 are true, then activityA2 executes. Since the trace verification system expects acitivtyA1 to execute when both C2 and C1 are true, therefore the fault is caught. If only C2 is true, activityA1 executes while the expected execution is actiivtyA2. Analogously, If only C1 is true, activityA2 executes while the expected execution is actiivtyA1. In both cases, the fault is caught. If neither C1 nor C2 is true, no activity will execute and no activity is expected. In this case, the fault is not caught.

- **Fault model 2: IF (C1*) {activityA1→SKIP} ELSE {IF (C2*) {activityA2→SKIP}}**

Fault model2 is caused when the conditions for the activities, A1 and A2, are modified incorrectly. Under some circumstances when a process executes in fault model2, the failure doesn't change the execution trace of activities, so that our monitoring system cannot catch the failure. Under other circumstances, the failure can lead to the departure of execution trace of activities, so that the failure can be caught. For instance, in the following implementation of the Simple Merge workflow pattern

```
<switch>
   <case condition= "bpws:getVariableProperty(stockResult,level) &gt; 100">
       activityA1
   </case>
   <case condition="bpws:getVariableProperty(stockResult,level) &gt;=0">
       activityA2"
   </case>
</switch>
```

its trace specification is : IF (stockResult.level >100) {activityA1→SKIP} ELSE {IF (stockResult.level>0) {activityA2→SKIP}}. Suppose this implementation is in fault model 2. For instance, the first condition is incorrectly written as " bpws: getVariableProperty (stockResult, level) > 200. Then when stockResult.level is 201, the first condition expression returns true, and activityA1 executes. This trace of activities matches what the trace verification system expects and no fault is reported. It means this fault cannot be caught by the trace verification system when stockResult.level is 201. Contrarily, if stcokResult.level is 199, then the first condition expression returns false and the second condition expression returns true. Therefore the activityA2 executes. For the trace verification system, since value "199" is bigger than 100 hence it still expects activityA1 to execute. Therefore, the fault is caught.

4.6 Multi-Choice

The Multi-Choice pattern describes a point in the workflow process, where, based on a decision or control data, a number of branches are chosen and executed as parallel threads [WOH02]. In BPEL4WS, we can only use link construct to implement multi-choice. The solution of multi-choice pattern is identical to the solution of exclusive choice pattern we mentioned in section 4.4 that if more than one transition condition in the source elements can be true at the same time, then the workflow pattern will transform from exclusive choice to multi-choice.

As to the example implementation of exclusive choice pattern in section 4.4, if the transition conditions are not exclusive, then the implementation can be considered as an example of multi-choice pattern.

With $CSP_{BPEL,}$ The trace behavior of multi-choice pattern can be specified as

activityA1→(IF (C1){activityA2→SKIP}|| IF (C2){activityA3→SKIP})

Deferred Choice pattern has three major fault models.

- **Fault model 1: activityA1→ (IF (C2){activityA2→SKIP}|| IF (C1){activityA3→SKIP})**

  This fault model is quite similar with fault model 1 for Exclusive Choice pattern. They both are about switching the conditions between two activities. Therefore, we don't repeat the discussion for this fault model.

- **Fault model 2: activityA1→ (IF (C1){activityA2→SKIP}|| IF (C3){activityA3→SKIP})**

  In this fault model, the condition applied to activityA3 is modified so that conditions C1 and C3 cannot be true at the same time. As conditions cannot be

true at the same time, activityA2 and activityA3 cannot execute at the same time. Suppose a BPEL4WS process runs in this fault model. If conditions C1 and C2 become true at a certain point, while condition C3 is not true at that point, then the trace verification system will expect the execution of activityA2 and activityA3 and it won't receive the event of activiytA3. Therefore, the fault is caught.

- **Fault model 3: activityA1→ ((activityA2→SKIP)|| (activityA3→SKIP))**

  This fault model is quite similar with fault model 2 for Exclusive Choice pattern. They both are about missing the conditions for conditional activities. Therefore, we don't repeat the discussion for this fault model.

4.7 Synchronizing Merge

The Synchronizing merge pattern describes a point in the process where multiple paths converge into one single thread [WOH02]. Some of these paths are being executed and some are not. If only one path is active, the activity after the merge is triggered as soon as this path completes. If more than one path is active, synchronization of all active paths needs to take place before the next activity is triggered [WOH02]. The solution of synchronization merge is based on the solution of multi choice. List 7 is an example of synchronization merge.

Listing 4.7

```
1 <flow>
2   <links>
3     <link name="L1"/>
4     <link name="L2"/>
5     <link name="L1s"/>
6     <link name="L2s"/>
7   </links>
8   <empty>
9     <source linkName="L1" transitionCondition ="C1"/>
```

```
10      <source linkName="L2" transitionCondition="C2"/>
11  </empty>
12  activityA1
13      <target linkName="L1"/>
14      <source linkName="L1s"/>
15  activityA2
16      <target linkName="L2"/>
17      <source linkName="L2s"/>
18  activityC
19      joinCondition ="L1s OR L2s"
20      <target linkName="L1s"/>
21      <target linkName="L2s"/>
22 </flow>
```

In this solution, line1 to line17 excluding the declaration of links L1s and L2s composes a multi-choice pattern. ActivitiesA1 and A2 are branches, which are controlled by the conditions C1 and C2 respectively. If C1 is evaluated to be true, then activityA1 can execute, similarly if C2 is evaluated to be true, then activityA2 can start. Besides the multi-choice pattern, links L1s and L2s synchronize activityA1 with activityA2 before the execution of activityC. The relationship between the activityA1 and activityA2 is not a strict synchronization, and it is different from the standard synchronization stated in section 3.3. It is activityC's element "joinCondition = "L1s OR L2s" " that causes this difference. It ensures that if both activitiesA1 andA2 are active, then they should really synchronize before the execution of activityC. It also ensures that if only one activity, A1 orA2, is active, then activityC can execute after the completion of the active activity. If there is no this element or its "OR" is replaced by "AND", then the solution becomes a standard synchronization pattern.

Using $CSP_{BPEL,}$ The trace behavior of the example can be specified as

(IF (C1) {activityA1→SKIP} || IF (C2) {activityA2→SKIP})→IF (C1 | C2)

{activityC→SKIP} ELSE {SKIP}

Synchronization Merge pattern has one major fault models.

- **Fault model 1: (IF (C1){activityA1→SKIP}|| IF (C2){activityA2→SKIP})→IF(C1 &C2) activityC→SKIP}ELSE{SKIP}**

  In this fault model, for activityC, the relationship between its conditions C1 and C2 changes from OR to AND, so that activityC doesn't execute until both condition C1 and condition C2 are satisfied. . In line 19 of listing 4.7, changing operator 'OR' to 'AND' causes this kind of fault. When a BPEL4WS process runs in this fault model, if condition C1 is true and condition C2 is false, activityC won't execute. The trace verification system will expect the execution of activityC as long as condition C1 is satisfied, therefore the fault is caught.

4.8 Implicit Termination

The Implicit termination pattern means that a given sub process is terminated and the termination doesn't require an explicit termination activity [WOH02]. An example explicit termination activity is a <terminate> activity. This pattern cannot be implemented by user programming, since it is an inherent characteristic of BPEL4WS program and it is implemented by the BPEL4WS engine. In BPEL4WS, the Flow construct is the only activity that can perform this workflow pattern.

Implicit termination pattern doesn't have observable trace behavior, since it only means the termination of a sub process and the termination of a sub process is indicated by its trace specification. Therefore, we don't need to specify its trace behavior with CSP_{BPEL} and analyze its associated fault models.

4.9 Deferred Choice

The Deferred Choice pattern describes a point in a process where one among several alternative branches is chosen based on information which is not necessarily available when this point is reached [WOH02]. It is different from Exclusive Choice pattern, in that the choice is not made immediately when the point is reached. The Deferred Choice pattern can only be implemented through a pick construct. An example implementation of the Deferred Choice pattern is listing 4.8

```
<pick>
  <onMessage name="message1" partnerLink="buyer" portType="orderEntry"
   operation="inputLineItem"variable="lineItem">
    activityA
  </onMessage>
  <onMessage name="message2" partnerLink="buyer" portType="orderEntry"
   operation="orderComplete"  variable="completionDetail">
    activityB
  </onMessage>
  <onAlarm for="'P3DT10H'">
    activityC
  </onAlarm>
</pick>
```

Listing 4.8

With CSP_{BPEL}, the trace behavior of the example can be specified as

WHILE(BPEL_WAIT_Name1<P3DT10H){IF(BEPL_Message=buyer_orderEntry _inputLineItem_lineItem){activityA→ SKIP } ELSE { IF(BEPL_Message = buyer_orderEntry_orderComplete_completionDetail){acitivytB→SKIP }}→IF(BEPL_W AIT_Name1>=P3DT10H){activityC→ SKIP }

Deferred Choice pattern has one major fault models.

- **Fault model 1:**

  WHILE(true){IF(BEPL_Message=buyer_orderEntry_inputLineItem_lineIte

m){activityA→ SKIP }ELSE { IF(BEPL_Message=

buyer_orderEntry_orderComplete_completionDetail){acitivytB→ SKIP }}

In this fault model, there is no alarm defined. Therefore, when a BPEL4WS process runs in this fault model, if there is no expected message received, the BPELWS process will keep waiting for the message no matter how long it will wait. On the other side, the trace verification system thinks that there is an alarm defined and expects to receive the events of alarm hander after the waiting period of alarm is over, therefore the fault is caught.

4.10 Cancel Activity

The Cancel Activity pattern describes the abnormal termination of a running instance of an activity [WOH02]. It can be implemented with fault and compensation handlers. An example implementation of this workflow pattern is listing 4.9

```
<scope>
  <faultHandlers>
    <catch faultName="Fault1" faultVariable="Var1">
      activityA
    </catch>
  </faultHandlers>
   activityB
</scope>
```

Listing 4.9

For listing 4.9, when a fault happens during the execution of activityB, if its name is "Fault1" and its variable is "Var1", then the execution of activityB is terminated and the fault handler activityA is performed. Otherwise, after the termination of activityB, the fault is thrown out.Using $CSP_{BPEL,}$ the trace behavior of the example can be specified as

activityB.b→IF(BPEL_Fault==true{IF((BPEL_FaultName==Fault1)&(BPEL_Fa ultVar==Var1)){activityA}→STOP}ELSE{activityB.e→SKIP}

This Workflow pattern has one major fault models.

- **Fault model1:**

  activityB.b→IF(BPEL_Fault==true{IF((BPEL_FaultName==Fault2)&(BPE L_FaultVar==Var2)){activityA}→STOP}ELSE{activityB.e→SKIP}

  In this fault model, the fault type or the data variable associated with the fault is not correct. When a BPEL4WS process runs in this fault model, if a fault, whose name is "fault1" and data variable is "Var1", happens during the execution of activityB , the fault won't be caught because of the mismatch of fault name and data variable. Therefore the fault handler activityA won't be triggered. According to the trace specification, the trace verification system expects the event of activityA. Because of the fault, the event of activityA won't happen, so that the trace verification system cannot receive the expected event, and then the fault is caught.

4.11 Cancel Case

The Cancel Case pattern means the termination of an entire BPEL4WS process instance [WOH02]. This pattern can be solved with the <terminate> activity. The trace behavior of this workflow pattern can be specified as "STOP".

This Workflow pattern has one major fault model.

- **Fault model 1:**

  STOP

In this fault model, a termination happens while it is not expected. When a termination happens, the instance of BPEL4WS process stops running. As a result, the trace verification system cannot receive any event from that BPEL4WS instance while it still waits for the events. After waiting for a period of time the trace verification system throws out a timeout error, therefore it means the fault is caught.

In section 4.1, when we explored fault models for the workflow pattern Sequence, its fault model 2 was defined as a type of fault, where an unexpected termination happens because of an extra <terminate> activity or the failure of monitored activities. Since the fault model 2 in the workflow pattern Sequence and this fault model in workflow pattern Cancel Case both are about unexpected termination of a process instance, we consider them identical and we won't analyze them separately when we discuss the implementation for every fault model

5 Faults and Mutation processes for BPEL4WS processes

5.1 Introduction

In section 4, we proposed one or more fault models for each workflow pattern. Each fault model was proposed based on a specific instance of a workflow pattern. In other words, the definitions of the fault models there were not generic. In this chapter, we will discuss the generic definitions of these models. Further, we will explore how we can generate fault models by mutating BPEL4WS processes. These fault models are then utilized to evaluate our system by generating mutants which we attempt to discover.

Mutation testing is a fault-based testing technique originally proposed in 1978 [LIP78]. In mutation testing, small faults are inserted into the original program to create faulty versions of the original program. The small faults are created by applying mutation operators, which describe syntactic changes to be made to the program. If a test set can produce different results compared with the original program for a mutant, the mutant is killed. The test sets are adequate if they can kill all the mutants. Although mutation testing has a rich history, most mutation operators have been developed for procedural programs [MAY02]. Further, there is no precise definition regarding how small, or the size, a fault should possess.

When we apply the methodology of mutation testing to BPEL4WS processes, we define a BPEL4WS process with an injected fault as a mutant, and we use terminology "mutation process" to represent the process of how a fault is generated. In mutation testing, a fault injected into the original program should be small. Since in other researches there is no precise definition of how small the fault should be, we tried to define the size of a fault in our research. In order to define this, we applied the concept of Levenshtein distance [LEV01]. The Levenshtein distance is used to measure the difference between two strings by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution on a single character. For BEPL4WS processes, we defined that an operation is an insertion, deletion, or substitution on a single token within a trace specification. The reason for us to apply an operation to a single token rather than a single character is that in most cases an operation to a single character in a BPEL4WS process cannot cause a fault within an execution

trace. Based upon the definition of operation, the size of fault for BPEL4WS process is measured by the minimum number of operations needed to transform it to a fault. When we design faults for BPEL4WS processes, we try to design faults with minimum size. The following two examples illustrate the concept of fault size.

- Example A:

  The trace specification of the workflow pattern Sequence is <EVENT_IDENT1><TRACE_PREFIX_OPERATOR><EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>. If we replace the token <TRACE_PREFIX_OPERATOR> with token <TRACE_TERMINATION_OPERATOR>, then we get the fault "Unexpected Termination". The size of the fault is 1, since the fault is implemented by only one substitution operation.

- Example B

  We still use the trace specification of the workflow pattern Sequence. If we replace the token <TRACE_PREFIX_OPERATOR> with the token <TRACE_PARALLEL_OPERATOR>, then we get the fault "Sequence to Parallel". The size of the fault is 1, since the fault is implemented by only one substitution operation. There is one thing we want to clarify for this case. Since the token <TRACE_PARALLEL_OPERATOR> syntactically can only be used to connect two processes, we have to attach tokens "<TRACE_PREFEX_OPERATOR><TRACE_SKIP_OPERATOR>" after token "<EVENT_IDENT1>" AND "<EVENT_IDENT2>" to make the trace specification correct in the syntax.

After we define the faults with minimum size, we will analyze the mutation process from two perspectives. Firstly, we analyze it from the perspective of CSP_{BPEL}. CSP_{BPEL} is an abstract representation of BPEL4WS processes. It focuses on the trace behavior of BPEL4WS process and doesn't deal with the details of BPEL4WS implementation. Therefore, representations in format of CSP_{BPEL} give us a high level overview of the mutation processes of BPEL4WS or a specification of the process. We don't use actual trace specifications to represent a workflow pattern, since they would not be actual generic. In order to generically describe the specification of the mutation processes, we describe each specification pattern by providing a BNF description of the CSP_{BPEL} construct required to represent a workflow pattern.

In order to demonstrate the implementation of mutation process defined upon CSP_{BPEL} and provide more detailed direction for the generation of mutants in actual-world testing, we also illustrate the mutation processes from the perspective of BPEL4WS itself. Again, actual BPEL4WS programs are not appropriate to describe the mutation processes, since there are two many variations of BPEL4WS programs for each work pattern. To tackle this issue, we again use BNF representations of BPEL4WS programs to describe the mutation process. The BNF representations precisely represent actual programs and they are sufficiently generic so that each BNF representation is able to represent various implementations of a workflow pattern. We build our BNF for BPEL4WS based on the BNF defined in [ZIJ06]. We inherit most definitions from this source. However, the form of the definitions is not ideal for our purpose, and hence a set

of revised definitions specifically aimed at describing BPEL4WS activities is presented in Appendix C.

Before we illustrate the mutation processes, we defined some operators that will be used to represent mutation processes. These operators can be used for both BNF of CSP_{BPEL} and BPEL4WS.

Definition: Operator Replace (denoted as '$\overrightarrow{\rightarrow}$') is used to represent replacing the token(s) on the left of operator with the token(s) on the right. For example, <a1>$\overrightarrow{\rightarrow}$<a2> means replacing token <a1> with token a2. Analogously, <b1><b2>$\overrightarrow{\rightarrow}$ <b3> means replacing two adjoining tokens <b1> <b2> with one token <b3>.

Definition: Operator Switch (denoted as '$\overrightarrow{\leftarrow}$') is used to represent replacing two tokens with each other. For example, <a1>$\overrightarrow{\leftarrow}$<a2> means replacing token <a1> with token a2 and replacing token <a2> with token <a1>

Definition: Operator Followed By (denoted as '►') is used to represent that one token is followed by another token. For example, <a>►<b> means that token <a> and token <b> are conjoint and token <a> is followed by token <b>

Definition: Operator Not (denoted as '!') is used to represent the opposite action of an operator. Therefore, it always used in a combination with another operator. For example, '! ►' means a token is not followed by the other token.

Definition: Operator Included By (denoted as '↕') is used to represent that two tokens are included by another set of tokens. The usage format of this operator is (<a>, <b>) ↕ <c>.

Definition: Symbol P is used to represent a workflow pattern, defined in the format of BNF of CSP_{BPEL}.

Besides the mutation processes from two perspectives, we will also discuss how the mutation patterns are actualized in BPEL4WS.

5.2 Sequence

5.2.1 Requirement of mutation process

For the sequence workflow pattern, of the BNF representation of the CSP_{BPEL} is:

<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>

As we mentioned in section 4.1, there are four major fault models for the sequence workflow pattern. Those fault models respectively are

1) Sequential to Parallel (FM1)

   (<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)><TRACE_PARALLEL_OPERATOR>

   (<EVENT_IDENT2><TRACE_PREFIX_OPERATOR>><TRACE_SKIP_OPERATOR>)

2) Unexpected Termination (FM2)

   <EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_TERMINATION_OPERATOR> **OR**

   <EVENT_IDENT1><TRACE_PREFIX_OPERATOR><EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_TERMINATION_OPERATOR>

   OR

   <TRACE_TERMINATION_OPERATOR>

3) Switch Two Activity Events (FM3)

<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><EVENT_IDENT1><T

RACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>

4) Deadlock (FM4)

<TRACE_DEADLOCK_OPERATOR>

5.2.1.1 Sequential to Parallel

We can find that the essence of the fault model Sequential to Parallel is that the relationship between two activity events is changed from sequential to parallel. Consequently, we name the mutation process for that fault model as Sequence to Parallel. This mutation process has two steps:

1) **IF** <EVENT_IDENT>!▶< TRACE_PREFIX_OPERATOR> **THEN**
<EVENT_IDENT>↛<EVENT_IDENT>
TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>

ELSE <EVENT_IDENT> <TRACE_PREFIX_OPERATOR>
<TRACE_SKIP_OPERATOR>↛ (<EVENT_IDENT>
TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)

2) **IF** <TRACE_PREFIX_OPERATOR> !▶<TRACE_SKIP_OPERATOR>
THEN <TRACE_PREFIX_OPERATOR>↛
<TRACE_PARALLEL_OPERATOR>

5.2.1.2 Unexpected Termination

As to the fault model Unexpected Termination, its major characteristic is that termination happens unexpectedly. For this characteristic, we call the mutation process for that fault model simply "Unexpected Termination". Due to different behavior of this fault model, there are three possibilities for this mutation process. They are

1) <EVENT_IDENT1>→<TRACE_TERMINATION_OPERATOR>

2) <EVENT_IDENT2>→<TRACE_TERMINATION_OPERATOR>

3) <TRACE_SKIP_OPERATOR>→<TRACE_TERMINATION_OPERATOR>

5.2.1.3 Switch Two Activity Events

The fault model Switch Two Activity Events essence is that the order of two activity events is switched. Therefore, the mutation process for that fault model is called Switch Two Activity Events. This mutation process has one step:

1) <EVENT_IDENT1>⇄<EVENT_IDENT2>

5.2.1.4 Deadlock

The fault model Deadlock essence is that there is a deadlock between the two activity events. The mutation process for that fault model is named Deadlock based on this essence. This mutation process has only one step:

1) <EVENT_IDENT1>→<TRACE_DEADLOACK_OPERATOR>

5.2.2 Implementation of mutation process

For the Sequence workflow pattern, there are two types of implementation. One is using the <sequence> activity, and the other one is using the link construct. For each type of implementation, we are going to illustrate the mutation process for each fault model.

5.2.2.1 Implementation using <sequence> activity

For the implementation of the Sequence workflow pattern, which uses <sequence> activity, its BNF is

```
<sequence standard-attributes>
```

```
    standard-elements
    activity_a activity_b activity*
</sequence>
```

5.2.2.1.1 Sequential to Parallel

The implementation of the fault model Sequential to Parallel is:

```
<flow standard-attributes>
    standard-elements
    activity_a activity_b activity*
</flow>
```

The mutation process for this fault model is:

sequence → flow

5.2.2.1.2 Unexpected Termination

For this type of fault model, there are three variations of implementation. The difference between these three variations is the place where the termination happens. For each variation of the fault model, we describe its implementation and its corresponding mutation process respectively.

- **Implementation A:**

```
<sequence standard-attributes>
  standard-elements
  activity_a
  <terminate/>
  activity*
</sequence>
```

The mutation process is:

activity_b → <terminate>

- **Implementation B:**

```
<sequence standard-attributes>
  standard-elements
  <terminate/>
  Activity_b
  activity*
</sequence>
```

The mutation process is:

Activity_a → <terminate/>

- **Implementation C:**

```
<sequence standard-attributes>
 standard-elements
 activity_a
 activity_b
 <terminate/>
 activity*
</sequence>
```

The mutation process is:

activity_b → activity_b <terminate/>

5.2.2.1.3 Switch Two Activity Events

The implementation of the fault model Switch Two Activity Events, written in BNF, is:

```
<sequence standard-attributes>
    standard-elements
    activity_b
    activity_a
    activity*
</sequence>
```

The mutation process for this fault model is

$activity_b \rightleftarrows activity_a$

5.2.2.1.4 Deadlock

The implementation of the fault model Deadlock is:

```
<flow>
<links>
   <link name ="name1"/>
</links>
<sequence standard-attributes>
    standard-elements
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1"/> <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name1"/> <target linkName="ncname2"/>*
    <source linkName="ncname2"
    transitionCondition="bool-expr"?/>*
    activity-end-element_b
```

```
    activity*
</sequence>
</flow>
```

This implementation is feasible only when $activity_a$ and $activity_b$ are included in a <flow> activity, since if those activities are not included in a <flow> activity, the source or target elements cannot be used within them. The mutation process for this fault model is:

IF ($activity_a$, $activity_b$) ↕<flow> **THEN**

<links> → <links> <link name ="name1"/>

activity-start-element$_a$ → activity-start-element$_a$ <target linkName="name1"/>

activity-start-element$_b$ → activity-start-element$_b$ <source linkName="name1"/>

5.2.2.2 Implementation using link construct

For the implementation of the Sequence workflow pattern, which uses a link construct, its BNF is

```
<flow standard-attributes>
standard-elements
    <links>
      <link name="name1"/>
      link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1"/> <source linkName="ncname1"
     transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/> <target linkName="ncname2"/>*
    <source linkName="ncname2"
     transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

5.2.2.2.1 Sequential to Parallel

The implementation of this fault model is:

```
<flow standard-attributes>
    standard-elements
```

```
    <links>
        <link name="name1"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model is

<source linkName="name1"/> $\overrightarrow{\neq}$ Φ

<target linkName="name1"/> $\overrightarrow{\neq}$ Φ

5.2.2.2.2 Unexpected Termination

When the Sequence workflow pattern is implemented through a link construct, the time when the <terminate> activity executes is undeterminable. On the other side, we can ensure that the <terminate> activity will definitely execute. Therefore, the implementation of this fault model doesn't have the three variations described in section 5.1.2.1.2. Instead, it only has one variation, and where we put the <terminate> activity doesn't matter. The implementation of this fault model is

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1"/> <source linkName="ncname1"
     transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/> <target linkName="ncname2"/>*
    <source linkName="ncname2"
     transitionCondition="bool-expr"?/>*
    activity-end-element_b
    <terminate/>
    activity*
```

```
</flow>
```

The mutation process for this fault model is

activity-end-element$_b$ → activity-end-element$_b$ <terminate/>

5.2.2.2.3 Switch Two Activity Events

The implementation of the fault model Switch Two Activity Events is:

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1"/> <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name1"/> <target linkName="ncname2"/>*
    <source linkName="ncname2"
     transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model is

<target linkName="name1"/> ⇄ <source linkName="name1"/>

5.2.2.2.4 Deadlock

The implementation of fault model Deadlock is:

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1"/>
    <target linkName="name2"/>
    <source linkName="ncname1"
     transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/>
    <source linkName="name2"/>
```

```
    <target linkName="ncname2"/>*
    <source linkName="ncname2"
     transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model has three steps

1) `<source linkName="name1"/>` → `<source linkName="name1"/>` `<target linkName="name2"/>`

2) `<target linkName="name1"/>` → `<target linkName="name1"/>` `<source linkName="name2"/>`

3) **`<link name="name1"/>`** → `<link name="name1"/>` `<link name="name2"/>`

5.2.3 BPEL4WS examples

For each fault model of a workflow pattern, when it is applied to an actual BPEL4WS program, there might be more than one mutant (BPEL4WS program with fault injected) being generated. The number of the mutants is determined by the implementation variation of the workflow pattern. It means that if there is more than one type of implementation for a workflow pattern, then there will be more than one mutant for a fault model. The reason is that each type of implementation has its unique mutation process for a fault model. Meanwhile, each mutation process can be used to derive a mutant. As we mentioned in section 5.1.2, there are two types of implementations for the Sequence workflow pattern. Therefore, in this section, for each fault model, we present two mutants based on the two mutation processes for that fault model.

Initially, we considered using actual BPEL4WS programs to represent the mutants. However, when we analyzed the original BPEL4WS programs and their mutants, we found that there were many identical codes among them, such as attributes of an activity.

These identical codes take much space, however they are not valuable for the representation of fault models. For the sake of simplicity, we use a symbol to represent a snippet of BPEL4WS code, which is identical in both the original BPEL4WS program and its mutants. The format of symbol is an underlined BPEL4WS activity tag, and that activity tag may have a subscript. An example of symbol is $\underline{<receive_{name}>}$. For instance, in the following equations, the symbol on the left is used to represent the BPEL4WS code on the right side. We will use these two symbols in the description of examples.

- <receive> = *<receive name="receive1" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT" variable="request/">*

- <invoke>= *<invoke inputVariable="request" name="invokeapprover" operation="approve" outputVariable="approvalInfo" partnerLink="approver" portType="apns:loanApprovalPT/>*

5.2.3.1 Sequential to Parallel

When the sequence work flow pattern is implemented using the <sequence> activity, if the original BPEL4WS program possesses the same structure as the program in listing 5.1(a), according to the mutation process (defined in section 5.1.2.1.1), the mutant will be listing 5.1(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<sequence> <receive> </receive> <invoke> </invoke> </sequence>	<flow> <receive> </receive> <invoke> </invoke> </flow>

Listing 5.1

When sequence workflow pattern is implemented through a link construct, if the original BPEL4WS program possesses the same structure as the program in listing 5.2(a), according to the mutation process (defined in section 5.1.2.2.1), the mutant will be listing 5.2(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="R-to-A"/> </links> <receive> <source linkName="R-to-A"/> </receive> <invoke> <target linkName="R-to-A"/> </invoke> </flow>	<flow> <links> <link name="R-to-A"> </links> <receive> </receive> <invoke> </invoke> </flow>

Listing 5.2

5.2.3.2 Unexpected Termination

When the sequence work flow pattern is implemented through the <sequence> activity, the implementation of this mutation process is inserting a <terminate> activity into the <sequence> activity. There are three variations of this mutation process. Corresponding to those three variations, we inject a <terminate> activity into different locations in the <sequence> activity. If the original BPEL4WS program possesses the same structure as the program in listing 5.3(a), according to the mutation process (defined in section 5.1.2.1.2), the mutant will be listing 5.3(b), 5.3(c) and 5.3(d).

Original BPEL4WS program(a)	Mutant1 (b)	Mutant2 (c)	Mutant3 (d)
<sequence> <receive> </receive> <invoke>	*<sequence>* *<terminate/>* <receive> </receive>	*<sequence>* <receive> </receive> *<terminate/>*	*<sequence>* *<receive>* *</receive>* *<invoke>*

</invoke> </sequence>	<invoke> </invoke> </sequence>	<invoke> </invoke> </sequence>	</invoke> <terminate/> </sequence>

Listing 5.3

When the sequence workflow pattern is implemented through a <flow> activity, if the original BPEL4WS program possesses the same structure as the program in listing 5.4(a), according to the mutation process (defined in section 5.1.2.2.2), the mutant will be listing 5.4(b).

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="R-to-A"/> </links> <receive> <source linkName="R-to-A"/> </receive> <invoke> <target linkName="R-to-A"/> </invoke> </flow>	<flow> <links> <link name="R-to-A"/> </links> <receive> <source linkName="R-to-A"/> </receive> <invoke> <target linkName="R-to-A"/> </invoke> <terminate/> </flow>

Listing 5.4

5.2.3.3 Switch Two Activity Events

When sequence work flow pattern is implemented through activity <sequence>, if the original BPEL4WS program is like listing 5.5(a), the mutant of mutation process (defined in section 5.1.2.1.3) will be listing 5.5(b). The fault injected in this mutant is that the order of activtyA and activityB in a <sequence> activity is switched.

Original BPEL4WS program	Mutant BPEL4WS program
<sequence>	<sequence>

<receive> </receive> <invoke> </invoke> </sequence>	<invoke> </invoke> <receive> </receive> </sequence>

Listing 5.5

When sequence workflow pattern is implemented through link construct, if the original BPEL4WS program is like listing 5.6(a), the mutant of mutation process (defined in section 5.1.2.2.3) will be listing 5.6(b). The fault injected in this mutant is that the link attribute of activtyA and activityB in a <flow> activity is switched.

Original BPEL4WS program (a)	Mutant BPEL4WS program
<*flow*> <*links*> <*link name*=" R-to-A "> </*links*> <receive> <*source linkName*=" R-to-A "/> </receive> <invoke> <*target linkName*=" R-to-A "/> </invoke> </*flow*>	<*flow*> <*links*> <*link name*=" R-to-A "> </*links*> <receive> <*target linkName*=" R-to-A "/> </receive> <invoke> <*source linkName*=" R-to-A "/> </invoke> </*flow*>

Listing 5.6

5.2.3.4 Deadlock

When sequence work flow pattern is implemented through activity <sequence>, , if the original BPEL4WS program is like listing 5.7(a), the mutant of mutation process (defined in section 5.1.2.1.4) will be listing 5.8(b).

Original BPEL4WS program	Mutant BPEL4WS program
<*flow*> <*sequence*> <receive> </receive> <invoke> </invoke> </*sequence*>	<*flow*> <*links*> <link name="R-to-A"> </*links*> <sequence> <receive> <*target linkName*="R-to-A"/>

</flow>	</receive> <invoke> <source linkName="R-to-A"/> </invoke> </sequence> </flow>

Listing 5.7

When sequence workflow pattern is implemented through link construct, if the original BPEL4WS program is like listing 5.8(a), the mutant of mutation process (defined in section 5.1.2.2.4) will be listing 5.8(b).

Original BPEL4WS program	Mutant BPEL4WS program
<flow> <links> <link name=" R-to-A "> </links> <receive> <source linkName=" R-to-A "/> </receive> <invoke> <target linkName=" R-to-A "/> </invoke> </flow>>	<flow> <links> <link name=" R-to-A "> <link name=" A-to-R "> </links> <receive> <source linkName=" R-to-A "/> <target linkName=" A-to-R "/> </receive> <invoke> <target linkName=" R-to-A "/> <source linkName=" A-to-R "/> </invoke> </flow>

Listing 5.8

5.3 Parallel Split

5.3.1 Requirement of mutation process

For the Parallel workflow pattern, its representation in the BNF of CSP_{BPEL} is:

(<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>

)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_PREFIX_OPER

ATOR><TRACE_SKIP_OPERATOR>)

As we mentioned in section 4.2, there are two major fault models for Parallel Split workflow pattern. Those fault models respectively are

1) Parallel to Sequential (FM5)

   <EVENT_IDENT1><TRACE_PREFIX_OPERATOR><EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR >or

   <EVENT_IDENT2><TRACE_PREFIX_OPERATOR><EVENT_IDENT1><TRACE_PREFIX_OPERATOR>><TRACE_SKIP_OPERATOR >

2) Unexpected Termination (FM2)

   < TRACE_TERMINATION_OPERATOR>

5.3.1.1 Parallel to Sequential

The mutation process for this fault model has three steps:

1) P→P ><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR >
2) (<EVENT_IDENT>
   <TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR)→
   <EVENT_IDENT>
3) <TRACE_PARALLEL_OPERATOR>→<TRACE_PREFIX_OPERATOR>

5.3.1.2 Unexpected Termination

Fault model Unexpected Termination of workflow pattern Parallel Split is little different from that of workflow pattern Sequence. As we described in section 5.1.2.1.2, in the workflow pattern Sequence, this fault model has three types of variations. Each variation differs at the time when a termination happens. In the workflow pattern Parallel Split, this

fault model has only one type of behavior, because when a termination is going to happen, we can not tell when it will happen. The mutation process for this fault model is:

(<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)↛
< TRACE_TERMINATION_OPERATOR>

It is easy to find out that we can apply the fault model Unexpected Termination to every workflow pattern, since we can all inject a <terminate> activity. Since, both the deterministic and non-deterministic versions of this model have already been described and discussed, for further analyzed workflow patterns , their fault model Unexpected Termination will not be examined.

5.3.2 Implementation of mutation process

For the workflow pattern Parallel Split, there is only one type of implementation. That is using the <flow> activity. The BNF of the implementation of this workflow pattern is:

```
<flow standard-attributes>
    standard-elements
    links?
    activity_a
    activity_b
    activity*
</flow>
```

5.3.2.1 Parallel to Sequential

There are two kinds of implementation for this fault model. The first kind, in BNF, is:

```
<sequence standard-attributes>
    standard-elements
    activity_a
    activity_b
    activity*
</sequence>
```

The mutation process, which contributes to this implementation, is:

flow $\overset{\rightarrow}{\rightarrow}$sequence

The second kind of implementation is:

```
<flow standard-attributes>
    standard-elements
    <links>
      <link name="name1"/>
      link*
    </links>
    activity-start-element_a
      <target linkName="ncname1"/>*
      <source linkName="name1"/> <source linkName="ncname1"
      transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
      <target linkName="name1"/> <target linkName="ncname2"/>*
      <source linkName="ncname2"
        transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process, which contributes to this implementation, includes three steps:

1) **IF** (<links>,</links>)↕<flow>, **THEN** <links>$\overset{\rightarrow}{\rightarrow}$<links><link name="name1"/>

ELSE <flow>$\overset{\rightarrow}{\rightarrow}$ <flow><links><link name="name1"/></links>

2) **activity-start-element$_a$**$\overset{\rightarrow}{\rightarrow}$ **activity-start-element$_a$**
<source linkName="name1"/>

3) **activity-start-element$_b$**$\overset{\rightarrow}{\rightarrow}$ **activity-start-element$_b$**
<target linkName="name1"/>

5.3.2.2 Unexpected Termination

The implementation of this fault model is

```
<flow standard-attributes>
    standard-elements
    links?
    activity_a
    activity_b
    activity*
    <termination/>
</flow>
```

The mutation process, which contributes to this implementation, is:

</flow>→<termination/></flow>

5.3.3 BPEL4WS examples

For the sake of simplicity, we inherit the two symbols <invoke> and <receive> defined in section 5.1.3 to describe the examples in this section.

5.3.3.1 Parallel to Sequential

We already know that for the workflow pattern Parallel Split, there are two kinds of implementation for its fault model Parallel to Sequential. Therefore we present a example for each kind of implementation.

For the first kind (using <sequence> activity) , if the original BPEL4WS program is of the structure given in listing 5.9(a), then the mutant of mutation process (defined in section 5.2.2.1) will possess the structure described in listing 5.9(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <receive> </receive> <invoke> </invoke> </flow>	<sequence> <receive> </receive> <invoke> </invoke> </sequence>

Listing 5.9

When we implement the fault model with a link construct, if the original BPEL4WS program is of the structure given in listing 5.10(a), then the mutant of mutation process (defined in section 5.2.2.1) will possess the structure described in listing 5.10(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow>	<flow>

<receive> </receive> <invoke> </invoke> </flow>	<links> <link name="R-To-A> </links> <receive> <source linkName="R-To-A"/> </receive> <invoke> <Target linkName="R-To-A"/> </invoke> </sequence>

Listing 5.10

5.3.3.2 Unexpected Termination

If the original BPEL4WS program is of the structure given in listing 5.11(a), the mutant of mutation process (defined in section 5.2.2.2) will possess the structure described in listing 5.11(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<*flow*> <receive> </receive> <invoke> </invoke> </*flow*>	<*flow*> <receive> </receive> <invoke> </invoke> <terminate/> </*sequence*>

Listing 5.11

5.4 Synchronization

5.4.1 Requirement of mutation process

For the Parallel workflow pattern, its representation in BNF of CSP_{BPEL} is:

(<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>

)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_PREFIX_OPER

ATOR><TRACE_SKIP_OPERATOR>)<TRACE_PREFIX_OPERATOR><EVENT_ID ENT3><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>

As mentioned in section 4.3, there are five major fault models for the Synchronization workflow pattern. Those fault models respectively are

- Synchronization to Sequence (FM6)

  <EVENT_IDENT1><TRACE_PREFIX_OPERATOR><EVENT_IDENT2><TR ACE_PREFIX_OPERATOR><EVENT_IDENT3>

  <TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR >

- Synchronization to Parallel (FM7)

  (<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPER ATOR>)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_ PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)

  <TRACE_ PARALLEL_OPERATOR> (<EVENT_IDENT3>

  <TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)

- Extra Single Condition (FM8)

  (<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPER ATOR>)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_ PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)<TRACE_PREFIX_OP ERATOR>< TRACE_IF_OPERATOR>(<Expression>)

  {<EVENT_IDENT3><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPER ATOR>}

- Extra AND Conditions (FM9)

  (<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPER

ATOR>)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)<TRACE_PREFIX_OPERATOR>< TRACE_IF_OPERATOR>((<Expression$_1$>) <AND>(<Expression$_2$>)){<EVENT_IDENT3><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}

- Extra OR Conditions (FM10)

(<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)<TRACE_PREFIX_OPERATOR>< TRACE_IF_OPERATOR>((<Expression$_1$>) <OR>(<Expression$_2$>)){<EVENT_IDENT3><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}

5.4.1.1 Synchronization to Sequence

The essence of the fault model Synchronization to Sequence is that the relationship between two activity events is changed from parallel to sequential so that the synchronization relationship among three activities becomes sequence. Therefore, the essence of this fault model is the same as the essence of fault model Parallel to Sequential. That Fault model has been previously explored in the workflow pattern Parallel Split. We have described the mutation process for that fault model in section 5.2.1.1. Therefore we don't repeat the discussion here.

5.4.1.2 Synchronization to Parallel

The essence of the fault model Synchronization to Parallel is that the relationship between two activity events is changed from sequential to parallel so that the synchronization relationship among three activities becomes parallel. Therefore, the essence of this fault model is the same as that of the Fault model Sequential to Parallel, which was discussed in the workflow pattern Sequence. Since we have described the mutation process for that fault model in section 5.1.1.1, we don't repeat it here.

5.4.1.3 Extra Single Condition

The essence of this fault model is that an extra condition is injected right after the synchronization point. If the condition is not satisfied, then the activity right after the condition won't be executed. Initially, that activity is expected to perform after the process reaches the synchronization point. The mutation process for this fault model is

<EVENT_IDENT3>→< TRACE_IF_OPERATOR> (<Expression>)
{<EVENT_IDENT3><TRACE_PREFIX_OPERATOR>
<TRACE_SKIP_OPERATOR>}

5.4.1.4 Extra AND Conditions

The essence of the fault models Extra AND Conditions is that two extra conditions are injected right after the synchronization point. Those two conditions have an AND relationship. If any of the conditions are not true, then the activity right after the conditions won't be executed. Initially, that activity is expected to execute after the process reaches the synchronization point. The mutation process for this fault model is

<EVENT_IDENT3>→< TRACE_IF_OPERATOR> ((<Expression$_1$>) <AND>
(<Expression$_2$>)){<EVENT_IDENT3>
<TRACE_PREFIX_OPERATOR>

<TRACE_SKIP_OPERATOR>}

5.4.1.5 Extra OR Conditions

The essence of the fault model Extra OR Conditions is that two extra conditions are injected right after the synchronization point. Those two conditions have an OR relationship. If both conditions are not true, then the activity right after the conditions won't be executed. Initially, that activity is expected to execute after the process reaches the synchronization point. The mutation process for this fault model is

<EVENT_IDENT3>→< TRACE_IF_OPERATOR> ((<Expression$_1$>) <OR>
(expression$_2$)){<EVENT_IDENT3>
<TRACE_PREFIX_OPERATOR>
<TRACE_SKIP_OPERATOR>}

5.4.2 Implementation of mutation process

For the Synchronization workflow pattern, there are two types of implementation. One is using the <sequence> activity (refer to listing 4.3), and the other one is using the link construct (refer to listing 4.4). For each type of implementation, we are going to illustrate the mutation process for each fault model.

5.4.2.1 Implementation using the <sequence> activity

For the implementation of the workflow pattern Synchronization, which uses the <sequence> activity, its BNF is

```
<sequence standard-attributes>
    standard-elements
    <flow standard-attributes>
        standard-elements
        activity_a
        activity_b
    </flow>
    activity_c
</sequence>
```

5.4.2.1.1 Synchronization to Sequence

An implementation of the fault model Synchronization to Sequence is:

```
<sequence standard-attributes>
    standard-elements
    <sequence standard-attributes>
        standard-elements
        activity_a
        activity_b
    </sequence>
    activity_c
</sequence>
```

The mutation process for this fault model is: flow $\vec{\rightarrow}$sequence

5.4.2.1.2 Synchronization to Parallel

An implementation of the fault model Synchronization to Parallel, is:

```
<flow standard-attributes>
    standard-elements
    <flow standard-attributes>
        standard-elements
        activity_a
        activity_b
    </flow>
    activity_c
</flow>
```

The mutation process for this fault model is: sequence $\vec{\rightarrow}$flow

5.4.2.1.3 Extra Single Condition

The fault model Extra Single Condition cannot be applied in the implementation using the <sequence> activity, since in a <sequence> activity, the link construct is not allowed. If we cannot use a link construct, then we are not allowed to inject transition conditions into the source or target element of an activity. Analogously, the other two types of fault

model, Extra AND Conditions and Extra OR Conditions cannot be applied to this type of implementation.

5.4.2.2 Implementation using the <link> construct

When we implement the workflow pattern Synchronization using the <link> activity, the BNF of the implementation is

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1"/> <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name2"/> <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition = "name1 AND name2"
    <target linkName="name1"/> <target linkName="name2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
</flow>
```

5.4.2.2.1 Synchronization to Sequence

An implementation of the fault model Synchronization to Sequence is:

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1"/> <source linkName="ncname1"
```

```
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/>
    <source linkName="name2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    <target linkName="name2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
</flow>
```

The mutation process for this fault model has three steps:

1) <target linkName="name1"/> $\vec{\to}$ Φ
2) <source linkName="name2"/> $\vec{\to}$ <target linkName="name1"/> <source linkName="name2"/>
3) joinCondition = "name1 AND name2" $\vec{\to}$ Φ

5.4.2.2.2 Synchronization to Parallel

An implementation of the fault model Synchronization to Parallel is:

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1"/> <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name2"/> <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition = "name1 AND name2"
    <target linkName="name1"/> <target linkName="name2"/>
    <target linkName="ncname2"/>*
```

```
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
</flow>
```

The mutation process for this fault model has five steps:

1) <source linkName="name1"/> $\overrightarrow{\exists}$ Φ
2) <source linkName="name2"/> $\overrightarrow{\exists}$ Φ
3) <target linkName="name1"/> $\overrightarrow{\exists}$ Φ
4) <target linkName="name1"/> $\overrightarrow{\exists}$ Φ
5) joinCondition = "name1 AND name2" $\overrightarrow{\exists}$ Φ

5.4.2.2.3 Extra Single Condition

An implementation of the fault model Extra Single Condition, is:

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1" transitionCondition ="bool-expr1"/>
    <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name2"/> <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition = "name1 AND name2"
    <target linkName="name1"/> <target linkName="name2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
</flow>
```

The mutation process for this fault model is:

<source linkName="name1"/> ⇉ <source linkName="name1"
transitionCondition ="bool-expr1"/>

5.4.2.2.4 Extra AND Conditions

An implementation of the fault model Extra AND Conditions is:

```
<flow standard-attributes>
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1" transitionCondition ="bool-expr1"/>
    <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name2" transitionCondition ="bool- expr2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition = "name1 AND name2"
    <target linkName="name1"/> <target linkName="name2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
</flow>
```

The mutation process for this fault model has two steps:

1) <source linkName="name1"/> ⇉ <source linkName="name1"
transitionCondition ="bool-expr1"/>

2) <source linkName="name2"/> ⇉ <source linkName="name2"
transitionCondition ="bool-expr2"/>

5.4.2.2.5 Extra OR Condition

An implementation of the fault model Extra OR Conditions is:

```
<flow standard-attributes>
    standard-elements
    <links>
```

```
        <link name="name1"/>
        <link name="name2"/>
        link*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1" transitionCondition ="bool-expr1"/>
    <source linkName="ncname1"
    transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <source linkName="name2" transitionCondition ="bool-expr2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition = "name1 OR name2"
    <target linkName="name1"/> <target linkName="name2"/>
    <target linkName="ncname2"/>*
    <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
</flow>
```

The mutation process for this fault model has three steps:

1) <source linkName="name1"/> → <source linkName="name1" transitionCondition ="bool-expr1"/>

2) <source linkName="name1"/> → <source linkName="name2" transitionCondition ="bool-expr2"/>

3) "name1 AND name2" → "name1 OR name2"

5.4.3 BPEL4WS examples

Again for the sake of simplicity, we defined three symbols, each of which represents a snippet of BPEL4WS. Then we use these symbols to illustrate the examples in this section. In the following equations, the symbol on the left is used to represent the BPEL4WS code on the right side.

<invoke_DecideOnShipper> =<invoke partnerLink="shipping"
name ="DecideOnShipper" portType="ship:shipping"
operation="requestShipping" inputVariable="shippingRequest"
outputVariable="shippingInfo">

<invoke$_{InitiatePriceCalculation}$>=<invoke partnerLink="invoicing"
name = "InitiatePriceCalculation" portType="inv:invoice"
operation="initiatePriceCalculation" inputVariable="PO"
outputVariable="Result">

<invoke$_{CompletePriceCalculation}$> =<invoke partnerLink="invoicing"
name = "CompletePriceCalculation" portType="inv:invoice"
operation="sendShippingPrice" inputVariable="shippingInfo"
outputVariable="Result">

5.4.3.1 **Synchronization to Sequence**

When the workflow pattern Synchronization is implemented using the <sequence> activity, if the original BPEL4WS program corresponds to listing 5.12(a), the mutant of mutation process (defined in section 5.3.2.1.1) has the structure shown in listing 5.12(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<sequence> <flow> <invoke$_{DecideOnShipper}$> </invoke> <invoke$_{InitiatePriceCalculation}$> </invoke> </flow> <invoke$_{CompletePriceCalculation}$> </invoke> </sequence>	<sequence> <sequence> <invoke$_{DecideOnShipper}$> </invoke> <invoke$_{InitiatePriceCalculation}$> </invoke> </sequence> <invoke$_{CompletePriceCalculation}$> </sequence> </flow>

Listing 5.12

When the workflow pattern Synchronization is implemented through link construct, if the original BPEL4WS program corresponds to listing 5.13(a), the mutant of mutation process (defined in section 5.3.2.2.1) has the structure shown in listing 5.13(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links>	<flow> <links>

<link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <source linkName=" calclulate-price"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>	<link name="ship-to-invoice"/> <link name="calculate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <target linkName="ship-to-invoice"/> <source linkName="calculate-price"/> </invoke> </flow> <invoke$_{CompletePriceCalculation}$> <target linkName="calculate-price"/> </invoke> </flow>

Listing 5.13

5.4.3.2 Synchronization to Parallel

When the workflow pattern Synchronization is implemented using the <sequence> activity, if the original BPEL4WS program corresponds to listing 5.14(a), the mutant of mutation process (defined in section 5.3.2.1.2) has the structure of listing 5.14(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<sequence> <flow> <invoke$_{DecideOnShipper}$> </invoke> <invoke$_{InitiatePriceCalculation}$> </invoke> </flow> <invoke$_{CompletePriceCalculation}$> </invoke> </sequence>	<flow> <flow> <invoke$_{DecideOnShipper}$> </invoke> <invoke$_{InitiatePriceCalculation}$> </invoke> </flow> <invoke$_{CompletePriceCalculation}$> </invoke> </flow>

Listing 5.14

When the workflow pattern Synchronization is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.15(a), the mutant of mutation process (defined in section 5.3.2.2.2) has the structure of listing 5.15(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <source linkName=" calclulate-price"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>	<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> </invoke> <invoke$_{InitiatePriceCalculation}$> </invoke> </flow> <invoke$_{CompletePriceCalculation}$> </invoke> </flow>

Listing 5.15

5.4.3.3 Extra Single Condition

When the workflow pattern Synchronization is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.16(a), the mutant of mutation process (defined in section 5.3.2.2.3) has the structure of listing 5.16(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$>	<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$>

<source linkName=" calclulate-price"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>	<source linkName=" calclulate-price" transitionCondition="bpws:getVariableData(Result, 'amount')<10"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>

Listing 5.16

5.4.3.4 Extra AND Conditions

When the workflow pattern Synchronization is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.17(a), the mutant of mutation process (defined in section 5.3.2.2.4) has the structure of listing 5.17(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <source linkName=" calclulate-price"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>	<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice" transitionCondition="bpws:getVariableData('ShippingInfo', "schedule")!=Null"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <source linkName=" calclulate-price" transitionCondition="bpws:getVariableData(Result, 'amount')<10"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>

Listing 5.17

5.4.3.5 Extra OR Condition

When the workflow pattern Synchronization is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.18(a), the mutant of mutation process (defined in section 5.3.2.2.5) has the structure of listing 5.18(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <source linkName=" calclulate-price"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 AND name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>	<flow> <links> <link name="ship-to-invoice"/> <link name="calclulate-price"/> </links> <invoke$_{DecideOnShipper}$> <source linkName=" ship-to-invoice" transitionCondition="bpws:getVariableData('ShippingInfo', 'schedule')!=Null"/> </invoke> <invoke$_{InitiatePriceCalculation}$> <source linkName=" calclulate-price" transitionCondition="bpws:getVariableData('Result', 'amount')<10"/> </invoke> <invoke$_{CompletePriceCalculation}$> joinCondition = "name1 OR name2" <target linkName=" ship-to-invoice"/> <target linkName=" calclulate-price"/> </invoke> </flow>

Listing 5.18

5.5 Exclusive Choice

5.5.1 Requirement of mutation process

For the workflow pattern Exclusive Choice, its representation in the BNF of CSP_{BPEL} is:

<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(

<Expression>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_

OPERATOR>}<TRACE_ELSE_OPERATOR>{<EVENT_IDENT3><TRACE_PREFI X_OPERATOR><TRACE_SKIP_OPERATOR>}

As we mentioned in section 4.3, there are two major fault models for the workflow pattern Exclusive Choice. Those fault models respectively are

- Switch Condition (FM11)

  <EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATO R>(<Expression>){<EVENT_IDENT3><TRACE_PREFIX_OPERATOR><TR ACE_SKIP_OPERATOR>}<TRACE_ELSE_OPERATOR>{<EVENT_IDENT2 ><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}

- Exclusive Choice to Parallel (FM12)

  <EVENT_IDENT1><TRACE_PREFIX_OPERATOR>(<EVENT_IDENT2><T RACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>) <TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT3><TRACE_PREFIX_ OPERATOR><TRACE_SKIP_OPERATOR>)

5.5.1.1 Switch Condition

The essence of the fault model Switch Condition is that two exclusive conditions, each of which determines the execution of a subsequent activity, are switched. Therefore, each subsequent activity executes in an invalid condition.

The mutation process for this fault model is:

<EVENT_IDENT2>⇄<EVENT_IDENT3>

5.5.1.2 Exclusive Choice to Parallel

The essence of the fault model Exclusive Choice to Parallel is that the exclusive conditions that determines the execution of subsequent activities are missed, therefore the subsequent activities can execute in parallel. Initially, only one subsequent activity is expected to execute.

The mutation process for this fault model has two steps:

1) <TRACE_IF_OPERATOR>(<Expression>) $\rightarrow \Phi$
2) <TRACE_ELSE_OPERATOR>$\rightarrow$<TRACE_PARALLEL_OPERATOR>

5.5.2 Implementation of mutation process

For the Sequence workflow pattern, there are two types of implementation. One is using the link construct, and the other one is using the <switch> activity. For each type of implementation, we are going to illustrate the mutation process for each fault model.

5.5.2.1 Implementation using the link construct

The BNF of the implementation of this workflow pattern is:

```
<flow standard-attributes >
    standard-elements
     <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1" transitionCondition="bool-expr1"/>
    <source linkName="name2" transitionCondition="bool-expr2" />
 <source linkName="ncname1" transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
```

```
    activity-start-element_c
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
    activity*
</flow>
```

5.5.2.1.1 Switch Condition

The implementation of the fault model Switch Conditions is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <source linkName="name1" transitionCondition="bool-expr_1"/>
    <source linkName="name2" transitionCondition="bool-expr_2" />
<source linkName="ncname1"  transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/> <target linkName="ncname2"/>*
 <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
    activity*
</flow>
```

The mutation process for this fault model is: Name1⇄name2

5.5.2.1.2 Exclusive Choice to Parallel

The implementation of the fault model Exclusive Choice to Parallel is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    activity-start-element_a
```

```
    <target linkName="ncname1"/>*
    <source linkName="name1""/>
    <source linkName="name2"" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name1"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_c
    activity*
</flow>
```

The mutation process for this fault model has two steps:

1) transitionCondition="bool-expr1" $\twoheadrightarrow$ Φ

2) transitionCondition="bool-expr2" $\twoheadrightarrow$ Φ

5.5.2.2 Implementation using the <switch> activity

The BNF of the implementation of this workflow pattern is:

```
<sequence>
    activity_a
    <switch standard-attributes >
        standard-elements
        <case condition="bool-expr_1" />
            activity_b
        <case condition="bool-expr2" />
            activity_c
        (<case condition="bool-expr" /> Activity)*
    </switch>
</sequence>
```

5.5.2.2.1 Switch Condition

The implementation of the fault model Switch Condition is:

```
<sequence>
    activity_a
    <switch standard-attributes >
        standard-elements
        <case condition="bool-expr_1" />
            activity_c
        <case condition="bool-expr2" />
```

```
                activity_b
            (<case condition="bool-expr" /> activity) *
        </switch>
    </sequence>
```

The mutation process for this fault model is: $activity_b \rightleftarrows acticity_c$

5.5.2.2.2 Exclusive Choice to Parallel

When the workflow pattern Exclusive Choice is implemented using the <switch> activity, the fault model Exclusive Choice to Parallel cannot be applied to it.

5.5.3 BPEL4WS examples

Again for the sake of simplicity, we defined three symbols, each of which represents a snippet of BPEL4WS. Then we use these symbols to illustrate the examples in this section. In the following equations, the symbol on the left is used to represent the BPEL4WS code on the right side.

```
<receive_receiveReuest> = <receive createInstance="yes" name="receiveRequest"
                operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="request">
```

```
<invoke_inovokeApprover>= <invoke inputVariable="request"
                name="invokeapprover" operation="approve"
                outputVariable="approvalInfo"
                partnerLink="approver" portType="apns:loanApprovalPT">
```

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

5.5.3.1 Switch Condition

When the workflow pattern Exclusive is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.19(a), the mutant of mutation process (defined in section 5.4.1.1) has the structure of listing 5.19(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="receive-to-approval"/> <link name="receive-to-assess"/> </links> <receive$_{receiveReuest}$> <source linkName="receive-to-approval" transitionCondition="bpws:getVariableData(' request', 'amount')>=10000"/> <source linkName="receive-to-assess" transitionCondition="bpws:getVariableData(' request', 'amount')<10000"/> </receive> <invoke$_{inovokeApprover}$> <target linkName="receive-to-approval"/> </invoke> <invoke$_{invokeAssessor}$> <target linkName="receive-to-assess"/> </invoke> </flow>	<flow> <links> <link name="receive-to-approval"/> <link name="receive-to-assess"/> </links> <receive$_{receiveReuest}$> <source linkName="receive-to-approval" transitionCondition="bpws:getVariableData ('request', 'amount')>=10000"/> <source linkName="receive-to-assess" transitionCondition="bpws:getVariableData ('request', 'amount')<10000"/> </receive> <invoke$_{inovokeApprover}$> <target linkName="receive-to-assess"/> </invoke> <invoke$_{invokeAssessor}$> <target linkName="receive-to-approval"/> </invoke> </flow>

Listing 5.19

When the workflow pattern Exclusive is implemented through the <switch> activity, if the original BPEL4WS program corresponds to listing 5.20(a), the mutant of mutation process (defined in section 5.4.2.1) has the structure of listing 5.20(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<sequence> <receive$_{receiveReuest}$> <switch> <case condition="bpws:getVariableData ('request', 'amount')>=10000"/>	<sequence> <receive$_{receiveReuest}$> <switch> <case condition="bpws:getVariableData ('request', 'amount')>=10000"/>

<invoke$_{inovokeApprover}$> </case> <case condition="bpws:getVariableData ('request', 'amount')<10000"/> <invoke$_{invokeAssessor}$> </case> </sequence>	<invoke$_{invokeAssessor}$> </case> <case condition="bpws:getVariableData ('request', 'amount')<10000"/> <invoke$_{inovokeApprover}$> </case> </sequence>

5.5.3.2 Exclusive Choice to Parallel

When workflow pattern Exclusive is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.21(a), the mutant of mutation process (defined in section 5.4.1.2) has the structure of listing 5.21(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow> <links> <link name="receive-to-approval"/> <link name="receive-to-assess"/> </links> <receive$_{receiveReuest}$> <source linkName="receive-to-approval" transitionCondition="bpws:getVariableData(' request', 'amount')>=10000"/> <source linkName="receive-to-assess" transitionCondition="bpws:getVariableData(' request', 'amount')<10000"/> </receive> <invoke$_{inovokeApprover}$> <target linkName="receive-to-approval"/> </invoke> <invoke$_{invokeAssessor}$> <target linkName="receive-to-assess"/> </invoke> </flow>	<flow> <links> <link name="receive-to-approval"/> <link name="receive-to-assess"/> </links> <receive$_{receiveReuest}$> <source linkName="receive-to-approval" /> <source linkName="receive-to-assess" /> </receive> <invoke$_{inovokeApprover}$> <target linkName="receive-to-assess"/> </invoke> <invoke$_{invokeAssessor}$> <target linkName="receive-to-approval"/> </invoke> </flow>

Listing 5.21

5.6 Simple Merge

5.6.1 Requirement of mutation process

For the workflow pattern Simple Merge, its representation in BNF of CSP_{BPEL} is:

<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_ELSE_OPERATOR>{<TRACE_IF_OPERATOR>(<Expression2>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}}

As we mentioned in section 4.5, there are two major fault models for the workflow pattern Simple Merge. Those fault models respectively are

- Switch Condition (FM11)

  <TRACE_IF_OPERATOR>(<Expression2>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_ELSE_OPERATOR>{<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}}

- Simple Merge to Multi-Choice (FM13)

  (<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})<TRACE_PARALLEL_OPERATOR>(<TRACE_IF_OPERATOR>(<Expression3>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})

5.6.1.1 Switch Condition

The essence of the fault model Switch Condition is that two conditions, each of which determines the execution of a subsequent activity, are switched. Therefore, each subsequent activity executes in an inappropriate condition.

The mutation process for this fault model is: <expression1>⇄<expression2>

5.6.1.2 Simple Merge to Multi-Choice

The Fault model Simple Merge to Multi-Choice can be implemented only when the workflow pattern Simple Merge is implemented through a link construct. In this implementation, the two transition conditions are exclusive. The essence of this fault model is that a transition condition is modified so that those two exclusive conditions become inclusive Therefore, it is possible that all the subsequent activities can execute in parallel. Initially, at most one subsequent activity is expected to execute.

The mutation process for this fault model has two steps:

1) <TRACE_ELSE_OPERATOR> ⇉ <TRACE_PARALLE_OPERATOR>

2) <expression2> ⇉ <expression3>

5.6.2 Implementation of mutation process

For the workflow pattern Simple Merge, there are two types of implementation. One is using the link construct, and the other one is using the <switch> activity. For each type of implementation, we will illustrate the mutation process for each fault model.

5.6.2.1 Implementation using link construct

The BNF of the implementation of this workflow pattern is:

```
<flow standard-attributes >
    standard-elements
```

```
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
     <source linkName="name1" transitionCondition="bool-expr1"/>
     <source linkName="name2" transitionCondition="bool-expr2"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

5.6.2.1.1 Switch Condition

The implementation of the fault model Switch Condition is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr2"/>
    <source linkName="name2" transitionCondition="bool-expr1"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model is: bool-expr2⇄bool-expr1

5.6.2.1.2 Simple Merge to Multi-Choice

The implementation of the fault model Switch Condition is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr1"/>
    <source linkName="name2" transitionCondition="bool-expr3"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model is: bool-expr2 ↛ bool-expr3

5.6.2.2 Implementation using the <switch> activity

The BNF of the implementation of this workflow pattern is:

```
<switch standard-attributes >
    standard-elements
    <case condition="bool-expr_1" />
        activity_a
    <case condition="bool-expr2" />
        activity_b
    (<case condition="bool-expr" />  activity)*
</switch>
```

5.6.2.2.1 Switch Condition

The implementation of the fault model Switch Condition is:

```
<switch standard-attributes >
    standard-elements
```

```
        <case condition="bool-expr2" />
            activity_a
        <case condition="bool-expr1" />
            activity_b
        (<case condition="bool-expr" /> activity) *
    </switch>
```

The mutation process for this fault model is: bool-expr2⇄bool-expr1

5.6.2.2.2 Simple Merge to exclusive Choice

When the workflow pattern Simple Merge is implemented using the <switch> activity, the fault model Exclusive Choice to Parallel cannot be applied to it.

5.6.3 BPEL4WS examples

Again for the sake of simplicity, we defined two symbols, each of which represents a snippet of BPEL4WS code. Then we use these symbols to illustrate the examples in this section. In the following equations, the symbol on the left is used to represent the BPEL4WS code on the right side.

```
<invoke_chargeCreditCard> = <invoke partnerLink="creditCardSerivce"
                              name ="chargeCreditCard"
                              portType="credit:creditCardService"
                              operation="chargeCreditCard"
                              inputVariable="totalCharge">

<invoke_payWithCredit>= <invoke partnerLink="creditSerivce"
                         name ="payWithCredit"
                         portType="credit:creditService"
                         operation="payWithCredit"
                         inputVariable="totalCharge">
```

5.6.3.1 **Switch Condition**

When the workflow pattern Simple Merge is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.22(a), the mutant of mutation process (defined in section 5.5.1.1) has the structure presented in listing 5.22(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow standard-attributes > standard-elements <links> <link name="creditCard"/> <link name="credit"/> <link>* </links> <empty> <source linkName="creditCard" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('balance','number')"/> <source linkName="credit" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('credit','number') AND bpws:getVariableData ('totalCharge','number')> bpws: getVariableData('balance','number')"/> </empty> <invoke_{chargeCreditCard}> <target linkName="creditCard" /> </invoke> <invoke_{payWithCredit}> <target linkName="credit"/> </invoke> </flow>	<flow standard-attributes > standard-elements <links> <link name="creditCard"/> <link name="credit"/> <link>* </links> <empty> <source linkName="creditCard" transitionCondition="="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('credit','number') AND bpws:getVariableData ('totalCharge','number')> bpws: getVariableData('balance','number')"/> <source linkName="credit" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('balance','number')"/> </empty> <invoke_{chargeCreditCard}> <target linkName="creditCard" /> </invoke> <invoke_{payWithCredit}> <target linkName="credit"/> </invoke> </flow>

Listing 5.22

When the workflow pattern Simple Merge is implemented through the <switch> activity, if the original BPEL4WS program corresponds to listing 5.23(a), the mutant of mutation process (defined in section 5.5.2.1) has the structure presented in listing 5.23(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<switch name="compareShippingPrice> standard-elements <case condition="bpws:getVariableData('totalCharge','number') <= bpws: getVariableData('balance','number')> <invoke_{chargeCreditCard}> </invoke> </case> <case	<switch name="compareShippingPrice> standard-elements <case condition="bpws:getVariableData('totalCharge','number') <= bpws: getVariableData('credit','number')> <invoke_{chargeCreditCard}> </invoke> </case> <case

condition="bpws:getVariableData('total Charge','number') <= bpws: getVariableData('credit','number')> <invoke payWithCredit> </invoke> </case> </switch>	condition="bpws:getVariableData('total Charge','number') <= bpws: getVariableData('balance','number')> <invoke payWithCredit> </invoke> </case> </switch>

Listing 5.23

5.6.3.2 Simple Merge to Multi-Choice

When the workflow pattern Simple Merge is implemented through the link construct, if the original BPEL4WS program corresponds to listing 5.24(a), the mutant of mutation process (defined in section 5.5.1.2) has the structure presented in listing 5.24(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow standard-attributes > standard-elements <links> <link name="creditCard"/> <link name="credit"/> <link>* </links> <empty> <source linkName="creditCard" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('balance','number')"/> <source linkName="credit" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('credit','number') AND bpws:getVariableData ('totalCharge','number')> bpws: getVariableData('balance','number')" /> </empty> <invoke chargeCreditCard> <target linkName="creditCard" /> </invoke> <invoke payWithCredit> <target linkName="credit"/> </invoke> </flow>	<flow standard-attributes > standard-elements <links> <link name="creditCard"/> <link name="credit"/> <link>* </links> <empty> <source linkName="creditCard" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('balance','number')"/> <source linkName="credit" transitionCondition="bpws:getVariableData('totalCharge','number')<= bpws:getVariableData('credit','number')"/> </empty> <invoke chargeCreditCard> <target linkName="creditCard" /> </invoke> <invoke payWithCredit> <target linkName="credit"/> </invoke> </flow>

Listing 5.24

5.7 Multi- Choice

5.7.1 Requirement of mutation process

For the workflow pattern Multi- Choice, its representation in BNF of CSP_{BPEL} is:

(<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})<TRACE_PARALLE_OPERATOR>(<TRACE_IF_OPERATOR>(<Expression2>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})

As we mentioned in section 4.6, there are three major fault models for workflow pattern Multi-Choice. Those fault models respectively are

- Switch Condition (FM11)

  (<TRACE_IF_OPERATOR>(<Expression2>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})<TRACE_PARALLE_OPERATOR>(<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})

- Multi-Choice to Simple Merge (FM14)

  <TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_ELSE_OPERATOR>{<TRACE_IF_OPERATOR>(<Expression3>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}}

- Multi-Choice to Parallel (FM15)

  (<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPER

ATOR>)<TRACE_PARALLEL_OPERATOR>(<EVENT_IDENT2><TRACE_

PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>)

5.7.1.1 Switch Condition

The essence of the fault model Switch Condition is that two conditions, each of which determines the execution of a subsequent activity, are switched. Therefore, each subsequent activity executes in an invalid condition.

The mutation process for this fault model is: <Expression1> ⇄ <Expression2>

5.7.1.2 Multi-Choice to Simple Merge

The essence of the fault model Multi-Choice to Simple Merge is that one transition condition is modified so that those two inclusive conditions become exclusive. Therefore, those two conditions can never be true at the same time, and at most one subsequent activity can execute at a time. Initially, it is possible that multiple subsequent activities execute in parallel.

The mutation process for this fault model is:

1) <TRACE_PARALLE_OPERATOR> ⇉ <TRACE_ELSE_OPERATOR>
2) <Expression2>⇉<Expression3>

5.7.1.3 Multi-Choice to Parallel

The essence of the fault model Multi-Choice to Parallel is that the transition conditions are missed so that the subsequent activities execute without preconditions.

The mutation process for this fault model is:

<TRACE_IF_OPERATOR> (Expression) ⇉ Φ

5.7.2 Implementation of mutation process

For the workflow pattern Multi-Choice, there is only one type of implementation. That is using the link construct. The BNF of the implementation of this workflow pattern is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr1"/>
    <source linkName="name2" transitionCondition="bool-expr2"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

5.7.2.1 Switch Condition

The implementation of the fault model Switch Condition is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr2"/>
    <source linkName="name2" transitionCondition="bool-expr1"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
```

```
    activity*
</flow>
```

The mutation process for this fault model is: bool-expr2⇄bool-expr1

5.7.2.2 Multi-Choice to Simple Merge

The implementation of the fault model Multi-Choice to Simple Merge is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr1"/>
    <source linkName="name2" transitionCondition="bool-expr3"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1"transitionCondition="bool-expr"? />*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model is: bool-expr2→bool-expr3

5.7.2.3 Multi-Choice to Parallel

The implementation of the fault model Multi-Choice to Parallel, is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1"/>
    <source linkName="name2"/>
    </empty>
```

```
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/> <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity*
</flow>
```

The mutation process for this fault model is

1) transitionCondition="bool-expr1" $\rightarrow \Phi$
2) transitionCondition="bool-expr2" $\rightarrow \Phi$

5.7.3 BPEL4WS examples

Again for the sake of simplicity, we defined two symbols, each of which represents a snippet of BPEL4WS code. Then we use these symbols to illustrate the examples in this section. In the following equations, the symbol on the left is used to represent the BPEL4WS code on the right side.

<invoke sendInvoiceViaMail> =
```
<invoke
 partnerLink="mail"
 name ="sendInvoiceViaMail"
 portType="mail:sendMail"
 operation="sendInvoiceViaMail"
 inputVariable="Invoice">
```

<invoke sendINvoiceViaEmail> =
```
<invoke
partnerLink="mail"
name ="sendINvoiceViaEmail"
portType="mail:sendMail"
operation="sendInvoiceViaEMail"
inputVariable="Invoice">
```

5.7.3.1 Switch Condition

If the original BPEL4WS program corresponds to listing 5.25(a), the mutant of mutation process (defined in section 5.6.1.1) has the structure presented in listing 5.25(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws:getVariableData('PO','needPaperIn voice') = 'Yes'" /> <source linkName=" Eamil-invoice" transitionCondition=" bpws:getVariableData('PO','needElectro nicInvoice') = 'Yes'"/> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> </invoke> <invoke pay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> </invoke> </flow>	< flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws: getVariableData('PO','needElectronicIn voice')= 'Yes'" /> <source linkName=" Eamil-invoice" transitionCondition=" bpws: getVariableData('PO','needPaperInvoice ') = 'Yes'"/> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> </invoke> <invoke pay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> </invoke> </flow>

Listing 5.25

5.7.3.2 Multi-Choice to Simple Merge

If the original BPEL4WS program corresponds to listing 5.26(a), the mutant of mutation process (defined in section 5.6.1.2) has the structure presented in listing 5.26(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws:getVariableData('PO','needPaperIn voice') = 'Yes'" /> <source linkName=" Eamil-invoice"	<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws:getVariableData('PO','needPaperIn voice') = 'Yes'" /> <source linkName=" Eamil-invoice"

transitionCondition=" bpws:getVariableData('PO','needElectro nicInvoice') = 'Yes'"/> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> </invoke> <invokepay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> </invoke> </flow>	transitionCondition=" bpws:getVariableData('PO','needElectro nicInvoice') = 'yes'" AND bpws:getVariableData('PO','needPaperIn voice') = 'no'" /> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> </invoke> <invokepay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> </invoke> </flow>

Listing 5.26

5.7.3.3 Multi-Choice to Parallel

If the original BPEL4WS program corresponds to listing 5.27(a), the mutant of mutation process (defined in section 5.6.1.3) has the structure presented in listing 5.27(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws:getVariableData('PO','needPaperIn voice') = 'Yes'" /> <source linkName=" Eamil-invoice" transitionCondition=" bpws:getVariableData('PO','needElectro nicInvoice') = 'Yes'"/> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> </invoke> <invokepay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> </invoke> </flow>	<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice"/> <source linkName="Eamil-invoice"/> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> </invoke> <invokepay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> </invoke> </flow>

Listing 5.27

5.8 Synchronizing Merge

5.8.1 Requirement of mutation process

For the workflow pattern Synchronizing Merge, its representation in BNF of CSP_{BPEL} is:

(<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT1><TRACE_PREFIX_
OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_PARALLE_OPERATOR>(<T
RACE_IF_OPERATOR>(<Expression2>){<EVENT_IDENT2><TRACE_PREFIX_OP
ERATOR><TRACE_SKIP_OPERATOR>}) <TRACE_PREFIX_OPERATOR>
<TRACE_IF_OPERATOR>((<Expression1>) <OR> <Expression2>))
{<EVENT_IDENT3><TRACE_PREFIX_OPERATOR>
<TRACE_SKIP_OPERATOR>} <TRACE_ELSE_OPERATOR>
{<TRACE_SKIP_OPERATOR>}

As we discussed in section 4.7, there is one major fault model for the workflow pattern Synchronizing Merge. Those fault models respectively is

- Synchronizing Merge to Synchronization with AND conditions (FM16)

  (<TRACE_IF_OPERATOR>(<Expression1>){<EVENT_IDENT1><TRACE_P
  REFIX_OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_PARALLE_O
  PERATOR>(<TRACE_IF_OPERATOR>(<Expression2>){<EVENT_IDENT2>
  <TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>})
  <TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(<
  Expression1>)<AND> (<Expression2>)) {<EVENT_IDENT3>
  <TRACE_PREFIX_OPERATOR> <TRACE_SKIP_OPERATOR>}
  <TRACE_ELSE_OPERATOR>{<TRACE_SKIP_OPERATOR>}

5.8.1.1 Synchronizing Merge to Synchronization with AND conditions

For the fault model Synchronizing Merge to Synchronization with AND Conditions, its essence is that for the activity after the merge point, the relationship between its preconditions changes from OR to AND. Therefore the activity after the merge point can only execute when all the activities before the merge point are finished.

The mutation process for this fault model is: <OR> ➔ <AND>

5.8.2 Implementation of mutation process

For the workflow pattern Synchronization Merge, there is only one type of implementation. That is using the link construct. The BNF of the implementation of this workflow pattern is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link name="name3"/>
        <link name="name4"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr1"/>
    <source linkName="name2" transitionCondition="bool-expr2"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
    <source linkName="name3"/>
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/>
    <source linkName="name4"/>
    <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition= "name3 OR name4"
    <target linkName="name3"/>
```

```
    <target linkName="name4"/>
    <target linkName="ncname3"/>*
  <source linkName="ncname3" transitionCondition="bool-expr"?/>*
    activity-end-element_c
    activity*
</flow>
```

5.8.2.1 Synchronizing Merge to Synchronization with AND conditions

The implementation of the fault model Synchronizing Merge to Synchronization with AND conditions is:

```
<flow standard-attributes >
    standard-elements
    <links>
        <link name="name1"/>
        <link name="name2"/>
        <link name="name3"/>
        <link name="name4"/>
        <link>*
    </links>
    <empty>
    <source linkName="name1" transitionCondition="bool-expr1"/>
    <source linkName="name2" transitionCondition="bool-expr2"/>
    </empty>
    activity-start-element_a
    <target linkName="ncname1"/>*
    <target linkName="name1" />
    <source linkName="name3"/>
  <source linkName="ncname1" transitionCondition="bool-expr"?/>*
    activity-end-element_a
    activity-start-element_b
    <target linkName="name2"/>
    <source linkName="name4"/>
    <target linkName="ncname2"/>*
  <source linkName="ncname2" transitionCondition="bool-expr"?/>*
    activity-end-element_b
    activity-start-element_c
    joinCondition= "name3 AND name4"
    <target linkName="name3"/>
    <target linkName="name4"/>
    <target linkName="ncname3"/>*
  <source linkName="ncname3" transitionCondition="bool-expr"?/>*
    activity-end-element_c
    activity*
</flow>
```

The mutation process for this fault model is: OR ⇉ AND

5.8.3 BPEL4WS examples

Again for the sake of simplicity, we use the two symbols defined in section 5.6.3 and define one more symbol in this section. Subsequently, we will use these tokens to illustrate the examples in this section. In the following equation, the symbol on the left is used to represent the BPEL4WS code on the right side.

<reply> =
```
<reply partnerLink="purchasing"
 name="InvoiceProcessing"
 portType="lns:purchaseOrderPT"
 operation="sendPurchaseOrder"
 variable="Invoice">
```

5.8.3.1 Synchronizing Merge to Synchronization with AND conditions

If the original BPEL4WS program corresponds to listing 5.28(a), the mutant of mutation process (defined in section 5.7.1.1) has the structure presented in listing 5.28(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link name="sendpaper-invoice"/> <link name="sendEmail-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws:getVariableData('PO','needPaperIn voice') = 'Yes'" /> <source linkName=" Eamil-invoice" transitionCondition=" bpws:getVariableData('PO','needElectro nicInvoice') = 'Yes'"/> </empty> <invoke sendInvoiceViaMail > <target linkName="Paper-invoice" /> <source linkName="sendPaper-invoice"/> </invoke> <invokepay sendINvoiceViaEmail > <target linkName="Eamil-invoice"/> <source linkName="sendEmail-invoice"/> </invoke>	<flow standard-attributes > standard-elements <links> <link name="Paper-invoice"/> <link name="Eamil-invoice"/> <link name="sendpaper-invoice"/> <link name="sendEmail-invoice"/> <link>* </links> <empty> <source linkName="Paper-invoice" transitionCondition=" bpws:getVariableData('PO','needPaperIn voice') = 'Yes'" /> <source linkName=" Eamil-invoice" transitionCondition=" bpws:getVariableData('PO','needElectro nicInvoice') = 'Yes'"/> </empty> <invoke sendInvoiceViaMail > <target linkName=" Paper-invoice" /> <source linkName="sendPaper-invoice"/> </invoke> <invokepay sendINvoiceViaEmail > <target linkName=" Eamil-invoice"/> <source linkName="sendEmail-invoice"/> </invoke>

```
  <reply>
   joinCondition="sendPaper-invoice OR
sendEmail-invoice"
 <target linkName="sendPaper-invoice">
 <target linkName="sendEmail-invoice">
 </reply>
</flow>
```

```
  <reply>
   joinCondition="sendPaper-invoice
AND sendEmail-invoice"
 <target linkName="sendPaper-invoice">
 <target linkName="sendEmail-invoice">
 </reply>
</flow>
```

Listing 5.28

5.9 Deferred Choice

5.9.1 Requirement of mutation process

For the workflow pattern Deferred Choice, its representation in BNF of CSP_{BPEL} is:

<TRACE_WHILE_OPERATOR>(<BPEL_WATI_NAME><LT><BPEL_WAIT_TIME>){<TRACE_IF_OPERATOR>(<BPEL_MESSAGE_NAME><EQ><BEPL_MESSAGE>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_ELSE_OPERATOR>{<TRACE_IF_OPERATOR>(<BPEL_MESSAGE_NAME><EQ><BEPL_MESSAGE>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}}<TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(<BPEL_WATI_NAME><GE><BPEL_WAIT_TIME>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}

As we discussed in section 4.9, there is one major fault model for the workflow pattern Deferred Choice. Those fault models respectively is

- Missing Alarm (FM17)

  <TRACE_WHILE_OPERATOR>(<BPEL_WATI_NAME><LT><BPEL_WAIT_TIME>){<TRACE_IF_OPERATOR>(<BPEL_MESSAGE_NAME><EQ><BEPL_ME

SSAGE>){<EVENT_IDENT1><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}<TRACE_ELSE_OPERATOR>{<TRACE_IF_OPERATOR>(<BPEL_MESSAGE_NAME><EQ><BEPL_MESSAGE>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}}<TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(<BPEL_WATI_NAME><GE><BPEL_WAIT_TIME>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>}

5.9.1.1 Missing Alarm

For the fault model Missing Alarm, its essence is that while the running BPEL4WS process is waiting for the incoming message, there is no deadline set up. Therefore, if the expected message doesn't come, the running process will wait forever.

The mutation process for this fault model is:

1) <BPEL_WATI_NAME><LT><BPEL_WAIT_TIME>↛ <TRUE>

2) <TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(<BPEL_WATI_NAME><GE><BPEL_WAIT_TIME>){<EVENT_IDENT2><TRACE_PREFIX_OPERATOR><TRACE_SKIP_OPERATOR>} ↛ Φ

5.9.2 Implementation of mutation process

For the workflow pattern Deferred Choice Merge, there is only one type of implementation. That is using the pick construct. The BNF of the implementation of this workflow pattern is:

<pick>

<onMessage name="message1" partnerLink="ncname1" portType="qname1"
operation="ncname1"variable="ncname1">
activity$_a$
</onMessage>
<onMessage name="message2" partnerLink="ncname2" portType="qname2"
operation="ncname2" variable="ncname2">
activity$_b$
</onMessage>
onMessage*
onAlarm
</pick>

5.9.2.1 Missing Alarm

The implementation of the fault model Missing Alarm is:

<pick>
<onMessage name="message1" partnerLink="ncname1" portType="qname1"
operation="ncname1"variable="ncname1">
activity$_a$
</onMessage>
<onMessage name="message2" partnerLink="ncname2" portType="qname2"
operation="ncname2" variable="ncname2">
activity$_b$
</onMessage>
onMessage*
</pick>

The mutation process for this fault model is: onAlarm $\rightarrow$ Φ

5.9.3 BPEL4WS examples

Again for the sake of simplicity, we define three symbols in this section. Subsequently, we will use these tokens to illustrate the examples in this section. In the following equations, the symbol on the left is used to represent the BPEL4WS code on the right.

<invoke$_{addItem}$> = <invoke partnerLink="purchasing"
portType="lns:purchaseOrderPT"
operation="addItem"
variable="item">

<invoke$_{completeOrder}$> = <invoke partnerLink="purchasing"
portType="lns:purchaseOrderPT"
operation="completeOrder"

```
                    variable="invoice">

<reply_timeout>     =_<reply partnerLink="purchasing"
                    portType="lns:purchaseOrderPT"
                    operation="sendPurchaseOrder"
                    variable="timeOutMessage">
```

5.9.3.1 Synchronizing Merge to Synchronization with AND conditions

If the original BPEL4WS program corresponds to listing 5.29(a), the mutant of mutation process (defined in section 5.8.1.1) has the structure presented in listing 5.29(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)
<pick> <onMessage partnerLink="buyer" portType="orderEntry" operation="inputLineItem" variable="lineItem"> <invoke$_{addItem}$> </onMessage> <onMessage partnerLink="buyer" portType="orderEntry" operation="orderComplete" variable="completionDetail"> <invoke$_{completeOrder}$> </onMessage> <reply$_{timeout}$> <onAlarm for="'P3DT10H'"> </onAlarm> </pick>	<pick> <onMessage partnerLink="buyer" portType="orderEntry" operation="inputLineItem" variable="lineItem"> <invoke$_{addItem}$> </onMessage> <onMessage partnerLink="buyer" portType="orderEntry" operation="orderComplete" variable="completionDetail"> <invoke$_{completeOrder}$> </onMessage> </pick>

Listing 5.29

5.10 Cancel Activity

5.10.1 Requirement of mutation process

For the workflow pattern Cancel Activity, its representation in BNF of CSP_{BPEL} is:

<EVENT_IDENT1.b><TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(<BPEL_FAULT><EQ><TRUE>){<TRACE_IF_OPERATOR><BPEL_FAULT_NAME><EQ><FAULT1><AND>(<BPEL_FAULT_VAR><EQ><VAR1>)){<EVENT-IDNET2>}<TRACE_PREFIX_OPERATOR><TRACE_STOP_OPERATOR> }

<TRACE_ELSE_OPERATOR>{EVENT-IDENT1.e>

<TRACE_PREFIX_OPERATOR><TRACEN_SKIP_OPERATOR>}

As we discussed in section 4.10, there is one major fault model for the workflow pattern Cancel Activity. Those fault models respectively is

- Incorrect Fault Matching (FM18)

<EVENT_IDENT1.b><TRACE_PREFIX_OPERATOR><TRACE_IF_OPERATOR>(<BPEL_FAULT><EQ><TRUE>){<TRACE_IF_OPERATOR><BPEL_FAULT_NAME><EQ><FAULT2><AND>(<BPEL_FAULT_VAR><EQ><VAR2>)){<EVENT-IDNET2>}<TRACE_PREFIX_OPERATOR><TRACE_STOP_OPERATOR> }

<TRACE_ELSE_OPERATOR>{EVENT-IDENT1.e>

<TRACE_PREFIX_OPERATOR><TRACEN_SKIP_OPERATOR>}

5.10.1.1 Incorrect Fault Matching

For the fault model Incorrect Fault Matching, its essence is that the fault type or data variable, which is used to determine what fault should be fault, is modified incorrectly. Therefore, if the expected fault is thrown, it won't be caught. The mutation process for this fault model is:

1) <FAULT1>→<FAULT2>
2) <VAR1>→<VAR2>

5.10.2 Implementation of mutation process

For the workflow pattern Cancel Activity, there is only one type of implementation. That is using the fault handler construct. Its BNF of the implementation is:

<scope>

```
<faultHandlers>
  <catch faultName="qname1" faultVariable="ncname1">
    activity_a
  </catch>
</faultHandlers>
 activity_b
</scope>
```

5.10.2.1 Incorrect Fault Matching

The implementation of the fault model Incorrect Fault Matching is:

```
<scope>
  <faultHandlers>
    <catch faultName="qname2" faultVariable="ncname2">
      activity_a
    </catch>
  </faultHandlers>
   activity_b
</scope>
```

The mutation process for this fault model is :

1) qname1 ⇉ qname2
2) ncname1 ⇉ ncname2

5.10.3 BPEL4WS examples

5.10.3.1 Incorrect Fault Matching

If the original BPEL4WS program corresponds to listing 5.30(a), the mutant of mutation process (defined in section 5.9.1.1) has the structure presented in listing 5.30(b)

Original BPEL4WS program (a)	Mutant BPEL4WS program (b)

`<scope>`
`<faultHandlers>`
`<catch`
`faultName="lns:cannotCompleteOrder"`
`faultVariable="POFault">`
`<reply partnerLink="purchasing"`
`portType="lns:purchaseOrderPT"`
`operation="sendPurchaseOrder"`
`variable="POFault"`
	`<scope>`
	`<faultHandlers>`
	`<catch`
	`faultName="lns:invalidOrder"`
	`faultVariable="orderFault">`
	`<reply partnerLink="purchasing"`
	`portType="lns:purchaseOrderPT"`
	`operation="sendPurchaseOrder"`
	`variable="POFault"`

```
    faultName="cannotCompleteOrder"/>
  </catch>
 </faultHandlers>
 <invoke partnerLink="Seller"
  portType="SP:Purchasing"
  operation="SyncPurchase"
  inputVariable="sendPO"
  outputVariable="getResponse">
 </invoke>
</scope>
```

```
    faultName="cannotCompleteOrder"/>
  </catch>
 </faultHandlers>
 <invoke partnerLink="Seller"
  portType="SP:Purchasing"
  operation="SyncPurchase"
  inputVariable="sendPO"
  outputVariable="getResponse">
 </invoke>
</scope>
```

Listing 5.30

5.11 Conclusion

In this chapter, for each workflow pattern supported by the BPEL4WS, except workflow patterns Implicit Termination and Cancel Case, we discussed the requirement and implementation of its every fault model. We didn't discuss the workflow pattern Implicit Termination since it doesn't have observable trace behavior as we analyzed in section 4.8. We didn't discuss the workflow pattern Cancel Case since its only fault model is identical to the fault model 2 in workflow pattern Sequence, which we already mentioned in section 4.11.

The requirement of a fault model was described in BNF of CSP_{BPEL}. The BNF representation of a fault model means that the fault model is generic rather than just applicable to a specific example. The implementation of a fault model was described in BNF of BPEL4WS. The BNF representation of an implementation ensures that it is generic and applies to a type of BPEL4WS program rather than a specific one. Since a workflow pattern may have different types of implementation, when we explored the implementation of a fault model, we explored it responding to a specific type of workflow pattern implementation. Besides discussing the fault model itself, we also explored the mutation process of a fault model, which is used to generate mutant from

original BPEL4WS program. Further, we provided an example BPEL4WS program for each fault model.

From the discussion above, we can conclude that the definition of fault model is complete, generic and realistic. It can serve as criteria to evaluate the effectiveness and efficiency of our BPEL4WS run time monitoring system. Furthermore, in actual work, the mutation processes can greatly help us to generate mutants from original BPEL4WS program.

6 Empirical Investigation and Results

6.1 Overview

The investigation of the effectiveness of a testing strategy lacks formal analytical methods. In general, experimental evaluation is required. In this thesis, we designed two experiments to test the fault-detection capability of the proposed BPEL4WS run time monitoring system. For experiment, we initially tried to locate a long running and large sized BPEL4WS process, which is being used by an organization. Long running is a major characteristic of a BPEL4WS process, and comparing to a small sized BPEL4WS process, a large sized BPEL4WS process can be injected with more faults. Hence, testing a long running and large sized BPEL4WS process can more efficiently examine the testing capability of our proposed system. However, after we conducted a thorough searching through the available resources, we couldn't locate such a BPEL4WS process. The reason might be that BPEL4WS is a fairly new technology and it is still not widely applied in industry. Since we can not directly utilize an existed BPEL4WS process, we

decided to build our own experimental processes. When we built our own process, we tried to make it sufficiently large so that it could contain several workflow patterns. Since all the fault models we analyzed are based on the workflow patterns, more workflow patterns imply more faults can be injected into the BPEL4WS processes. Consequently, we implemented two experimental BPEL4WS processes, and a large number of faults were systematically seeded into the BPEL4WS processes. The BPEL4WS processes were executed upon our BPEL4WS runtime monitoring system to see how many faults can be detected.

The steps to conduct the experiment are:

- ***Define the trace specification of the experimental process***
  The first step is to define the expected trace behavior of the experimental process. The expected trace behavior is defined as a trace specification. The trace specification is used to control the instrumentation of the monitoring code. Furthermore, it is translated into a serial of activity events and then those activity events are compared with the actual activity events in execution to determine if there is an error in the execution.
- ***Create the mutants for the experiment process***
  The concept of mutant here is the same as in mutant testing, in which each process where a fault is seeded is referred to as a mutant. In general, in order to avoid interaction effects between faults, each time only one fault is seeded into the process. A mutant is killed when a test case cause it to fail [OFF95]. A mutant is killed means that the test strategy is effective to catch the fault in the mutant. The more mutants are killed, the higher effectiveness our test strategy has.

Therefore, in this experiment, the quantity and quality of mutants significantly affect the effectiveness of the evaluation. In order to get the best evaluation result, a large amount of systematic mutants are required to be seeded into the process under test

- ***Instrument monitoring code into the mutants***
  The trace specification is written in a text file. That file is an input for the instrument system. With the trace specification and mutant processes, the instrument system of our test framework instruments monitoring code into each mutant process. After being instrumented with monitoring code, the filename of the mutant process is not changed.

- ***Deploy and execute mutants***

  After the monitoring code is instrumented into a mutant process, the mutant process is deployed into a BPEL4WS engine manually or by some deployment toolkit. Then we build a client program to invoke the execution of the mutant process

- ***Analyze and evaluate the result of execution***
  During the execution of all the mutant process, the test results are written into log files by the trace verification system of our test framework. The test results are then analyzed to evaluate the effectiveness of the proposed BPEL4WS runtime monitoring system

6.2 Experiment 1

6.2.1 Introduction

The first experimental BPEL4WS process is a Purchase-order processing application.

On receiving the purchase order from a customer, the process initiates three tasks concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer.

The trace specification of this experimental process is

receivePurchaseOrder→ (((quoteShipper1→SKIP)||(quoteShipper2→SKIP))
→IF(shipperInfo1.price<=shipperInfo2.price){useShipper1}ELSE{useShipeer2}→((che
ckBalance→SKIP)||(checkCredit→SKIP))
→IF(totalChange.number<totalCredit.number){payWithCredit }ELSE
{IF(totalCharge.number<totalBalance.number){payWithBalance }ELSE {errorPay }} →
ArrangeLogistics → SKIP)||
(InitiatePirceCalculation→CompletePriceCalculation→SKIP)
||(InitiateProducationScheduling →CompleleteProductionScheduling →SKIP))
→(IF(po.needPaperInvoce=="yes"){sendPaperInvoice→SKIP})||(IF(po.needElectronicI
nvoice=="yes"){sendElectronicInvoice→SKIP})→replyPurchaseOrder→STOP

6.2.2 Mutants and Test Cases

After defining the trace specification of the experimental process, we designed mutants for it. As we discussed previously, currently there is little work on mutant operators specifically to BPEL4WS process, hence we use the mutation processes defined in

chapter 5 to generate mutants. This experimental BPEL4WS process doesn't implement all the workflow patterns, so we can only apply to it those mutation processes, which belong to the workflow patterns it implemented. The workflow patterns implemented in this experimental process are Sequence, Parallel Split, Synchronization, Simple Merge and Multi-Choice and Synchronizing Merge. According to the fault models explored in chapter 5, we can apply fault models FM 1, FM2, FM3, FM4, FM5, FM6, FM8, FM9, FM10, FM11, FM14, FM15 and FM16. Fault models FM7, FM12 and FM13 belong to other workflow patterns implemented in this experimental process, however those workflow patterns have more than one type of implementation and the implementations of those workflow patterns in the experimental process are not suitable to FM7, FM12 and FM13. Therefore, FM7, FM12 and FM13 can not be applied.

For each fault model that can be applied to this experimental process, we generate one mutant according to the mutation process defined in chapter5. Therefore, in total we generated 13 mutants. In addition, we generated 5 more mutants using some frequently used mutant operators in a general programming language. Those mutant operators respectively are "Mathematics operators exchanged"

"Variable by variable replacement", "Increment/decrement variables/constants" and "Output missing" [SOU00]. The details of all the mutants are in Table E.1 in appendix E. Our run time monitoring system doesn't specifically target at those errors. We introduced those mutants mainly in order to validate the ability of our system to catch some common errors in a general programming language.

The test cases were designed to cover all the branches in the experimental process. We tried to design as few as possible test cases to cover all the branches to minimize the execution of the test cases. Finally we generated four test cases for this experimental process. These test cases differed in the message exchanged between the process and external Web services, and those different messages made the process execute in different branches.

6.2.3 Test Execution Results and Analysis

After generating mutants and defining test cases, we executed the test cases against the mutant injected BPEL4WS process. As we stated previously, a mutant is killed when at least one of the test cases causes it to fail. On the other side, a mutant is "alive" when none of the test cases causes it to fail. The results of the experiment are shown in Table 6.1. In Table 6.1, the columns TC1 to TC4 represent the test cases. Each row represents one mutant. The "*" sign means that the test case is evaluated to "fail" and the mutant is killed, and the "-" sign means that the test case is evaluated to "succeed" and the mutant is "alive". Table 6.2 summarizes the data shown in Table 6.1.

	TC1	TC2	TC3	TC4
Mutant1	*	*	*	*
Mutant2	*	*	*	*
Mutant3	*	*	*	*
Mutant4	*	*	*	*
Mutant5	*	*	*	*
Mutant6	*	*	*	*

Mutant7	-	*	-	*
Mutant8	*	*	-	*
Mutant9	-	*	-	-
Mutant10	*	-	-	*
Mutant11	-	*	-	-
Mutant12	*	-	*	*
Mutant13	*	-	*	*
Mutant14	-	-	-	-
Mutant15	-	-	-	-
Mutant16	-	-	-	-
Mutant17	-	-	-	-
Mutant18	-	*	*	-

Table 6.1 The Detailed Results of Experiment 1

Fault Type	Total Mutants	Live Mutants	Killed Mutants	Equivalent Mutants	Killed/Total
Fault models of Workflow patterns	13	0	13	0	100%
Common programming fault	5	4	1	0	20%

Table 6.2 The Summary of Experiment 1

Upon analyzing the data from the Table 6.1 and Table 6.2, the following behavior was discovered. Firstly, there were six workflow patterns implemented in this experimental

process, and 100% of mutants responding to the fault modes of these workflow patterns were killed. This partially proves that our run time monitoring system provides sufficient coverage to investigate the specified defect types. When we looked at the details about how every mutant was killed by the test cases, we found that some mutants were killed by any test case, such as the mutant for the fault model Sequence to Parallel, and some mutants were only killed by just one test case, such as the mutant for the fault model Multi-Choice to Simple Merge. As we stated previously, different test cases differed in the messages exchanged between the process and external Web services. BPEL4WS processes have a characteristic of long running. During the running period, the messages exchanged between the process and external Web services changes as time goes on. This message changing has the same effect as different test cases, therefore if there is a fault of workflow pattern existing in the BPEL4WS process, it can be caught by our run time monitoring system.

Secondly, only 20% of mutants responding to common programming faults were killed. It indicated that our approach doesn't provide coverage to catch the common programming faults. As we stated before, our system doesn't target at this type of fault, hence this result was expected. The reason of the low coverage of this type of fault is that our system does not provide facilities to monitor the pre and post conditions for each activity. Many of common programming faults are value related, for example "Mathematics operator exchanged", "Variable by variable exchanged" and "Increment/decrement variables/constants". If the fault is not able to cause failure of trace, then our approach couldn't catch that fault. Therefore, in order for our system to

improve its coverage of common programming faults, we should introduce mechanisms to monitor the variable values in the pre and post conditions of activities.

6.3 Experiment 2

6.3.1 Introduction

Experiment 1 doesn't implement all the workflow patterns supported by BPEL4WS. Further, for the workflow patterns, which have more than one type of implementation, experiment 1 doesn't have all types of implementation for them. In order to fully explore the ability of our system to cover faults within workflow patterns, we designed experiment 2. Experiment 2 implemented some workflow patterns which were not implemented experiment 1. Further, experiment 2 has different types of implementation for some workflow patterns which were found in experiment 1.

Experiment 2 is an auto insurance claim processing application. On receiving the auto insurance claim for an accident from a client, the process calculates the total expense of the accident, including the auto repairing expense and the victim's medical expense. Depending on the total expense and the police report on the accident, the process invokes the appropriate external Web service to calculate the new insurance rate. After that the payment is made through direct deposit or a mailing check. Finally a report is sent back to the client.

The trace specification of experiment 2 is:

ReceiveInsuranceClaim→

retrieveAccountInfo→((hospitalExpense→SKIP)||(repairExpense→SKIP)) →

getPoliceReport→ readPoliceReport→ ((IF(claim.totalExpense>1000){

newInsuranceRate1→SKIP })|| (IF((claim.totalExpense<=1000)&

(claim.totalExpense>200)) {newInsuranceRate2→SKIP })) → ((

IF(claim.directDeposite=='yes') {directDeposit→SKIP })||(IF(claim.directDeposite

!='yes'){mailCheck→SKIP }))→STOP

6.3.2 Mutants and Test Cases

After defined the trace specification of the experimental process, we designed mutants for it. We used the same approach discussed in experiment 1 to generate mutants for experiment 2. The workflow patterns implemented in experiment 2 are Sequence, Synchronization, Simple Merge, Exclusive Choice, Deferred Choice and Cancel Activity. According to the fault models explored in chapter 5, we can apply fault models FM 1, FM2, FM4, FM6, FM7, FM11, FM12, FM13, FM17 and FM18. Although we implemented the workflow pattern Sequence in experiment 2 and the fault model Switch Two Activity Events (FM3) responds to it, we couldn't apply this fault model since it causes a syntactic error. Analogously, we implemented the workflow pattern Synchronization and the fault models FM8, FM9 and FM10, which respond to this workflow pattern. While it looks like we can apply those fault models to experiment 2, fault models FM8, FM9 and FM10 only apply to the implementation which uses the link construct. In experiment 2, the implementation of the workflow pattern Synchronization uses activities <sequence> and <flow> rather than the link construct, so fault models FM8, FM9 and FM10 couldn't be applied to experiment 2. Therefore, in total we generated 10 mutants upon fault models of workflow patterns. In addition, we generated

5 more mutants using some frequently used mutant operators in a general programming language. The details of all the mutants are shown in Table E.3 in appendix E. Using the same approach as in experiment 1 we generated four test cases in experiment 2.

6.3.3 Test Execution Results and Analysis

As before, the results of this experiment are shown in Table 6.3. Table 6.4 summarizes the data shown in Table 6.3.

	TC5	TC6	TC7	TC8
Mutant19	*	*	*	*
Mutant20	*	*	*	*
Mutant21	*	*	*	*
Mutant22	*	*	*	*
Mutant23	*	*	*	*
Mutant24	*	*	*	*
Mutant25	*	*	*	*
Mutant26	-	*	*	-
Mutant27	*	*	-	-
Mutant28	*	*	-	-
Mutant29	-	-	-	-
Mutant30	-	-	-	-
Mutant31	-	-	*	*
Mutant32	*	*	-	-

Mutant33	-	-	-	-

Table 6.3 The Detailed Results of Experiment 2

Fault Type	Total Mutants	Live Mutants	Killed Mutants	Equivalent Mutants	Killed/Total
Fault models of Workflow patterns	10	0	10	0	100%
Common programming fault	5	3	2	0	40%

Table 6.4 The Summary of Experiment 2

Upon analyzing the data from the Table 6.3 and Table 6.4, we found that 100% of mutants responding to the workflow pattern faults were killed. So far, we have validated our run time monitoring system against all 18 fault models of workflow pattern, and all 18 fault models were caught. This result empirically demonstrates that our system provides coverage across all of our defined fault models. With respect to the common programming faults, again the system performed poorly in line with expectations. In experiment2 only a small percentage (40%) of the mutants were killed.

7 Conclusion and future works

This thesis presents a systematic and effective approach to monitor the chronological behaviour of BPEL4WS processes. Additionally, we analyze the trace behaviour of eleven major workflow patterns supported by BPEL4WS, and the possible fault models

associated with these behaviours. In order to verify the usefulness and effectiveness of our approach, two experimental BPEL4WS applications were designed, and faults, generated based on the fault models of workflow patterns, were injected into those experimental applications. Further, some not chronological oriented faults, not generated based on the fault models, were also injected into those experimental applications. As a result, in two experiments 100% of all 23 chronological oriented faults were caught, while only about 30% of all 10 not chronological oriented faults were caught.

From the empirical investigation, we find that our proposed methodology has several distinct advantages, specifically:

1. Our system is able to monitor the execution trace of BPEL4WS and catch the majority of faults. This feature has not been implemented in other research work on this topic.
2. Our system is embedded with a BPEL4WS engine, so that it doesn't affect the performance of BPEL4WS processes under test.
3. Monitoring code based on the expected trace behaviour can be automatically instrumented into the BPEL4WS processes under test.

Additionally, we systematically analyze the trace behaviour of eleven major workflow patterns supported by BPEL4WS and we also analyze the possible fault models of those workflow patterns. This analysis is again a unique contribution and provides a basic for fault injection testing of BPEL4WS processes from the perspective of chronological behaviour.

Several areas of possible future research are presented below:

1. Monitoring the correlation among multi instances of a BPEL4WS process is not explored in this thesis. Correlation is an important feature in BPEL4WS and it is critical to implement stateful business process, therefore exploring how to monitor its behaviour is a good candidate for our future research.

2. In our proposed approach, the implementation of language CSP_{BPEL} is still primitive. Currently it is not very easy to define the expected trace behaviour of a complex BPEL4WS process. Designing a more powerful CSP_{BPEL} language is included in our future research. We even hope our approach can provide a facility for automatic creation of the trace specification from the design specification of BPEL4WS.

3. Our proposed approach focuses on monitoring the execution trace of BPEL4WS process. However, it has limitations in exposing those faults that are not chronological oriented. Hence our future research includes exploring how to enhance that ability in our approach.

Bibliography

[AAL00] Wil M. P. van der Aalst. Workflow verification: Finding Control-flow Errors Using Petri-net-based Techniques. Business Process Management: Models, Techniques, and Empirical Studies, Lecture Notes in Computer Science, Springer-Verlag, volume 1806, pp. 161-183, 2000

[AAL02] Wil M. P. van der Aalst, Arthur H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages, Proceedings of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, volume 560, pp. 1-20, 2002

[ACT01] ActiveBPEL Engine. Online at: http://www.active-endpoints.com/open-source-documentation.htm; accessed on March 30, 2007

[BAR05] Luciano Baresi, Sam Guinea. Towards Dynamic Monitoring of WS-BPEL Processes, Proceedings of the Third International Conference of Service-oriented Computing, pp. 269–282, 2005

[BNF01] Introduction to BNF, online at: http://en.wikipedia.org/wiki/Backus-Naur_form; accessed on March 30, 2007

[BOX00] Don Box, David Ehnebuske. Simple Object Access Protocol (SOAP) 1.1, online at: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ ; accessed on March 30, 2007

[BPW01] Introduction to bpws4j, online at: http://www.alphaworks.ibm.com/tech/bpws4j/download; accessed on March 30, 2007

[BRU02] Mark Brorkens. Jassda Trace Assertions, Runtime Checking the Dynamic of Java Programs, International Conference on Testing of Communicating Systems, pp. 39-48, March 2002

[CHR01] Erik Christensen, Francisco Curbera. Web Services Description Language (WSDL) 1.1, online at http://www.w3.org/TR/wsdl; accessed on March 30, 2007

[CSP01] Introduction to CSP, online at: http://en.wikipedia.org/wiki/Communicating_sequential_processes ; accessed on March 30, 2007

[DEL04] Nelly Delgado. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools, IEEE Transactions on Software Enigneering, Volumne 30, Issue 12, pp. 859-872, December 2004

[DUM01] Marlon Dumas and Arthur H.M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language, Proceedings of the 4th International Conference on the Unified Modeling Language, pp. 76-90, 2001

[FIS00] Clemens Fischer. Combination and Implementation of Processes and Data: From CSP-OZ toJava. PhD thesis, University of Oldenburg, 2000

[FOS03] Howard Foster, Sebastian Uchitel. Model-based Verification of Web Service Compositions. Proceedings of the Eighteenth IEEE International Conference on Automated Software Engineering, pp. 152-163, 2003

[FUX04] Xiang Fu, Tevfik Bultan. Analysis of Interacting BPEL Web Services, thirteenth International World Wide Web Conference (WWW), pp. 621–630, May 2004

[GAN02] Sivakumar Ganesan, CSP: An Introduction. Seminar of Design and Validation of Computer Protocols, University of Technology Dresden, March 2002.

[GUN02] Hartwig Gunzer. Introduction to Web Services, Borland, March 2002

[HOA04] Charles Antony Richard Hoare. Communicating Sequential Processes, Prentice Hall, 1985.

[IBM03] IBM Corporation. "Business Process Execution Language for Web Services (BPEL4WS)," Version 1.1, May 2003

[INV01] Class invariant, online at: http://en.wikipedia.org/wiki/ Class_invariant; accessed on March 30, 2007

[JAI01] Roy Jaideep, Ramanujan Anupama. Understanding Web Services, IEEE IT Proessional, Volume 3, Issue 6, pp. 69–73, November/December 2001

[JAS01] JASS, online at: http://csd.informatik.uni-oldenburg.de/~jass/doc/grammar.html; accessed on March 30, 2007

[JCC01] JavaCC, online at: https://javacc.dev.java.net/doc/javaccgrm.html; accessed on March 30, 2007

[JJT01] JJTree, online at: https://javacc.dev.java.net/doc/JJDoc.html; accessed on March 30, 2007

[KOS03] Mariya Koshkina. Verification of Business Processes for Web Services. M.Sc. thesis, York University, 2003

[LEV01] Levenshtein distance, online at: http://en.wikipedia.org/wiki/Levenshtein_distance; accessed on March 30, 2007

[LIP78] Richard J. Lipton, Richard A. DeMillo. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer, Volume 11, Issue 4, pp. 34 – 41, April 1978.

[LIZ05] Zhongjie LI, Wei Sun. BPEL4WS Unit Testing: Framework and Implementation, Proceedings of the IEEE International Conference on Web Services, pp. 103 – 110, 2005

[LLP01] LL parser, online at: http://en.wikipedia.org/wiki/LL_parser; accessed on March 30, 2007

[MAY02] Yu-Seung Ma, Yong-Rae Kwon. Inter-Class Mutation Operators for Java, Thirteenth International Symposium on Software Reliability Engineering, pp. 352-363, 2002

[MAY06] Philip Mayer, Daniel Lübke. Towards a BPEL Unit Testing Framework, Proceedings of the 2006 workshop on Testing, Analysis, and Verification of Web Services and Applications, pp. 33 – 42, July 2006

[MUO02] Michael Moller. Specifying and Checking Java Using CSP, Workshop on Formal Techniques for Java-like Programs - FTfJP'2002

[NAK02] Shin Nakajima. Model-Checking Verification for Reliable Web Services, OOPSLA 2002 Workshop on Object-Oriented Web Services, pp. 1 -5, November 2002

[OFF02] Jeff Offutt. Quality Attributes of Web Software Applications, IEEE Software, Volume 19, Issue 2, pp. 25 – 32, March/April 2002

[OFF95] Jeff Offutt. A Practical System for Mutation Testing: Help for the Common Programmer.Twelfth International Conference on Testing Computer Software, pp. 99-109, June 1995.

[ORA01] A Hands-on Introduction to BPEL, online at: http://www.oracle.com/technology/pub/articles/matjaz_bpel1.html; accessed on March 30, 2007

[PEL03] Chris Peltz. Web Services Orchestration, a Review of Emerging Technologies, Tools, and Standards, Hewlett Packard Company, January 2003

[SOU00] Simone do Rocio Senger de Souza. Mutation Testing Applied to Estelle Specifications, pp. 8011 - 8020, Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8, 2000

[THA03] Do van Than. Web Service Orchestration, online at: http://www.eurescom.de/message/messageJun2003/Web_Service_Orchestration.asp; accessed on March 30, 2007

[TSA02] Wei-Tek Tsai, Ray Paul. Extending WSDL to Facilitate Web Services Testing, Proceedings of the Seventh IEEE International Symposium on High Assurance Systems Engineering, pp. 171- 172, 2002

[WOH02] Petia Wohed, Wil M. P.van der Aalst. Pattern Based Analysis of BPEL4WS, Technical Report FIT-TR-2002-04, Queensland University of Technology, April 2002.

[ZIJ06] Geert Zijlmans. A High-Level Petri Nets Based Modeling Language for Generation of BPEL4WS Processes, M.Sc. thesis, Eindhoven University of Technology, April 2006

Appendix A

This appendix gives the BNF of CSP_{BPEL}

```
<DEFAULT>
TOKEN:
{

    < TRACE_PARALLEL_OPERATOR: "||" >
  | < TRACE_CHOICE_OPERATOR: "[]" >
  | < TRACE_DEADLOCK_OPERATOR: "STOP" >
  | < TRACE_PREFIX_OPERATOR: "->" >
  | < TRACE_TERMINATION_OPERATOR: "TERM" >
  | < TRACE_CALL_OPERATOR:"CALL" >
  | < TRACE_IF_OPERATOR: "IF">
  | < TRACE_ELSE_OPERATOR: "ELSE">
  | < TRACE_WHILE_OPERATOR: "WHILE">
  | < BPELTRACE: "TRACE" >
}


/* SEPARATORS */

<DEFAULT>
TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < COMMA: "," >
| < DOT: "." >
}


/* OPERATORS */

<DEFAULT>
TOKEN :
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
```

```
| < BANG: "!" >
| < HOOK: "?" >
| < COLON: ":" >
| < PLUS: "+" >
| < TILDE: "~" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < SC_OR: "|" >
| < SC_AND: "&" >
| < XOR: "^" >
| < REM: "%" >
| < AT: "_">

}

TOKEN:
{
  <INT:"int">
| <CHAR: "char">
| <DOUBLE:"double">
| <BOOLEAN: "boolean">
| <NULL: "null">
}
/*****************************/

/* LITERALS */

TOKEN :
{
  < INTEGER_LITERAL:
       ["1"-"9"] (["0"-"9"])* (["l","L"])? // DECIMAL LITERAL
     | "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ (["l","L"])? //HEX LITERAL  >
|   < STRING_LITERAL:
     "\""
     (   (~["\"","\\","\n","\r"])
       | ("\\" ["n","t","b","r","f","\\","'","\""] )
     )*
     "\"" >
|<DURATION: "DU_"<STRING_LITERAL> >
|<DEADLINE:  "DE_"<STRING_LITERAL> >
}
```

```
TOKEN :
{
  < IDENTIFIER: (["a"-"z"] | ["A"-"Z"]) ( ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] )* >
| <EVENT_IDENT: <IDENTIFIER>"."<IDENTIFIER>>
| <TRACE_DEADLOACK_OPERATOR:  "BPEL_Dead">
| <BPEL_WAIT_NAME>     :  "BPEL_Wait_Name">
| <BPEL_WAITING_TIME> :  <DURATION>| <DEADLINE>
| <BPEL_MESSAGE_NAME:   "BPEL_Message_Name">
| <BPEL_FAULT_NAME    :   "BPEL_Fault_Name">
| <BPEL_VAR               :    <IDENTIFIER> >
| <BPEL_FAULT_VAR>     :   "BPEL_Falut_Var">
}

TraceAssertion          ::=     ( AssertionLabel )? TraceAssertionDeclaration
AssertionLabel          ::=     "[" <IDENTIFIER> "]"

TraceAssertionDeclaration    ::=     <BPELTRACE> <LPAREN> ( TraceConstant )*
( ProcessDeclaration )* <RPAREN>

TraceConstant           ::=     FieldDeclaration
FieldDeclaration        ::=     (Type) VariableDeclarator ( "," VariableDeclarator )* ";"
Type   ::=      ( PrimitiveType ) ( "[" "]" )*
PrimitiveType           ::=     <INT>| <CHAR> | <BOOLEAN>  | <DOUBLE>
VariableDeclarator      ::=     VariableDeclaratorId ( "=" VariableInitializer )?
VariableDeclaratorId    ::=     <IDENTIFIER> ( "[" "]" )*
VariableInitializer     ::=     ( ArrayInitializer | Expression )
ArrayInitializer        ::=     "{" ( VariableInitializer ( "," VariableInitializer )* )? ( "," )?
"}"

Expression     ::=      ConditionalExpression ( AssignmentOperator Expression )?
AssignmentOperator  ::=     ( "=" | "+=" | "-=" )
ConditionalExpression  ::=   ConditionalAndExpression ( "||"
ConditionalAndExpression )*

ConditionalAndExpression    ::=     EqualityExpression ( "&&" EqualityExpression )*
EqualityExpression    ::=     InstanceOfExpression ( ( "= =" | "!=" )
InstanceOfExpression )*
InstanceOfExpression          ::=     AdditiveExpression ( ( "<" | ">" | "<=" | ">=" )
AdditiveExpression )*

AdditiveExpression    ::=     MultiplicativeExpression ( ( "+" | "-" )
MultiplicativeExpression )*

MultiplicativeExpression      ::=     UnaryExpression ( ( "*" | "/" | "%" )
UnaryExpression )*
```

```
UnaryExpression      ::=     ( PreIncrementExpression | PreDecrementExpression |
PrimaryExpression )
PreIncrementExpression      ::=     "++" PrimaryExpression
PreDecrementExpression      ::=     "--" PrimaryExpression

PrimaryExpression :: = ( Literal  |  <IDENTIFIER> | "(" Expression ")")

Literal ::= <INTEGER_LITERAL> | <STRING_LITERAL> | <BOOLEAN> | <NULL>


ProcessDeclaration    ::=     ( ( ProcessDeclarator <LBRACE> ( FieldDeclaration )*
ProcessExpression <RBRACE> ) | ( FieldDeclaration )* ProcessExpression )

ProcessDeclarator     ::=     <IDENTIFIER> FormalParameters
FormalParameters      ::=     "(" ( FormalParameter( "," FormalParameter )* )? ")"
FormalParameter       ::=     Type( VariableDeclaratorId)
ProcessExpression     ::=     ProcessParallelExpression
ProcessParallelExpression     ::=     ProcessPrefixExpression (
<TRACE_PARALLEL_OPERATOR> ProcessPrefixExpression )*

 ProcessPrefixExpression      ::=     ( ProcessPrimaryExpression ) (
<TRACE_PREFIX_OPERATOR> ( ProcessPrimaryExpression ) )*

ProcessPrimaryExpression    ::=     ( <EVENT_IDENT> |BasicProcess |
ProcessIfElseExpression | ProcessWhileExpression | ( <LPAREN> ProcessExpression
<RPAREN> ) )

BasicProcess  ::=     ( <TRACE_DEADLOCK_OPERATOR> |
<TRACE_TERMINATION_OPERATOR> )

ProcessIfElseExpression      ::=     <TRACE_IF_OPERATOR> <LPAREN>
Expression <RPAREN> <LBRACE> ProcessExpression <RBRACE>
<TRACE_ELSE_OPERATOR> <LBRACE> ProcessExpression <RBRACE>

ProcessWhileExpression      ::=     <TRACE_WHILE_OPERATOR> <LPAREN>
Expression <RPAREN> <LBRACE> ProcessExpression <RBRACE>

Arguments     ::=     "(" ( ArgumentList )? ")"
ArgumentList  ::=     Expression ( "," Expression )
```

Appendix B

This appendix describes how we use JavaCC as a compiler-compiler for CSP_{BPEL}

After defining the BNF of CSP_{BPEL}, we need to build a compiler to check the syntax of the trace specification. It is not easy to build a complier from scratch, so we take advantage of the existed compiler-building tools to make it easy to build a compiler. An automatic compiler-building tool is called a compiler-compiler. With a compiler-compiler, we just need to create a grammar file, where we define the grammar specification of the trace specification language. Then the compiler-compiler can generate a compiler from the grammar file automatically. There are many compiler-compiler tools existing. Since the activeBPEL engine is built upon Java, so we choose a Java compiler-compiler to make things consistent. Currently there are several Java compiler-compiler tools available. After comparing these tools, we choose JavaCC [JCC01] as our java compiler-compiler. It is the most popular one those tools. It has the basic feature to generate a parser. In addition to that, it provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc [JJT01]. Furthermore, it allows us to embed actions in the grammar so that we can insert code to implement adding our monitoring code while parsing. Finally, it is a free tool and has been used by a large user group.

B.1 Introduction to JavaCC

JavaCC works with a grammar file, which is called a .jj file. The grammar of language is defined in the .jj file and then JavaCC converts that .jj file to a Java program, which is able to recognize matches to that grammar. The structure of grammar file is:

```
javacc_input ::= javacc_options
"PARSER_BEGIN" "(" <IDENTIFIER> ")"
java_compilation_unit
"PARSER_END" "(" <IDENTIFIER> ")"
( production )*
<EOF>
```

The grammar file starts with a list of options (which are optional). This is then followed by a Java compilation unit enclosed between "PARSER_BEGIN (name)" and "PARSER_END (name)" strings. After this is a list of grammar productions. We won't describe the options part here. Details regarding that can be found in [JCC01]. The java_compilation_unit could be any arbitrary Java compilation unit so long as it contains a class declaration whose name is the same as the name of the generated parser, which is the <IDENTIFIER> right after the "PARSER_BEGIN". In our instrumentation system, many features are implemented in this part, such as reading in the original BPEL4WS file into a DOM object, adding monitoring code for verifying variables and chronological information into the DOM object, and writing that DOM object back to the BPEL4WS file.

In JavaCC each rule of grammar is called a production. All the rules of grammar are defined in the production part. Production part has a structure like this,

```
production:: = javacode_production
 | regular_expr_production
 | bnf_production
 | token_manager_decls
```

Among these four production types, we just describe the BNF production here since it is most related to our work. The BNF production is the standard production used in specifying JavaCC grammars. Each BNF production has a left side which is a non-terminal specification. The BNF production then defines this non-terminal in terms of BNF expansions on the right side. The non-terminal is written exactly like a declared Java method. BNF expansions are composed of three layers. The first layer is expansion_choices. Expansion choices are written as a list of one or more expansions separated by "|"s. It's representation is

expansion_choices ::= expansion ("|" expansion)*

The second layer is expansion. An expansion is written as a sequence of expansion units, and its representation is

expansion ::= (expansion_unit)*

The third layer is expansion unit. Its representation is:

```
expansion_unit ::= local_lookahead
 | java_block
 | "(" expansion_choices ")" [ "+" | "*" | "?" ]
 | "[" expansion_choices "]"
 | [ java_assignment_lhs "=" ] regular_expression
 | [ java_assignment_lhs "=" ] java_identifier "(" java_expression_list ")"
```

An expansion unit has different representations. It can be a regular expression; but it can also be a java block. A java block is a set of Java declarations and code enclosed within braces (the Java block). This block is executed whenever the parsing process crosses this point successfully. It is the java block in an expansion unit where we embed actions to implement adding monitoring code. In a sample production:

```
(1) void activity():
(2) {Token actName;}
(3) {actName=<EVENT_IDENT>
(4)   {String activityName = actName.image;
```

```
(5)  System.out.println(" This is the activity name"+ activityName);
(6)  }}
```

Lines 4 and 5 are the expansion unit. They are used to retrieve the token name and print it out.

B.2. How to define a grammar in JavaCC

The grammar of our trace specification is defined in BNF notation, whilst JavaCC supports bnf_production. Comparing BNF and bnf_production, we can easily find that bnf_production is very similar to BNF. They both have a non-terminal specification on left side and regular expression on right side. The main difference between the bnf_production and standard BNF is that you can embed actions in the bnf_production. With the extensive similarity between bnf_production and BNF, it's fairly easy to translate from BNF to bnf_production. The major work what we need to do for the translation is writing each non-terminal as a declared Java method

For example, a BNF specification

```
ProcessParallelExpression::=  ProcessPrefixExpression (
            <TRACE_PARALLEL_OPERATOR> ProcessPrefixExpression )*
```

Can be translated into bnf_production:

```
void ProcessParallelExpression():
{}
{(ProcessChoiceExpression() ( <TRACE_PARALLEL_OPERATOR>
ProcessChoiceExpression() )*) }
```

Appendix C

This appendix describes the implementation of instrumentation system

C.1 Choose BPEL4WS engine

A BPEL4WS program has to run upon a BPEL4WS engine, so that we need to choose a BPEL4WS engine to support the running of test BPEL4WS programs. BPEL4WS engines can be categorized to free and commercial. Since we only use it for research purpose, we decide to choose a free engine. Currently, there are two popular free BPEL4WS engines, active-BPEL [ACT01] and bpws4j [BPW01]. Actvie-BPEL is provided by Active Endpoints, Inc. BPWS4J is provided by IBM.

After comparing these two engines from several aspects, we decide to choose active-BPEL in our research. The reasons why we choose active-BPEL are

- It is free
- It is open source
- It has regularly updated releases
- It has a big user group
- It is easy to extend with custom features

C.2 How monitoring code passes information to the trace verification system

Before we describe how to design the monitoring code to monitor the variable values and chronological information separately, we illustrate how that information is passed to trace verification system.

Original approach

In the standard BPEL4WS language, the only way to send the internal state information of a running BPEL4WS program out is to invoke an external web service and pass the internal state information as input parameters to that web service.

At first, our approach was to send information to a monitoring web service. The monitoring code in BPEL4WS program invokes that monitoring Web service, and sends variable information and chronological information as input parameters to the operations in the monitoring Web service. After we implemented this approach, we found this approach has an obvious disadvantage. It is slow. Invoking the monitoring Web service once may not take much time. However, the total amount of time spent on invoking the monitoring Web service for all the monitored activity events and variables is much bigger, and it heavily affects the performance of the BPEL4WS program.

New approach

After we realized that building a monitoring web service is not the best way to collect test information, we turned to the activeBPEL engine itself. We hoped it could provide some mechanisms to transmit internal state information. After analyzing the activeBPEL engine thoroughly, we found that it providse a mechanism to add custom functionality to the engine. Adding custom functions to activeBPEL engine implies creating some custom functions with Java, and changing the configuration file of the engine to inform it of the existence of these custom functionalities. Then the activeBPEL engine can treat these additional custom functions as a library of methods, and the monitoring code can call those custom functions from within the BPEL4WS program. Therefore, with this mechanism, we can implement the trace verification system as a custom function, and then the monitoring code in the BPEL4WS program passes variable and chronological information as input parameters to that custom function.

Procedure to add a custom function

The procedure to add a custom function has the following two steps.

- Write Java code to implement the specific feature. The Java code should follow a specific format required by the activeBPEL. The details of the format can be found in the sample code of how to write custom function. The sample code can be downloaded from [ACT01]
- Edit activeBPEL's configuration file aeEngineConfig.xml to add information about the custom functions. The default element defining the custom function in the configuration file is "<entry name="FunctionContexts"/>", We can change it to

```
<entry name="FunctionContexts">
<entry name="Custom XML String Functions">
 <entry name="Prefix" value="xmlstring"/>
 <entry name="Class" value="custom_function_class"/>
</entry>
</entry>
```

The name "Custom XML string Functions" doesn't have any special meaning. It just indicates that it's the main feature of the custom function. A call to the custom function is composed of two parts. The first part is the prefix, and the second part is the class name of custom function.

How to call custom function in BPEL4WS program

We have to call a custom function within a <Copy> element of an <Assign> activity. A snippet of sample BPEL4WS code to call a custom function is

```
<assign name="Test_ElementToXMLString_CustomFunction">
<copy> <from expression="xmlstring:elementToXMLString(
          bpws:getVariableData('brandYearTypeMessage', 'brandYearType') )" />
 <to variable="myString"/> </copy> </assign>
```

In this testing code, the element <from> invokes the custom function. The custom function is "elementToXMLString". It has one input parameter, part 'brandYearType' of variable 'brandYearTypeMessage'. The custom function returns a string value, and it is assigned to the variable "myString". In some cases, there is no value returned from the custom function, so that in order to make the Copy Operation valid in the grammar, we add a non-meaningful string as the return value of custom function and then assign that value to a dummy variable.

C.3 Monitoring code passing variable information

Before we pass the variable values to the trace verification system, we need to retrieve those values. In a BPEL4WS program, it is easy to retrieve the value of a variable. The grammar to do that is

bpws:getVariableData('variable name', 'part name')

After retrieving the values of the variables, we pass those values to the trace verification system through invoking a custom function, BPEL: sendVariable (name, value). This custom function is a part of the trace verification system. It not only receives variable values but also determines the expected trace behavior upon the variable values. This function has two input parameters, the name of variable and the value of variable. A sample monitoring code to call this custom function is:

```
<assign name="sendVarialbeValue">
<copy><from expression="BPEL:sendVariable(
          "order.steps",bpws:getVariableData('order','steps'))" />
        <to part="activityName" variable="bpelTesting" />
</copy>
</assign>
```

The monitoring code used to pass the variable values is simple. As to where we should instrument the monitoring code, it is not difficult either. Although the value of a variable can change during the running of a BPEL4WS program, what we want to monitor is the value that is used to determine the result of a conditional expression. Therefore we can insert the code immediately before the activity that includes the conditional expression. The location of that activity can be calculated from those activities immediately after the conditional expression in a trace specification.

C.4 Monitoring code passing chronological information of activities

Major approach

To determine the chronological order in which the activities are executed, our approach is to record the system time when activities start and end. The system time is the system time of computer where the BPEL4WS program runs. An activity may be invoking a remote Web service, and it is not easy to know the real time when the remote Web service starts and ends. For the sake of simplicity, we use two other times to represent the actual start and end time.

We use the time when the BPEL4WS process invokes a remote Web service to represent the time when the Web service really starts. In order to get the system time when invoking a remote web service, we instrument an <assign> activity immediately before an <invoke> activity. The assign activity is used to call a custom function to get the system time. In order to ensure that those two activities are executed one after another. We put them within a <sequence> activity.

We use the time when the BPEL4WS process receives the response from a remote Web service as the time when the Web service really ends. Unfortunately, BPEL4WS doesn't provide any facility to record the exact time when the BPEL4WS process receives the response from the remote Web service. Therefore, we can only use a time that most closely approximates that time. This is achieved by adding an <assign> activity immediately after the <invoke> activity. The <assign> activity is used to call a custom function to get the system time, and it is put within the sequence activity where the activity invoking the Web service resides.

What is monitoring code

For each monitored activity, we instrument two pieces of the code for testing. One is instrumented before the monitored activity, and the other one is instrumented after it. Each piece of monitoring code includes four < assign> activities. List C.1 is a snippet of sample monitoring code added before the monitored activity. The first <assign> activity is to call the custom function systemTime () in order to get the computer system's time. This time is assigned to the "startTime" part of the variable "bpelTesting". The second <assign> activity is to assign a constant value "None" to the 'endTime' part of the variable "bpelTesting". The constant value "None" is used as a tag to indicate that the variable "bpelTesting" is only to specify the start time of the activity. The third <assign> activity is to assign the name of activity to the "activityName" part of the variable "bpelTesting". The fourth <assign> activity is to call the custom function "bpelTrace". This custom function is our trace verification system. When invoking the custom function

"bpelTrace", there are four input parameters. Besides the "startTime", "endTime", and "activityName", there is a parameter called "ProcessTime". "ProcessTime" is a unique identifier for each distinct instance of a BPEL4WS process. With this identifier, we can separate one instance from the other one.

```
<assign name="reportStartTime">
 <copy>from expression="Time:systemTime()" />
        <to part="startTime" variable="bpelTesting" />
 </copy> </assign>
 <assign name="reportStartTime2">
 <copy><from expression="'None'" />
         <to part="endTime" variable="bpelTesting" />
 </copy>  </assign>
 <assign name="reportStartTime3">
 <copy><from expression="'invokeapprover'" />
        <to part="activityName" variable="bpelTesting" />
 </copy>  </assign>
 <assign name="reportStartTime4">
 <copy>
        <from expression="BPEL:bpelTrace( bpws:getVariableData('bpelTesting',
        'processTime'),bpws:getVariableData('bpelTesting',
        'activityName'),bpws:getVariableData('bpelTesting',
        'startTime'),bpws:getVariableData('bpelTesting', 'endTime') )" />
        <to part="activityName" variable="bpelTesting" />
 </copy>  </assign>
```

List C.1 Sample monitoring code added before an activity

These four <assign> activities are almost the same for all the activities to be monitored. The only difference is the value of expression in the third assign activity, since it is the name of the monitored activity. The monitoring code added after the monitored activity is quite similar to the monitoring code before the activity. The only difference between them is that in the monitoring code before, the "startTime" part is system time and the "endTime" part has value "None". While in the monitoring code after, the "startTime" Part has value "None" and the "endTime" part is system time.

Appendix D

This appendix describes the implementation of trace verification system

D.1 Introduction to parse tree

As we stated in appendix B, we use JavaCC to generate a parser to check the syntax of the trace specification and automatically instrument the monitoring code. However, the parser generated by JavaCC lacks the ability to represent the trace specification using an appropriate data structure, which is suitable for later processing and captures the internal hierarchy of the input. The most widely used data structure in parsing is tree structure, called parse tree. Many popular computer programming languages use parse tree to parse their input strings. Fortunately, JavaCC has a complementary component called JJTree [JJT01]. JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various locations in the JavaCC source. The output of JJTree is run through JavaCC to create a parser that can parse input with a parse tree.

In a parse tree, all the leaf nodes of the tree are labeled by terminals of the grammar. The root node of the tree is labeled by the start symbol of the grammar, and the interior nodes are labeled by the nonterminals. If an interior node has a label A, and it has n child nodes with labels X1, X2, ..., Xn from left to right, then the production rule A → X1 X2 .. Xn must exist in the grammar. For example, consider a grammar whose list of productions is

E::=E+E
E::=E*E
E::=id

The tree shown in Figure D.1 is a parse tree for a string id + id * id.

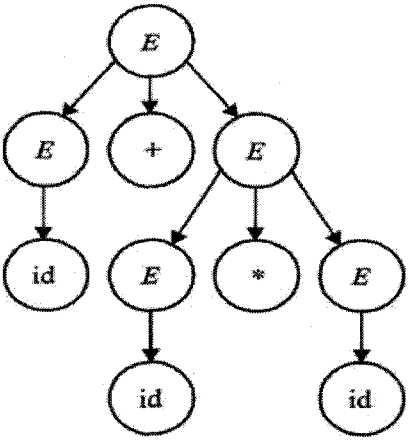


Figure D.1 Parse tree for string id+id*id

D.2 Use parse tree to determine the expected trace

The purpose of the parse tree is to determine the expected execution order of all the monitored activities. An event token in the BNF of CSP_{BPEL} is a terminal, hence in a parse tree all the nodes for events are leaf nodes. Among all those event nodes, we can find out which event or events should execute first based on the relationship among them. There are two types of basic relationships between any two events, sequence and parallel. Any other relationship can be represented as a composition of these two basic relationships.

How to determine the expected trace between two activities

If two activities have a sequential relationship, then it means that one activity should finish before the other activity starts. The formula to represent this relationship is:

$$activityA.s \rightarrow activityA._E \rightarrow activityB.s \rightarrow activityB._E$$

If two activities have a parallel relationship, then it means that any activity can start first as long as it starts before the ending of the other activity. The formulas representing this relationship are:

activityA.s->activityB.$_S$ -> activityB.$_E$->activityA.$_E$. OR

activityB.s->activityA.$_S$ -> activityB.$_E$->activityA.$_E$. OR

activityA.s->activityB.$_S$ -> activityA.$_E$->activityB.$_E$. OR

activityB.s->activityA.$_S$ -> activityA.$_E$->activityB.$_E$. OR

The relationship between two activities can be easily represented in a parse tree. If two activities, activityA and activityB, have a sequential relationship, their trace specification is activityA->activityB. When this trace specification is parsed into a parse tree, the parse tree is:

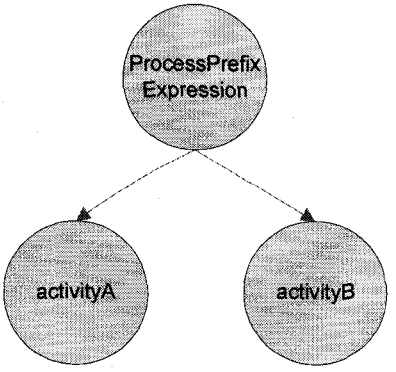


Figure D.2 Parse tree for two sequential activities

In a parse tree, we can programmatically determine the relationship between two activities. Since nodes activityA and activityB both are the child nodes of node ProcessPrefixExprresion, we can deduce that the child nodes of node ProcessPrefixExpresion have a sequential relationship. After determining that two activities have a sequential relationship, we have to know which one should execute first. JJTree is a LL parse tree. A LL parser is a top-down parser for a subset of context-free grammars. It parses the input from left to right, and constructs a leftmost derivation of the

sentence. [LLP01]. A leftmost derivation means replacing the left-most non-terminal during the process of parse. Since JJTree parses the input from left to right and in that trace specification token activityA is on the left of token activityB, therefore, token activityA is first parsed into the parse tree between itself and token activityB. Node ProcessPrefixExpression only has two child nodes, activityA and activityB. Since node activityA is parsed first, therefore its index as a child node is 0, and the index of acitivityB is 1. As we know that activityA should execute before acitivityB, hence we can deduce a rule that among the child nodes of a ProcessPrefixExpression node, the node with the smaller index will execute before the node with bigger index. There is a JJTree API method Node.getChild (index) to retrieve a child node by index, so that we can programmatically find out what is the first activity.

Analogously, if two activities, activityA and activityB, have a parallel relationship, their trace specification is activityA|| activityB. When this trace specification is parsed, its corresponding parse tree is:

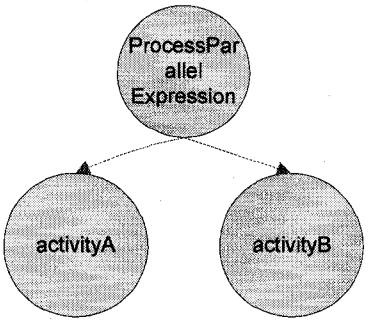


Figure D.3 Parse tree for two parallel activities

Since nodes activityA and activityB both are the child nodes of node ProcessParallelExprresion, we can deduce that the child nodes of node ProcessParallelExprresion have a parallel relationship. For two activities in a parallel, we don't have to determine which one can execute first since anyone can execute first.

How to determine the expected trace of a whole process

Normally, a BPEL4WS process includes more than two activities. In the previous section, we described how the two basic relationships of two activities are represented in a parse tree. Since the relationships between any two activities in a whole process are still those two basic relationships, therefore the parse tree of a whole process can be built based on the parse tree of two activities. For example, a process has four activities, and its trace specification is activityA-> (activityB | |activityC) ->activityD. In this trace specification, since activityB and activityC are included in a pair of brackets, they are treated as a unit when they are parsed. They have a parallel relationship, so in the parse tree they are child nodes of node ProcessParallelExpression. As a unit, they have a sequence relationship with activityA and activityD. Therefore in the parse tree, their parent node ProcessParallelExpression and nodes activityA and activiytD, are child nodes of node ProcessPrefixExpression. The parse tree for this trace specification is:

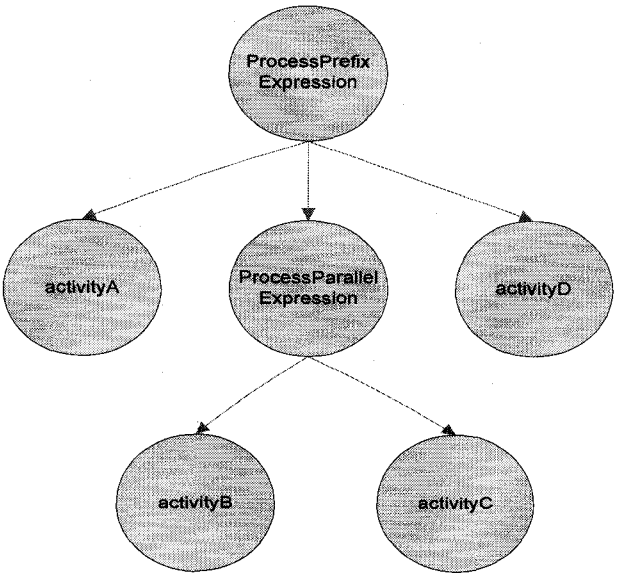


Figure D.4 Parse tree for activities with complex relationship

The approach to programmatically deduce the execution order of activities includes two major steps. Step 1 is to determine which activity should execute first (Figure D.5). Step 2 is to determine the expected trace of activities thereafter (Figure D.6).

In step 1, we declare a variable "currentNode". This variable indicates currently which activity is expected. It is used all through the implementation of the approach. From figure D.5 we can find that the activity, which should execute first, might not be deterministic in some situations. If there are a group of activities having a parallel relationship, then any activity in the group can execute first. It means we can not determine exactly which one should execute first, but determine that any activity in the group executing first is acceptable. For instance, for trace specification (activityA||activityB) -> activityC, activityA or activity B can execute first. To tackle this

issue, we define a collection and put all the activities that execute parallel into that collection, and then assign this collection to variable "currentNode."

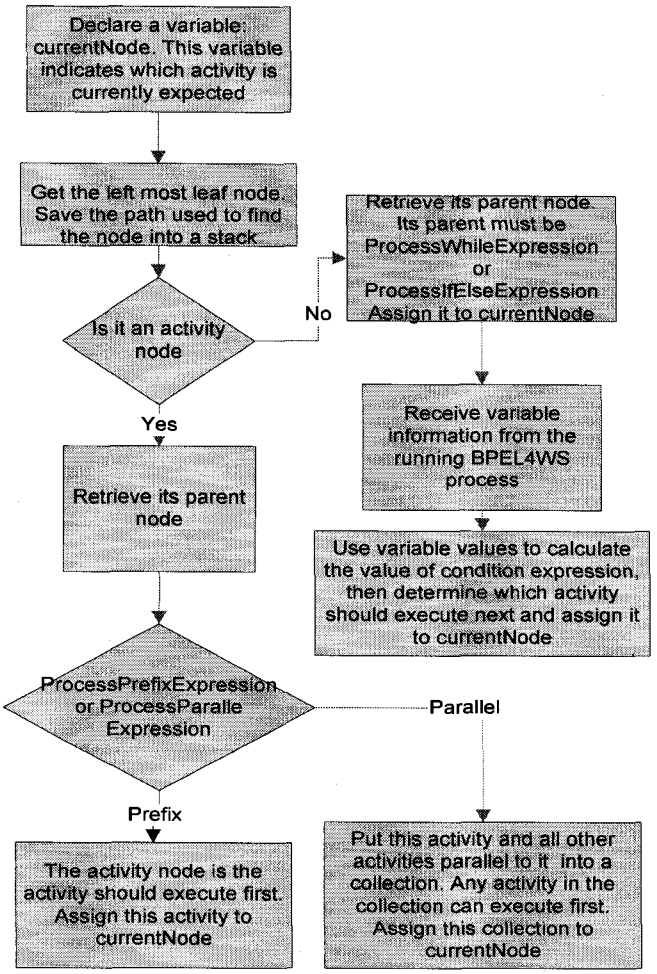


Figure D.5 Process to find the first expected activity

In step 2, we don't calculate which activity should execute next until we receive the current expected activity event from the running BPEL4WS program. An activity event is the chronologic information sent to the trace verification system . After receiving the chronological information, step 2 verifies if the activity event is correct. How to verify will be discussed in details in next section. After the verification action, we make use of a

stack structure to find out which is the next expected activity. The stack structure is created in step 1. When we look for the first expected activity, we push the name and index of all its parent nodes into the stack. The index of a node is its index among itself and all its brother nodes. When we determine the next expected activity, we pop up the parent node of current node and then use the index of current node to find out if the next activity is in its brother nodes or if we need to pop up a higher level node to find the next expected activity. The detailed process to find the next activity using a stack structure is shown in figure D.7.

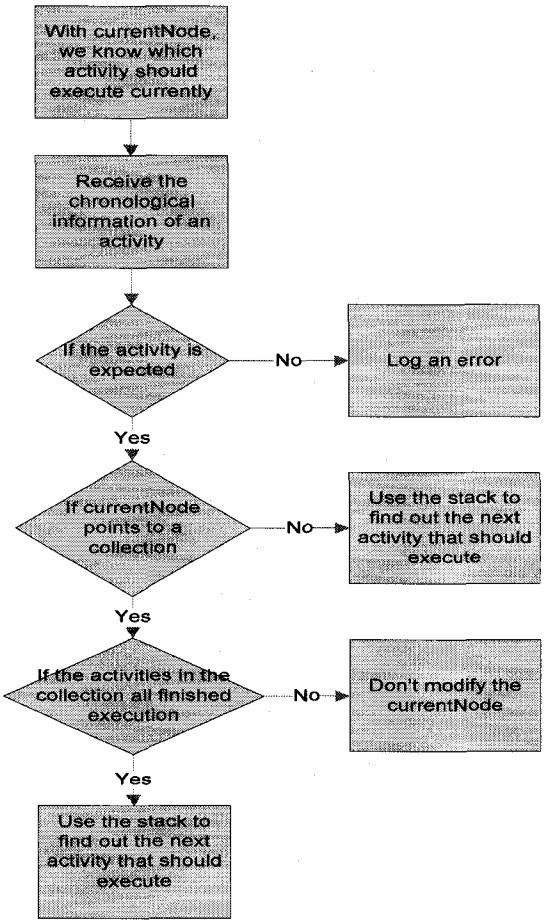


Figure D.6 Process to find the next expected activity

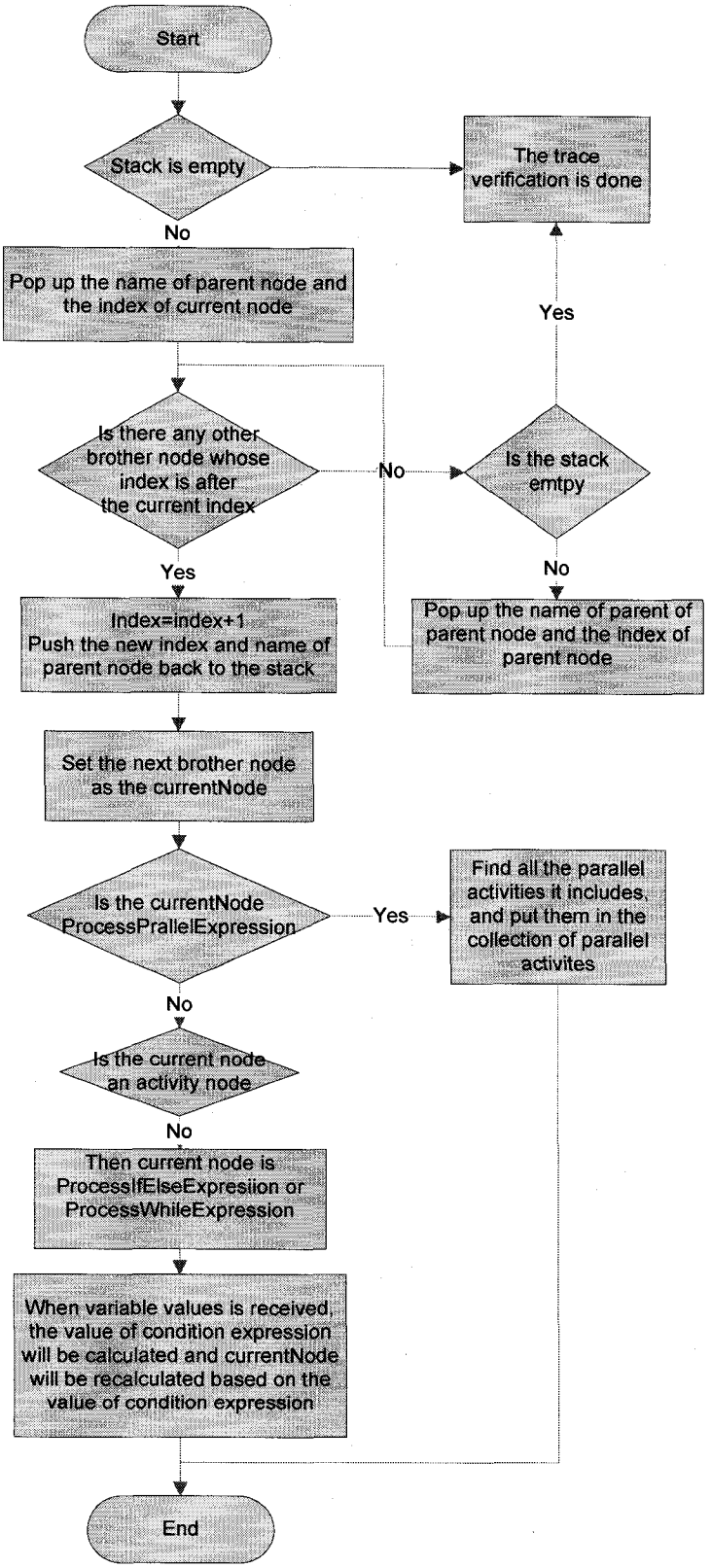


Figure D.7 How to use stack structure to find next expected activity

D.3 Verify the actual trace

In appendix C, we illustrated what is monitoring code and how the instrumentation system inserts monitoring code into appropriate locations in a BPEL4WS programs. During the execution of the BPEL4WS program, that monitoring code passes chronological information and variable values to the trace verification system. After receiving the actual trace information, the trace verification system verifies it upon the expected activity, which can be deduced by the approach described in the previous section. How to verify the actual trace is shown in figure D.8.

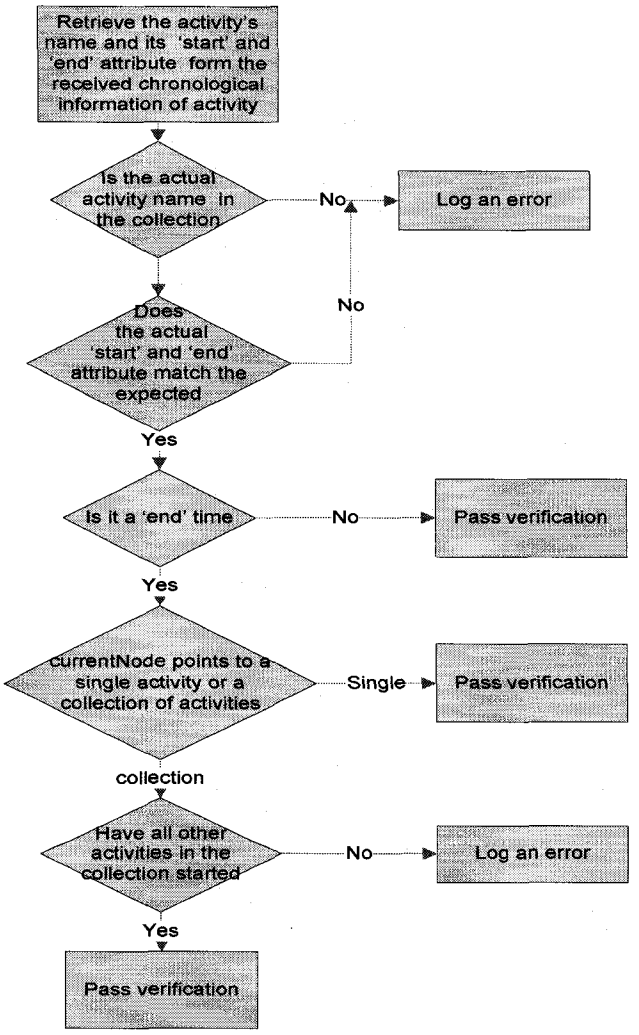


Figure D.8 How to verify the actual trace

When verifying the actual activity trace, we not only check if the actual activity is what we expect but also check if its start time and end time are correct. Since the sequence or parallel relationship between two activities can be represented as a relationship between their start and end time, as we describe in appendix C.4. Therefore, by verifying their real start time and end time, we can validate if the relationship between two activities is satisfied.

There is a special case where no chronological information is passed to the trace verification system due to the exception or deadlock in the process. When there is an exception in the process, the normal execution of the process is stopped and exception handling is performed. In the exception handler, as no monitoring code is instrumented, therefore no chronological information is passed. Analogously, when the process is in a deadlock state, no activity in the process is able to execute. Therefore, no activity can pass chronological information to the trace verification system either. When there is no chronological information is passed, the trace verification waits for a period of time then throws out a time out error. The length of the waiting time can be specified through configuration.

Appendix E

This appendix gives the details of mutants used in the experiments.

(Legend: "//" implies that we comment the code on its right side. Code without "//" on the left side is the code added)

No.	Mutants Description	Location (in file purchase.bpel)	Fault Model
1	//<sequence> <flow>	Line 189	Sequential to Parallel(FM1)
	//</sequence> </flow>	Line 194	

2	//<invoke partnerLink="scheduling" name ="InitiateProductionScheduling"
	// portType="sch:scheduling" operation="requestProductionScheduling"
	// inputVariable="PO" outputVariable="productionSchedule"/>
	
	<invoke partnerLink="scheduling" name ="CompleteProductionScheduling"
	portType="sch:scheduling" operation="sendShippingSchedule"
	inputVariable="shippingSchedule" outputVariable ="finalSchedule">
	<target linkName="ship-to-scheduling"/>
		Line 190	Switch Activity(FM3)

	//<invoke partnerLink="scheduling" name ="CompleteProductionScheduling"
	// portType="sch:scheduling" operation="sendShippingSchedule"
	// inputVariable="shippingSchedule" outputVariable ="finalSchedule">
	// <target linkName="ship-to-scheduling"/>
	
	<invoke partnerLink="scheduling" name ="InitiateProductionScheduling"
	portType="sch:scheduling" operation="requestProductionScheduling"
	inputVariable="PO" outputVariable="productionSchedule"/>
		Line 191	

3	//</links> <link name="shipper1First"/></links>	Line 108	Parallel to Sequential(FM5)
	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1"/><source linkName="shipper1First"/>	Line 119	
	//<source linkName="quoteShipper2"/> <source linkName="quoteShipper2"/><target linkName="shipper1First"/>	Line 122	
4	//</invoke> </invoke><terminate/>	Line 123	Unexpected Terminate (FM2)
5	//</links> <link name="start-scheduling"/></links>	Line 108	Deadlock (FM4)
	//</invoke> <target linkName="start-scheduling"/></invoke>	Line 191	
	//</invoke> <source linkName="start-scheduling"/></invoke>	Line 194	
6	//<source linkName="quoteShipper2"/> <target linkName="quoteShipper1"/> <source linkName="quoteShipper2"/>	Line 122	Synchronization to Sequence (FM6)
	//<target linkName="quoteShipper1"/>	Line 125	
	//joinCondition="quoteShipper1 AND quoteShipper2"	Line 127	
7	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1" transitionCondition= "bpws:getVariableData('shippingInfo1', 'price')<200"/>	Line 119	Extra Single Condition (FM8)

8	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1" transitionCondition= "bpws:getVariableData('shippingInfo1', 'price')<200"/>	Line 119	Extra And Conditions (FM9)
	//<source linkName="quoteShipper2"/> <source linkName="quoteShipper2" transitionCondition= "bpws:getVariableData('shippingInfo2', 'price')<200"/>	Line 122	
9	//<source linkName="quoteShipper1"/> <source linkName="quoteShipper1" transitionCondition= "bpws:getVariableData('shippingInfo1', 'price')<200"/>	Line 119	Extra OR Conditions (FM10)
	//<source linkName="quoteShipper2"/> <source linkName="quoteShipper2" transitionCondition= "bpws:getVariableData('shippingInfo2', 'price')<200"/>	Line 122	

	// joinCondition="quoteShipper1 AND quoteShipper2"
	joinCondition="quoteShipper1 OR quoteShipper2"
		Line 127	

10	// <case condition="bpws:getVariableData('shippingInfo1','price') <= // bpws:getVariableData('shippingInfo2','price')> <case condition="bpws:getVariableData('shippingInfo1','price') > bpws:getVariableData('shippingInfo2','price')>	Line 129	Switch Condition (FM11)

11	//transitionCondition="bpws:getVariableData('PO',"needElectronicInvoice") = 'yes'"
	transitionCondition=" bpws:getVariableData('PO','needElectronicInvoice') = 'yes'"
	AND bpws:getVariableData('PO','needPaperInvoice') = 'no'"
		Line 206	Multi-Choice to Simple Merge (FM14)

12	//transitionCondition="bpws:getVariableData('PO',"needPaperInvoice")= 'Yes'"	Line 205	Multi-Choice to Parallel (FM15)

	//transitionCondition="bpws:getVariableData('PO',"needElectronicInvoice") = 'Yes'"	Line 206	

13	// joinCondition="sendPaper-invoice OR sendEmail-invoice"
	joinCondition="sendPaper-invoice AND sendEmail-invoice"
		Line 217	Synchronizing Merge to Synchronization with AND conditions(FM16)

14	// <from expression="bpws:getVariableData('po','price') + // bpws:getVariableData('shippingInfo','price')"/> <from expression="bpws:getVariableData('po','price') - bpws:getVariableData('shippingInfo','price')"/>	Line 161	Mathematics operator exchanged

15	// outputVariable="shippingInfo2"
	outputVariable="shippingInfo1"
		Line 121	Variable by Variable replacement

16	// <from expression="bpws:getVariableData('po','price') + // bpws:getVariableData('shippingInfo','price')"/> <from expression="bpws:getVariableData('po','price') + bpws:getVariableData('shippingInfo','price')+ 100"/>	Line 161	Increment Variables
17	// <from expression="bpws:getVariableData('po','price') + // bpws:getVariableData('shippingInfo','price')"/> <from expression="bpws:getVariableData('po','price') + bpws:getVariableData('shippingInfo','price')- 100"/>	Line 161	Decrement Variables
18	//<case condition="bpws:getVariableData('totalCharge','number') <= //bpws:getVariableData('credit','number')> <case condition="bpws:getVariableData('totalCharge','number') <= 100	Line 168	Variable by constant replacement

Table E.1 Mutants of Experiment 1

No.	Mutants Description	Location (in file insurance.bpel)	Fault Model
19	//<source linkName="policeReport"/>	Line 117	Sequential to Parallel(FM1)
	//<target linkName="policeReport"/>	Line 124	
20	//</invoke> </invoke><terminate/>	Line 118	Unexpected Terminate (FM2)
21	//</links> <link name="policeReport"/> </links>	Line 115	Deadlock (FM4)
	//</invoke> <target linkName="extra-link"/></invoke>	Line 118	
	//</assign> <source linkName="extra-link"/></assign>	Line 125	
22	//</flow>	Line 105	Synchronization to Parallel (FM7)
	//</assign> </assign></flow>	Line 111	
23	//<flow> <sequence>	Line 100	Synchronization to Sequence (FM6)
	//</flow> </sequence>	Line 105	

24	//transitionCondition="bpws:getVariableData('claim','directDeposit")= 'yes'
	transitionCondition="bpws:getVariableData('claim','directDeposit")!= 'yes'
		Line 149	Switch Condition (FM11)

	//transitionCondition="bpws:getVariableData('claim','directDeposit")=! 'yes'
	transitionCondition="bpws:getVariableData('claim','directDeposit")= 'yes'
		Line 150	

25	//transitionCondition="bpws:getVariableData('claim','directDeposit")= 'yes'	Line 149	Exclusive Choice to Parallel (FM12)
	//transitionCondition="bpws:getVariableData('claim','directDeposit")!= 'yes'	Line 150	

26	// trnaiditionCondition="bpws:getVariableData('claim','totalExpense')<1000 AND
	// bpws:getVariableData('claim','totalExpense')>200"/>
	bpws:getVariableData('policeReport','liability')!='Full'
		Line 134	Simple Merge to Multi-Choice (FM13)

27	// <onAlarm for="'PT02M'"> // <throw faultName="lns:Timeout" faultVariable="Fault" /> // </onAlarm>	Line 83	Missing Alarm(FM17)
28	//<catch faultName="lns:InfoNotAvaliable" faultVariable="Fault"> <catch faultName="lns:Timeout" faultVariable="Fault">	Line 59	Incorrect Fault Matching(FM18)
29	//<from expression="bpws:getVariableData('medExpense','count')+ //bpws:getVariableData('repairExpense','count')"/> <from expression="bpws:getVariableData('medExpense','count') - bpws:getVariableData('repairExpense','count')"/>	Line 108	Mathematics operator exchanged
30	//<from expression="bpws:getVariableData('medExpense','count')+ //bpws:getVariableData('repairExpense','count')"/> <from expression="bpws:getVariableData('repairExpense','count') + bpws:getVariableData('repairExpense','count')"/>	Line 108	Variable by Variable replacement
31	//<source linkName="majorAccident" //transitionCondition="bpws:getVariableData('claim','totalExpense')>1000"/> <source linkName="majorAccident" transitionCondition="bpws:getVariableData('claim','totalExpense')>10000"/>	Line 133	Increment constants

32	//<source linkName="majorAccident" //transitionCondition="bpws:getVariableData('claim','totalExpense')>1000"/> <source linkName="majorAccident" transitionCondition="bpws:getVariableData('claim','totalExpense')>100"/>	Line 133	Decrement constants
33	//<from expression="bpws:getVariableData('medExpense','count')+ //bpws:getVariableData('repairExpense','count')"/> <from expression="bpws:getVariableData('medExpense','count') +500"/>	Line 108	Variable by constant replacement

Table E.2 Muntants of Experiment 2