

VERIFICATION OF BPEL SPECIFICATIONS USING MODEL CHECKING

by

Mehmet N. Akçay

B.S. in Computer Engineering, Istanbul Technical University, 2005

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Bogaziçi University

2008

ACKNOWLEDGEMENTS

First of all, I would like to thank to my advisor Professor Mehmet Ufuk Çağlayan, for his patience, support, and encouragement throughout my graduate study. Without his support, this thesis would never exist.

I want to specially thank to A. Burak Gürdağ who has always supported me to complete this work and has never given up on me. I would also want to thank him for his crucial support on PROMELA models and enlightening me always.

I want to thank to Evren Önem for his guidance and showing me the way out of the model checking methodologies and for his technical consulting.

And I also want to specially thank to Gürkan Gür who has always trusted in me and supported me writing this thesis and helped whenever I needed help.

ABSTRACT

VERIFICATION OF BPEL SPECIFICATIONS USING MODEL CHECKING

Service Oriented Architecture(SOA) is based on creating services which can be distributed on a network by different business flows. Business Process Execution Language (BPEL) is a recent specification language to express web service applications and business flows in an easier way in SOA implementations. In order to verify the correctness of BPEL processes during design, a number of software verification methods can be applied to BPEL specified processes. In this paper, the translation of BPEL process specifications into PROMELA specification language and verification by model checking by SPIN tool are studied. By creating a model of any BPEL process, several verification steps can be applied during design time by the help of SPIN tool. A subset of BPEL activities consisting of assign, switch, sequence, empty, while, terminate, scope, flow, throw, invoke, reply, receive, pick, wait, fault handlers and compensation handlers are modeled to check and verify the processes specified by using these activities. With the tool created in this work, a given set of inputs consisting of BPEL source code(s), wsdl files(s) and BPEL descriptor file(s) are used to create a PROMELA model, then the model is verified by using the SPIN model checking tool for counter claims, assertions and unreachable codes.

ÖZET

BPEL TANIMLAMALARININ MODEL KONTROL YÖNTEMİ İLE DOĞRULANMASI

Servis Odaklı Mimari (SOA) farklı iş akışları tarafından oluşturulabilen ve network üzerinden dağıtılabilir servislerin yaratılması ile oluşturulmaktadır. SOA Mimariisi ile oluşturulması hedeflenen iş akışları için ise bir çok dil bulunmaktadır. BPEL yakın zamanda ortaya çıkan bir yazılım dili olmakla birlikte web servis mimarileri ve iş akışlarını için kolay ve uygun bir dil özelliği taşımaktadır. Tasarım aşamasında BPEL tanımlamalarının doğrulanması için, BPEL ile yazılmış prosedürlere yazılım doğrulama yöntemleri uygulanabilir. Bu çalışma ile BPEL prosedürlerinin PROMELA prosedürlerine dönüştürülmesi ve bu dönüştürülmüş PROMELA prosedürlerinin SPIN aracının yardımı ile doğrulanması gerçekleştirilmiştir. Bu çalışmada assign, switch, sequence, empty, while, terminate, scope, flow, throw, invoke, reply, receive, pick, wait ve fault-handler BPEL aktiviteleri baz alınarak bir PROMELA modeli oluşturulmaktadır. Bu dönüşümü gerçekleştirmek için BPEL2PML diye isimlendirilen bir araç oluşturulup, oluşan PROMELA tanımlamaları üzerinde doğrulama işlemleri yapılabilmektedir. BPEL prosedürlerinin PROMELA modellerinin oluşturulmasının ardından, SPIN ile doğrulama yapılarak, negatif örnekler, onaylama bildirileri ve hiçbir zaman ulaşılamayan kod parçacıkları tespit edilebilmektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	ix
LIST OF TABLES	xiv
LIST OF SYMBOLS/ABBREVIATIONS	xv
1. INTRODUCTION	1
2. OVERVIEW OF SOA AND BPEL	5
2.1. SOA Concept	5
2.2. BPEL	6
2.2.1. Main structure	6
2.2.2. Partnerlinks	6
2.2.3. Variables	8
2.2.4. FaultHandlers	8
2.2.5. Event Handlers	8
2.2.6. Basic Activities	8
2.2.6.1. Receive	8
2.2.6.2. Reply	9
2.2.6.3. Invoke	9
2.2.6.4. Assign	10
2.2.6.5. Wait	10
2.2.6.6. Empty	11
2.2.6.7. Sequence	11
2.2.6.8. Scope	12
2.2.7. Control Activities	12
2.2.7.1. Switch	12
2.2.7.2. While	14
2.2.7.3. Pick	14
2.2.7.4. Flow	15

2.2.7.5. Terminate	16
2.2.7.6. Throw	16
2.2.7.7. Catch	16
2.2.7.8. Compensate	17
3. OVERVIEW OF MODEL CHECKING FOR SOFTWARE VERIFICATION	19
3.1. MODEL CHECKING	19
3.2. Linear Temporal Logic (LTL)	21
3.3. SPIN	22
4. MODEL CHECKING OF BPEL SPECIFICATIONS USING SPIN	24
4.1. Main Objective	24
4.2. Web Service, WSDL and XSD definitions	28
4.3. Modeling WSDL and XSD files in PROMELA	30
4.4. Modeling Expressions and Conditions	33
4.4.1. Lex Model	33
4.4.2. CUP Model	37
4.5. Modeling Activities in PROMELA	41
4.5.1. Assign	41
4.5.2. Sequence	42
4.5.3. Switch	43
4.5.4. While	45
4.5.5. Scope	46
4.5.6. Terminate	49
4.5.7. Flow	50
4.5.8. Throw	51
4.5.9. Pick	53
4.5.10. Empty	55
4.5.11. Wait	55
4.5.12. Invoke	56
4.5.13. Receive	58
4.5.14. Reply	60
4.5.15. Compensate	61
4.6. Modeling Fault Handlers In PROMELA	62

4.7. Modeling Compensation Handlers In PROMELA	64
4.8. Verification Process	65
4.9. Example Scenario	68
5. CONCLUSIONS	75
6. FUTURE WORK	77
APPENDIX A: BPEL Source	78
APPENDIX B: PROMELA Conversion	85
REFERENCES	94

LIST OF FIGURES

Figure 2.1.	Service oriented architecture metaModel[1]	5
Figure 2.2.	Layout of a BPEL process	7
Figure 2.3.	Receive activity	9
Figure 2.4.	Reply activity	9
Figure 2.5.	Invoke activity	9
Figure 2.6.	Assign activity	10
Figure 2.7.	Wait activity	10
Figure 2.8.	Wait activity	10
Figure 2.9.	Empty activity	11
Figure 2.10.	Sequence activity	11
Figure 2.11.	Scope activity	12
Figure 2.12.	Switch activity	13
Figure 2.13.	While activity	14
Figure 2.14.	Pick activity	15
Figure 2.15.	Flow activity	15

Figure 2.16. Terminate activity	16
Figure 2.17. Throw activity	16
Figure 2.18. Catch activity	17
Figure 2.19. Compensate activity	17
Figure 3.1. Verification methodology of model checking [2]	20
Figure 3.2. LTL semantics examples [3]	22
Figure 4.1. Software development cycle	25
Figure 4.2. Web services architecture stack[4]	29
Figure 4.3. A chunk of types and messages section from the wsdl definition . .	32
Figure 4.4. PROMELA conversion of the wsdl chunk	32
Figure 4.5. Jlex directives	34
Figure 4.6. Jlex symbol functions	35
Figure 4.7. Jlex end of file symbol	35
Figure 4.8. Jlex definitions	36
Figure 4.9. Chunk of jlex rules	36
Figure 4.10. CUP action code	38

Figure 4.11. CUP parser code	38
Figure 4.12. CUP scan with section	39
Figure 4.13. CUP terminal and non terminal symbols	39
Figure 4.14. CUP priorities	40
Figure 4.15. CUP grammar rules	41
Figure 4.16. Example assign activity	42
Figure 4.17. PROMELA conversion of the assign example in Figure 4.16	42
Figure 4.18. Example sequence activity	43
Figure 4.19. PROMELA conversion of the sequence in Figure 4.18	43
Figure 4.20. Example switch activity	44
Figure 4.21. PROMELA translation of the switch activity in Figure 4.20	45
Figure 4.22. While activity example	45
Figure 4.23. PROMELA translation of the while activity in Figure 4.22	46
Figure 4.24. Example BPEL scope activity	47
Figure 4.25. PROMELA translation of the scope activity above	48
Figure 4.26. Example terminate BPEL activity	49

Figure 4.27. PROMELA translation of the terminate activity in Figure 4.26 . . .	49
Figure 4.28. Example flow activity	52
Figure 4.29. PROMELA model of the flow in Figure 4.28	53
Figure 4.30. The model of the first block of the flow in Figure 4.28	54
Figure 4.31. Example throw activity	54
Figure 4.32. PROMELA model of the throw activities in Figure 4.31	55
Figure 4.33. Example pick activity	56
Figure 4.34. PROMELA model of the pick activity in Figure 4.33	57
Figure 4.35. Example invoke activity with its partnerlink definition	58
Figure 4.36. PROMELA translation of the invoke in Figure 4.35	59
Figure 4.37. Example receive activity with its partnerlink	60
Figure 4.38. PROMELA translation of the example receive activity	60
Figure 4.39. Example reply activity with its partnerlink	60
Figure 4.40. PROMELA translation of the example in Figure 4.39	61
Figure 4.41. Compensate activity example	62
Figure 4.42. PROMELA translation of the example in Figure 4.41	63

Figure 4.43. Example fault handlers section	63
Figure 4.44. PROMELA model of the fault handlers in Figure 4.43	64
Figure 4.45. PROMELA definitions of a scope snapshot	65
Figure 4.46. Compensation handler example	66
Figure 4.47. PROMELA translation of the compensation handler in Figure 4.46	67
Figure 4.48. Overview of XSPIN	68
Figure 4.49. XSPIN simulation example	69
Figure 4.50. XPIN verification settings	70
Figure 4.51. Unreachable code verification output	70
Figure 4.52. Service BPEL process	71
Figure 4.53. Verification output of the service process	72
Figure 4.54. Counter example for service process	74

LIST OF TABLES

Table 4.1.	BPEL activities	27
------------	---------------------------	----

LIST OF SYMBOLS/ABBREVIATIONS

SOA	Service Oriented Architecture
BPEL	Business Process Execution Language
WSAT	Web Service Analysis Tool
LTL	Linear Temporal Logic
XML	Extensible Markup Language
XSD	XML Schema Definition
PROMELA	Process Meta Language
SPIN	Simple Promela Interpreter
MSL	Model Schema Language
WSDL	Web Service Definition Language
LR	Left to Right
CUP	Constructor of Useful Parsers

1. INTRODUCTION

Service Oriented Architecture (SOA) is a new system architecture which is based on creating services that can be deployed to independent systems and contribute a distributed service network. SOA is the key concept of the service distribution idea, which means it can be implemented in several ways. BPEL for Web Services which is the most common and standardized way of SOA implementation (Business Process Execution Language) is a language for service oriented architectures (SOA) which is based on xml and was designed for distributed computed using web services over multiple systems.

Formal Verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property. There are mainly two approaches that are used in formal verification which are model checking and logical inference. Logical inference is the mathematical reasoning of the system that is to be verified. Model checking method is the process of modeling a software and exhaustively exploration of all states of the software model. In this work we will use model checking technique for verification of BPEL [5].

Verification by using model checking helps us to ensure that we get an error free system, since it checks all possible states. With the verification of BPEL specifications that is introduced in this work, any kind of error that could possibly cannot be found by testing phase of the software development cycle, can be found during design time. The risk of making a business flow mistake can be decreased, since BPEL is commonly used for business processes. Also in normal software development steps design time errors could occur while testing, if no verification is applied to the generated software. The cost of design time errors that could be realized during testing is more than the implementation errors in the development cycle, since it can effect the design step, development step and testing step. BPEL is commonly used for business processes which means that it comes before the software design phase of the development cycle. Any erroneous business flow implementation will result, too many resources if it is

detected during testing phase. Since it has to enter in design, implementation and testing phases again. In this work the verification of BPEL specification during the design time is introduced which minimizes the design time error costs.

In this work, the correctness of given BPEL specifications, will be checked with the LTL specification of the system by using model checking technique to the specified BPEL specifications. Verification of a BPEL process has been studied with different group of people so far. Verification of BPEL processes using petri nets are explained detailly in Servais's survey paper [6]. According to this model all control flows in a BPEL flow are mapped to petri nets. And after the conversion of BPEL flow into a petri net model a tool named wofbpel [7] is used in order to analyze the converted petri net flow. The detailed, bpel flow to petri net conversion mapping can be found in [8]. With the tool wofbpel it is possible to find unreachable activities, potential conflicting message receipts and discarded messages which are never picked from the message queue [8]. But this tool just gets one process as input which means it could just accomplish structural and basic verifications. A quick BPEL process analyzing is possible with this tool for pursuit of the cases listed above. This approach makes an abstraction of xpath queries, bpel expressions and the data manipulation part of a BPEL process, the scope of their work is the analyzing of the control flows [6].

Another method to verify BPEL processes is by using guarded automata conversion method which was introduced by Xiang Fu[9] in his work. Firstly the xml data is parsed and converted to guarded automata in this work. After the conversion, the guarded automata representation is converted into the PROMELA language by the tool wsat[10]. The translation from BPEL to guarded automata is explained in Fu's work with detail. He also mentions a different way of handling and converting xpath functions using MSL[9] (Model Schema Language) representation. After xml guarded automata is generated by the tool named WSAT which is used to convert the guarded automata representation into PROMELA and verification using SPIN can be done by using this approach. In WSAT a limited subset of BPEL is converted to PROMELA which are assign, flow, invoke, receive, reply, sequence activities in BPEL. A verification application is specified in the paper by producing an LTL specification of the common

stock BPEL flow example and running SPIN with the output of the tool WSAT. The LTL specification used for the common stock BPEL process is that: "if the register message contains a stockid, then eventually there should be a request containing that stockid, if nothing wrong happens". And it is shown that this condition is not always true by using SPIN to generate a counter example.

It is possible to create a direct mapping from BPEL processes to PROMELA processes which will be implemented in the scope of this work. The conversion of any BPEL process with any activities containing sequence, receive, assign, invoke, reply, switch, empty, while, terminate, scope, flow, throw, pick and fault handlers will be done and mapped to PROMELA processes. The main purpose of this work is to model any BPEL source file into a PROMELA model, so that it can be verified by SPIN. Our goal is to model any given BPEL workflow written in BPEL language and verify those process models using SPIN. By using the generated model which is explained in this paper, it is possible to search for unreachable codes, assertions and unhandled exceptions that might rise from the specified BPEL processes. It is also possible to give an LTL system specification and search for a counter example if that specification holds for the supplied BPEL processes. The reason why BPEL and SPIN picked for this process is that BPEL is widely used in SOA systems and SPIN is the most common used verification tool that is used by most of the communities.

In second chapter, the concept of SOA and BPEL are explained and we give a detailed information about BPEL language, its structure and the activities that form a BPEL process. Basicly a BPEL process consists of partnerlinks, variables, fault-handlers, compensation handlers, event handlers and activities. Those terms and explanation with examples are shown in this chapter.

In the third chapter an overview of formal verification methods is specified. The methods that are being used so far for different kind of verification purposes are mentioned and the method that will be used in this work is discussed in this part. The verification of a software and tools to verify specifications are explained in this chapter.

In the fourth chapter the work of this paper is discussed in detail starting with the objective of this work. The verification works done so far are compared with this work and the explanation of each difference is given in this chapter. The variable types which can be referenced from BPEL language are defined in wsdl and xsd files. Modeling of variables used in BPEL processes and the definition of variable types are explained. Then the solution of the mapping problem between PROMELA, BPEL expressions and conditions that can be used in processes are explained. After explaining lex and cup model in order to parse BPEL expressions, finally modeling the activities in a BPEL process and the verification step of the generated PROMELA model by using the SPIN tool are discussed in this chapter.

At the end in the fifth chapter, conclusions, the evaluation of the generated results and possible future directions are mentioned.

2. OVERVIEW OF SOA AND BPEL

2.1. SOA Concept

The purpose of SOA is to create a large amount of resources together to form a scalable distributed systems running like adhoc systems. A general SOA metamodel is shown in the figure below.

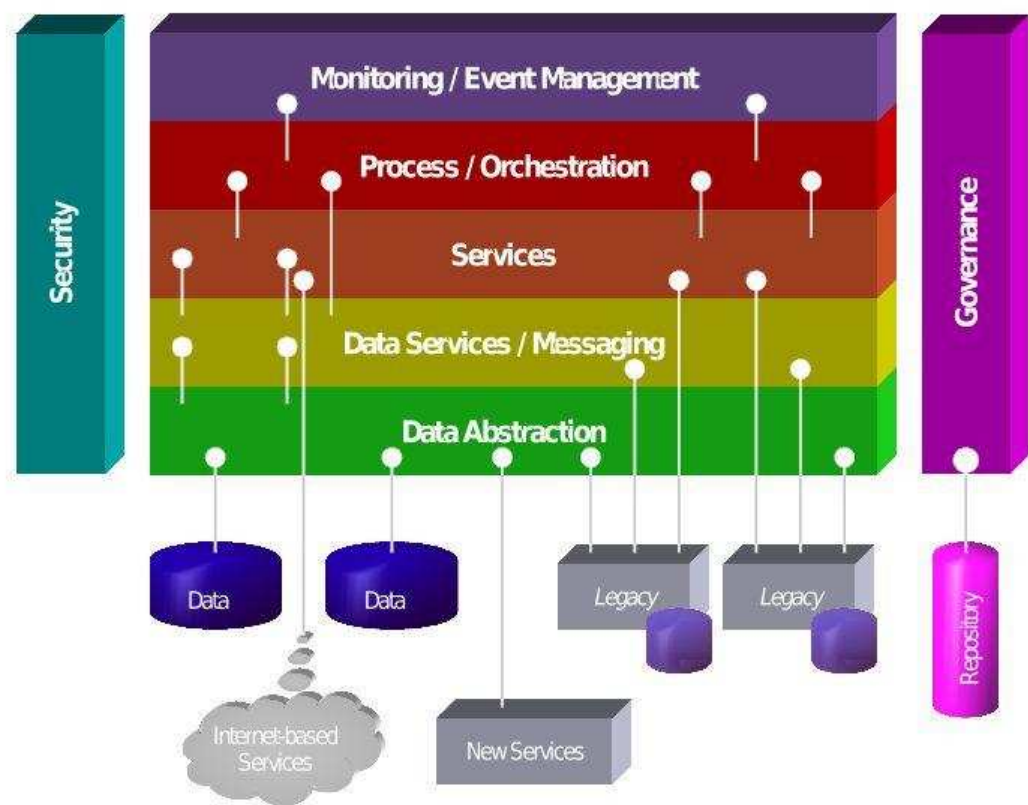


Figure 2.1. Service oriented architecture metaModel[1]

In service oriented architectures at the most bottom level the data abstraction and communication overhead is done. After those structures are completed the main block which is the service block is constructed and basically everything that will be given to the requester will be a registered service in the system. After services are generated another problem arises, which is the need of process execution order or basic routing decisions,

etc. Those problems lead us to a new block which is called the service orchestration layer. Service orchestration is also another new terminology which has an important role in a SOA architecture.

The most common orchestration technology is the BPEL technology. Bpel is a new language created for business process implementation by its own syntax. And it is easier to use BPEL in order to implement a service orchestration system to form a SOA architecture.

2.2. BPEL

BPEL is an XML based definition language that is used to form business processes that interact each other with web services which is widely used in implementation of service orchestration systems. BPEL is constituted by activities that form a business flow. A flow is the output of BPEL activities that can be run by a BPEL engine. The activities of a BPEL process are explained below detailly. And examples from oracle JDeveloper IDE are attached to make it more clear.

2.2.1. Main structure

A general BPEL process layout will look like the structure in Figure 2.2.

2.2.2. Partnerlinks

Partnerlinks section is the section that the partnerlinks, the BPEL process is going to use are described. A partnerlink is a link from the BPEL process to the outer world like a web service, a database adapter, etc. A partnerlink has a partnerlinktype and one or two roles depending on the remote process's behaviour.

If the remote process that will be called is an asynchronous process which means that it will make a callback invoke to the calling process after completing the process in background. Then two roles are required one of them is to initiate a call to the

```

<process name = "purchaseOrderProcess" targetNamespace = "http://example.com/ws-bp/purchase" xmlns =
"http://docs.oasis-open.org/wsbpel/2.0/process/executable" xmlns:lns =
"http://manufacturing.org/wsd/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 = "shippingRequest" messageType = "lns:shippingRequestMessage"/>
    <variable name = "shippingInfo" messageType = "lns:shippingInfoMessage"/>
    <variable name = "shippingSchedule" messageType = "lns:scheduleMessage"/>
  </variables>
  <faultHandlers>
    <catch faultName = "lns:cannotCompleteOrder" faultVariable = "POFault" faultMessageType =
"lns:orderFaultType">
      <reply partnerLink = "purchasing" portType = "lns:purchaseOrderPT" operation = "sendPurchaseOrder"
variable = "POFault" faultName = "cannotCompleteOrder"/>
    </catch>
  </faultHandlers>
  <sequence>
    <receive partnerLink = "purchasing" portType = "lns:purchaseOrderPT" operation = "sendPurchaseOrder"
variable = "PO" createInstance = "yes">
      </receive>
      ..... (activities described below)
      <reply partnerLink = "purchasing" portType = "lns:purchaseOrderPT" operation = "sendPurchaseOrder"
variable = "Invoice">
        </reply>
      </sequence>
    </process>

```

Figure 2.2. Layout of a BPEL process

remote asynchronous service and the second is to receive a callback from the remote process. Those roles are identified by the attributes "myrole" and "partnerrole" which makes their usage more obvious with those naming standards.

2.2.3. Variables

Variables section is the section that variables, the BPEL process is going to use are described. A variable is a declaration of a variable similar to most of the programming languages. The message types and schema types can be referenced and used here as a variable type after importing necessary definition files.

2.2.4. FaultHandlers

Faulthandlers is the section where fault handler for specific exceptions can be specified. In BPEL processes every fault has a faultname with a namespace, and by using those exception names a fault handler can be written for any fault in BPEL. The fault handler mechanism in BPEL also gives a faultvariable structure to pass a variable within the nested faults.

2.2.5. Event Handlers

Event handler is like the event handler mechanism of most of the object oriented programming languages. It handles specific events by the defined activities inside an event handler. In BPEL there should be at least one message handler which is defined with onMessage activity or one alarm handler which is defined by onAlarm inside the event handlers section. If an exception occurs during the execution of event handler the fault handlers of the scope, that has the event handler, should be executed.

2.2.6. Basic Activities

2.2.6.1. Receive: Receive is the activity that causes the business process to wait till the specified message arrives from the partnerlink stated in receive activity.

The receive activity in Figure 2.3 will wait for a message from the partnerlink client and operation name process and the value of the incoming data will be saved to the inputVariable.

```
<receive name="receiveInput" partnerLink="client" portType="client:makcay1" operation="process"
variable="inputVariable" createInstance="yes"/>
```

Figure 2.3. Receive activity

2.2.6.2. Reply: Reply activity allows a process to send a message according to a message that was received before. The combination of receive and reply activities form a request-response operation that is defined in the porttype section of a WSDL document[11].

```
<reply name="replyOutput" partnerLink="client" portType="client:makcay1" operation="process"
variable="outputVariable"/>
```

Figure 2.4. Reply activity

In the reply activity in the Figure 2.4 the variable `outputVariable` will be sent over the partnerlink name `client` with the operation name `process`.

2.2.6.3. Invoke: This activity allows the BPEL process to call external services which are defined by partnerlinks in the BPEL process's partnerlinks definition section. The call of the service might be one-way or two-way according to the remote procedure's definition.

```
<invoke name="Invoke_1" portType="ns1:makcay2" inputVariable="Invoke_1_process_InputVariable"
outputVariable="Invoke_1_process_OutputVariable" partnerLink="makcay2" operation="process"/>
```

Figure 2.5. Invoke activity

The invoke activity in Figure 2.5 makes an invocation on partnerlink name `makcay2` which is defined before. And supplies `Invoke_1_process_InputVariable` as input variable and the returning data from this invocation is saved in the outputvariable `Invoke_1_process_OutputVariable` by calling the `process` operation on the specified partnerlink.

2.2.6.4. Assign: Assign activity is used for manipulating the data in the BPEL process, such as copying the data from one variable to a new destination. Inside an assign activity multiple copy instances can be present, all of those copy instructions are called the copy rules and will be run sequentially in runtime.

```
<assign name = "Assign_1">
  <copy>
    <from variable = "inputVariable" part = "payload" query = "/client:makcay1ProcessRequest/client:input"/>
    <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
  </copy>
  <copy>
    <from expression = '"1"'/>
    <to variable = "Invoke_1_process_InputVariable" part = "payload" query =
"/ns1:makcay2ProcessRequest/ns1:input"/>
  </copy>
</assign>
```

Figure 2.6. Assign activity

The first activity in Figure 2.6 is copying from the input from inputVariable's payload part. The part is the same as the wsdl message type part definition. Note that not only basic data manipulations but also running queries on variables on specific parts are allowed.

2.2.6.5. Wait: This activity results the BPEL process to wait for a certain time after the flow reached to wait activity or it can cause the flow to wait until a specific date.

```
<wait name="Wait_1" for="P1DT1M25S"/>
```

Figure 2.7. Wait activity

```
<wait name="Wait_1" until="'2008-04-30T03:46:04+03:00'"/>
```

Figure 2.8. Wait activity

The first wait in Figure 2.7 causes the process to wait for 1 day, 1 minute and 25

seconds after the process has started to execute that part. The second wait in Figure 2.8 causes the process until the specified date comes.

2.2.6.6. Empty: Empty is the activity that does nothing and has no meaning at all. This activity is used in some cases that nothing has to be done. For example a fault that needs to be caught but suppressed this activity can be used[11]. This activity might also be used for synchronization of concurrent activities[11].

```
<empty name="Empty_1"/>
```

Figure 2.9. Empty activity

This activity has no meaning at all. Just a no operation activity which is equal to skip directive in PROMELA.

2.2.6.7. Sequence: Sequence activity is the activity that enables to create a sequence of activities inside it that will be run in sequential order by the BPEL engine as shown in Figure 2.10. Note that if there has to be more than one activity within a process, all of those activities has to be put in a sequence. Most of the IDEs put that sequence automatically.

```
<sequence name = "Sequence_1">
  <wait name = "Wait_1" until = "'2008-04-30T03:46:04+03:00'"/>
  <assign name = "Assign_2">
    <copy>
      <from variable = "Invoke_1_process_OutputVariable" part = "payload" query =
"/ns1:makcay2ProcessResponse/ns1:result"/>
      <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
    </copy>
  </assign>
  <empty name = "Empty_1"/>
</sequence>
```

Figure 2.10. Sequence activity

2.2.6.8. Scope: Scope activity is like the scopes in any other programming language, it is basically creating a block that has its own local variables, fault handlers and compensation handlers. A scope can be thought as a small BPEL process like any other blocks in other programming languages.

```
<scope name = "Scope_1">
  <faultHandlers>
    <catch faultName = "bpws:selectionFailure">
      <empty name = "selectionFailure"/>
    </catch>
  </faultHandlers>
  <sequence name = "Sequence_1">
    <wait name = "Wait_1" until = "'2008-04-30T03:46:04+03:00'"/>
    <assign name = "Assign_2">
      <copy>
        <from variable = "Invoke_1_process_OutputVariable" part = "payload" query =
"/ns1:makcay2ProcessResponse/ns1:result"/>
        <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
      </copy>
    </assign>
    <empty name = "Empty_1"/>
  </sequence>
</scope>
```

Figure 2.11. Scope activity

In the scope activity in Figure 2.11 if a selectionFailure exception occurs this exception will be handled by this scope and the error will be suppressed and the rest of the process will not be effected by this error.

2.2.7. Control Activities

2.2.7.1. Switch: Switch activity is used to make decisions based on the given conditions for each case. A switch block consists of several case blocks and an optional otherwise. Sequentially each case condition is checked and if any of the conditions is satisfied, then the first satisfied case block is entered and run by the BPEL engine. If no matching conditions found after checking all case conditions then the otherwise block is executed if specified any.

```

<switch name = "Switch_1">
  <case condition =
"bpws:getVariableData('Invoke_1_process_OutputVariable','payload','/ns1:makcay2ProcessResponse/ns1:result') =
&quot;1&quot;">
    <assign name = "Assign_3">
      <copy>
        <from expression = '"One"/>
        <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
      </copy>
    </assign>
  </case>
  <case condition =
"bpws:getVariableData('Invoke_1_process_OutputVariable','payload','/ns1:makcay2ProcessResponse/ns1:result') =
&quot;2&quot;">
    <assign name = "Assign_4">
      <copy>
        <from expression = '"Two"/>
        <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
      </copy>
    </assign>
  </case>
  <otherwise>
    <assign name = "Assign_6">
      <copy>
        <from expression = '"Unknown"/>
        <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
      </copy>
    </assign>
  </otherwise>
</switch>

```

Figure 2.12. Switch activity

In the switch activity in Figure 2.12 two case conditions are supplied and if none of the case conditions apply, the otherwise condition is selected in the runtime depending on the `Invoke_1_process_OutputVariable` variable's result field which is in the payload part. Note that `bpws: getVariableData` is a function that is implemented by the BPEL engine which is to get a BPEL variable data value inside an expression or condition.

2.2.7.2. While: While activity is similar like the while loops in all programming languages. While creates a loop and runs the block inside it till the condition specified for the while activity no longer holds.

```
<while name="While_1" condition="bpws:getVariableData('counter')&lt;10">
  <assign name="Assign_7">
    <copy>
      <from expression="bpws:getVariableData('counter')+1"/>
      <to variable="counter"/>
    </copy>
  </assign>
</while>
```

Figure 2.13. While activity

While activity in Figure 2.13 is a very basic example which makes a loop till the counter reaches value 10. There is no difference from the basic while structure from other structured programming languages.

2.2.7.3. Pick: Pick is an event based activity which actually waits for the events defined inside the pick block. It has onMessage and onAlarm options that can be used to specify events. The first occurred event will be picked according to the event time. The onMessage section waits for a message from a partnerlink and a specified operation. The coming message will be copied to the specified variable in the message handler. OnAlarm part is similar to the wait activity accept this time an event occurs when the time comes.

In the pick activity shown in the Figure 2.14 a basic timeout structure for an asynchronous invocation is implemented. The pick activity will check the events specified which are AsyncBPELService partnerlink's onResult operation and 15 second timeout. Since the first picked event will run if the remote part doesn't answer for at least 15 seconds, then the activity will terminate waiting.

```

<pick name="receiveResult">
  <onMessage partnerLink="AsyncBPELService" portType="services:AsyncBPELServiceCallback"
operation="onResult" variable="response">
    <assign>
      <copy>
        <from variable="response" part="payload"/>
        <to variable="output" part="payload"/>
      </copy>
    </assign>
  </onMessage>
  <onAlarm for="PT15S">
    <terminate/>
  </onAlarm>
</pick>

```

Figure 2.14. Pick activity

2.2.7.4. Flow: Flow activity enables multiple flow branches to run concurrently. No matter what happens when this activity is reached the sequences within this activity will be run parallel.

```

<flow name="Flow_1">
  <sequence name="Sequence_2">
    <invoke name="Invoke_3"/>
    <assign name="Assign_9"/>
    .....
  </sequence>
  <sequence name="Sequence_2">
    <invoke name="Invoke_2"/>
    <assign name="Assign_8"/>
    .....
  </sequence>
</flow>

```

Figure 2.15. Flow activity

In the flow activity in Figure 2.15 the two sequences will be initiated at the same time and the flow activity will finish after two sequences are over which means the next activity after flow activity will be executed after all the sequences inside a flow activity are over.

2.2.7.5. Terminate: Terminate activity is one of the most common structures in a programming language which causes the BPEL process to terminate and end the execution. It is similar to the exit structure in any programming language.

```
<terminate name="Terminate_1"/>
```

Figure 2.16. Terminate activity

Terminate activity's usage is very simple like the empty activity as it can be seen in Figure 2.16.

2.2.7.6. Throw: Throw activity is used in order to raise an exception during the execution of a BPEL process. Any custom or system defined exception can be thrown by this activity. Any fault can be thrown by a defined faultname with a namespace inside the process wsdl file. Another reasonable input for the throw activity is the variable that caused the fault.

```
<throw name="Throw_1" faultName="bpws:selectionFailure" faultVariable="inputVariable"/ >
```

Figure 2.17. Throw activity

The predefined bpws:selectionfailure fault is thrown in the throw activity in Figure 2.17, passing the faultvariable inputVariable. At this line the execution will stop and the execution will branch to the faulthandlers if there are any.

2.2.7.7. Catch: Catch is the activity that catches a fault that is thrown by the throw activity or coming from the BPEL engine itself. It is similar to the any exception enabled structured programming language. Catch blocks are placed inside the faulthandlers section in BPEL process and if none of the catch blocks catch an exception, there is an optional catchall primitive which basically catches all exceptions.

```

<faultHandlers>
  <catch faultName="bpws:selectionFailure" faultVariable="inputVariable">
    <empty name="Empty_4"/>
  </catch>
  <catchAll>
    <empty name="Empty_5"/>
  </catchAll>
</faultHandlers>

```

Figure 2.18. Catch activity

2.2.7.8. Compensate: Compensate activity enables a BPEL process to rollback the several actions that are successfully done before. In order to do that compensation handlers of scopes should be written. Compensation handler is only supported inside a scope. And a compensation handler can only be invoked from a fault handler or within another compensation handler. It is also possible to compensate an inner scope.

```

<scope name = "Scope_2">
  <faultHandlers>
    <catchAll>
      <sequence name = "Sequence_3">
        <empty name = "Empty_3"/>
        <compensate name = "Compensate_1"/>
      </sequence>
    </catchAll>
  </faultHandlers>
  <scope name = "Scope_3">
    <compensationHandler>
      <empty name = "rollback"/>
    </compensationHandler>
    <sequence name = "Sequence_2">
      <receive name = "receiveInput" partnerLink = "client" portType = "client:makcay1" operation = "process"
variable = "inputVariable" createInstance = "yes"/>
    </sequence>
  </scope>
  <throw name = "Throw_1" faultName = "bpws:selectionFailure" faultVariable = "inputVariable"/>
</scope>

```

Figure 2.19. Compensate activity

In the compensate activity in Figure 2.19 a scope and an inner scope's compensation

handler is shown. The scope will always throw exception and this exception will be handled by the fault handler and the fault handler will call the compensation handler block by the compensate activity. Although it is not very common to use this kind of compensation handlers and compensate activities in BPEL those activities satisfy a transactional view on the process.

3. OVERVIEW OF MODEL CHECKING FOR SOFTWARE VERIFICATION

The most commonly used principal methods for the validation of complex systems are testing, simulation, model checking and deductive verification methods [12]. Testing of software is done by people or some testing tools with the given test cases which are in most of the cases declared by the developer of the software. Testing is done by the usage of the system itself. One step further of testing is the simulation. In simulation a model of the system that is being validated is created and simulation with specific inputs is generated on this model. The main difference between testing and the simulation is that the testing is done on real system, but in simulation it is done on the model generated for this purpose. Model checking can be called as the exhaustive testing of all the behaviors of a generated software model [12]. This definition tells us that a model has to be created again like in simulation, but this time instead of executing a simple run, all the possible states will be explored. The last method is the proof of the software system mathematically which is done by manually by using mathematics and some proof assistant software.

3.1. MODEL CHECKING

Model checking for software verification purpose is the use of algorithms that are executed by software verification tools like SPIN in order to verify the correctness of a software system [2]. The first thing that has to be done to accomplish the verification process is to create a model representing the software that is wanted to be checked. Then this generated model is checked against the specifications supplied by the user which is the desired behavior of the software. After specification of the model and the specification of the software is supplied, the verification software runs and explores all the states in the generated model.

The purpose of exploring all the states in a model is to check if there exists any scenario that the specification of the software that is trying to be verified is not supplied

at all. If the created model of the software is right, then the counter example of the unsupplied specification indicates us that the software we are working on does not satisfy the specifications which results with a scenario that has to be corrected in the software model. At this point the necessary changes have to be done and the verification steps have to be repeated again till there exists no counter example that does not satisfy the specifications [2].

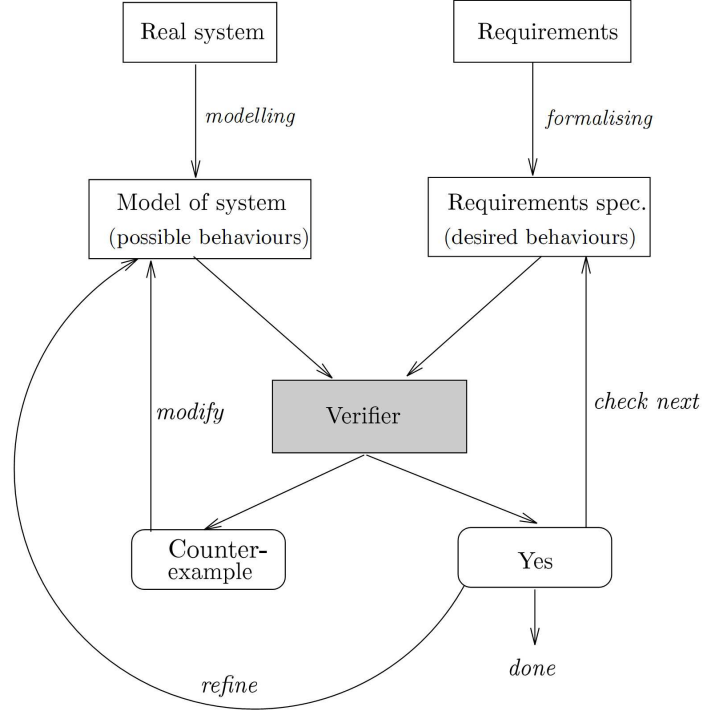


Figure 3.1. Verification methodology of model checking [2]

In Figure 3.1, the verification steps of a model created from a software that is wanted to verify is shown. First of all a model from the real software system has to be created which is shown by the possible behaviors above. Then the specifications of the real system have to be stated in a specific way. For our case this specification language is the LTL formulas that can be declared while the verification run of SPIN. After generating the model and the specifications (desired behaviors), those two inputs will be given to the verification tool which is the SPIN tool in this work. The aim of the verification step is to explore all the possible behaviors if the desired behaviors are satisfied for all possible executions. If a counter example can be found after the verification step, this means the software doesn't comply to the specs specified and

the software model has to be modified, so that the counter example is fixed. After fixing the problem, verification step is run again till the software system is verified. After the verification of the current system is completed with no errors, new features to the current software model can be added since the current model is verified with the current specifications.

3.2. Linear Temporal Logic (LTL)

During the verification process LTL expressions can be used to check if that LTL statement holds for all state spaces. Linear Temporal Logic (LTL) is a modal temporal logic with modalities that refer to time [13]. The power of LTL is that it is possible to generate a formula about future states of the software or a condition that will eventually hold. There are some special characters that have special meanings in an LTL formula which are used to define the semantics of LTL. The formula $\Box A$ means that A is true for all states, $\Diamond A$ means A is true eventually, $\bigcirc A$ means that A is true at the next state after the current state. $A \cup B$ means until B is true A has to be true. ARB means that B is true until the first state A is true or B is forever true if A is never true. The semantics of the Linear Temporal Logic is shown from the definition taken from the Korovin's lecture notes [3]. LTL semantics can be used during the verification process in SPIN when never claim expressions are being specified. SPIN tries to find if in any state in the state space of the verified software, the never claim LTL specification is not satisfied. If SPIN finds an example scenario where the specified LTL formula is not valid a counter example is reported by SPIN. The LTL semantics definition taken from [3] is shown below.

Let $\pi = s_0, s_1, s_2, \dots$ be a sequence of states and A be an LTL formula. We define the notion *A is true on π* , denoted by $\pi \models A$, by induction on A as follows. For all $i=0,1,\dots$ denote by π_i the sequence of states $s_i, s_{i+1}, s_{i+2}, \dots$ (note that $\pi_0 = \pi$).

- (1) $\pi \models \top$ and $\pi \not\models \perp$.
- (2) $\pi \models x = v$ if $s_0 \models x = v$.
- (3) $\pi \models A_1 \wedge \dots \wedge A_n$ if for all $j=1,\dots,n$ we have $\pi \models A_j$;

- $\pi \models A_1 \vee \dots \vee A_n$ if for some $j=1,\dots,n$ we have $\pi \models A_j$.
- (4) $\pi \models \neg A$ if $\pi \not\models A$.
- (5) $\pi \models A \rightarrow B$ if either $\pi \not\models A$ or $\pi \models B$;
 $\pi \models A \leftrightarrow B$ if either both $\pi \not\models A$ and $\pi \not\models B$ or both $\pi \models A$ and $\pi \models B$.
- (6) $\pi \models \bigcirc A$ if $\pi_1 \models A$;
 $\pi \models \Diamond A$ if for some $i = 0, 1, \dots$ we have $\pi_i \models A$;
 $\pi \models \Box A$ if for all $i = 0, 1, \dots$ we have $\pi_i \models A$.
- (7) $\pi \models A \cup B$ if for some $k = 0, 1, \dots$ we have $\pi_k \models B$ and $\pi_0 \models A, \dots, \pi_{k-1} \models A$;
 $\pi \models ARB$ if for all $k \geq 0$, either $\pi_k \models B$ or there exists $j < k$ such that $\pi_j \models A$. [3]

The graphical representation of the LTL semantics definition can be seen graphically in the Figure 3.2.

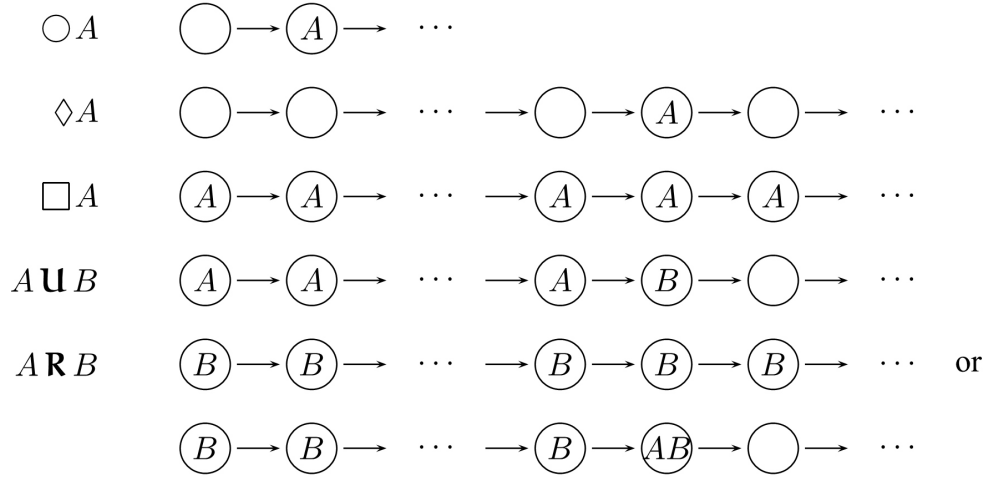


Figure 3.2. LTL semantics examples [3]

3.3. SPIN

SPIN is one of the most common tools that is used in software verification by model checking method. SPIN can simulate a model that is written in PROMELA and it also generates a verifier which is a C source code to verify the software by exploring all states. Possible verification methods SPIN can handle are random simulation, guided simulation and interactive simulation. In random simulation SPIN randomly chooses

one of the enabled branches which is an undeterministic behavior [14]. Guided simulation is the simulation mode only that can be applied before and done with error. Guided simulation runs the error trace or counter example generated by the previous run of SPIN. Interactive simulation is the simulation mode that SPIN asks the user interactively on each branching part. If more than one of the statements are enabled at the same time, then SPIN will ask the user to select the execution path.

There are four things that SPIN can do in the verifier implementations. First of all SPIN can detect the assertion violation rules coded with the assert directive of PROMELA. During the running process if the condition inside the assertion directive evaluates to false then an assertion violation is raised and the verification stops, since the assertion failed. Another capability that SPIN can verify is the acceptance test for the verifier which is checking the acceptance cycle runs or the opposite which is non progressive cycles. In order to use those capabilities accept or progress labels have to be specified in the model, then the verifier has to be run. SPIN also checks the valid end states of a process, if the label end is used correctly. If the process terminates with another state then the verification stops with the end state error. The last thing and may be one of the most important features of the SPIN verifier is the never claim check of the verification progress. During the execution of SPIN, it searches all the states and checks if the never claim statement holds for all cases. If a case occurs that the never claim doesn't happen to hold for this case, this scenario is returned by the verifier as a counter example by the verifier.

4. MODEL CHECKING OF BPEL SPECIFICATIONS USING SPIN

4.1. Main Objective

The main objective in this work is to apply model checking techniques that are used in software verification process to BPEL programming language and create an environment to verify any BPEL process in a specified subset of BPEL. In order to accomplish the verification of any BPEL process given, the specified BPEL process has to be modeled in PROMELA which is the language for SPIN tool. Regarding the wsdl types and xsd elements that can be used as a variable anywhere inside a BPEL process, those types have to be modeled in PROMELA too. BPEL language is based on xml and xpath queries which have to be modeled in PROMELA.

The reason why SOA and BPEL are chosen in this thesis is that SOA is the new increasing and developing architecture design those days. SOA enables the developers to develop, integrate and deploy business services rapidly [15]. SOA enables the use of previously deployed applications each also named as a service and BPEL is chosen as the SOA language within the scope of this work. Because BPEL is the most commonly used and well known SOA language that is widely used those days.

The reason, why software verification is important, is that today most of the software are running in areas where failure or system shutdown cannot be acceptable[16]. The most impressive example on that would be the Ariane 5 rocket which exploded on 1996, because of a floating number conversion error in the software. For those kind of software where no error is acceptable, there is a need of reliable software that its correctness can be verified. Some implementation or design errors that can be done before the testing step, could also never come up during the tests, since testing cannot dive into all possible states of a software. Software verification will guarantee that the software that is checked, complies with the LTL specifications we have. In Figure 4.1 the development cycle is shown and the place of model checking is shown during this

process.

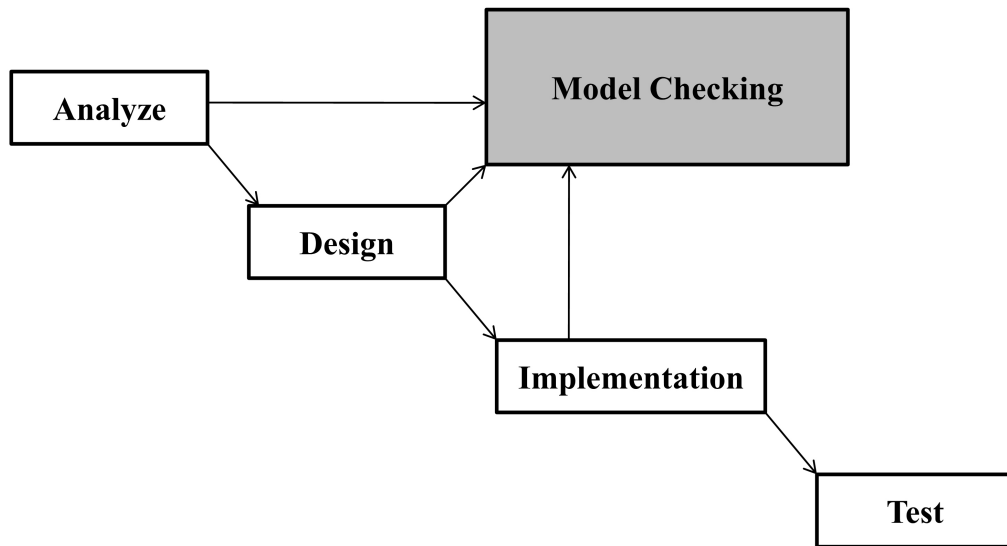


Figure 4.1. Software development cycle

Also, the importance of the verification of BPEL specifications is too high regarding the software development cycle shown in the Figure 4.1. Since BPEL is a language generally used by business analysts, the effects of mistakes in business process cost more than the errors that could possibly be done during the implementation step. Because in that case the business analysts should fix the error, then the correct design has to be done, then the implementation has to be done again and should go to the test again. So, the aim of this work is to minimize those kind of cost inefficient errors. Since the verification of BPEL specifications will give us the opportunity to realize those kind of errors during the design time.

In this work The main concepts of BPEL and xpath extensions and basic xsd types are modeled in PROMELA which are basically a subset of the BPEL activities of sequence, assign, switch, empty, while, terminate, scope, flow, throw, invoke, reply, receive, pick, wait and fault handlers. In the closest work done so far is the phd thesis of Xiang Fu[9] which was targeting same subject with a different way using guarded automata and MSL conventions. In this approach they focused on assign, flow, invoke, receive, reply, sequence activities instead. Invoke, receive and reply of a BPEL process is the communication of a business process with other world. In their work several

BPEL files and several wsdl files are used in verification process.

Another important mapping between BPEL and PROMELA is the mapping of the exception handling. SPIN has the capability of assertion verifications which is commonly used for software verification purposes. In BPEL, throw activity is an activity that causes to throw an exception. If we map this throw activity into assertion of PROMELA, then a verification of a BPEL process can be done automatically after the conversion process is done. Other exceptions that might come from the BPEL engine are not important for us like remote timeout caused by network. Because those kind of exceptions will never occur in our case, since we don't use any BPEL engine. So, exception throwing and catching concepts of BPEL are modeled in PROMELA and unhandled exceptions are modeled to PROMELA assertions to be caught during the verification process by SPIN. Assertions in PROMELA cause the same effect with unhandled exceptions in BPEL. An assertion in PROMELA will cause the execution of a process to stop with an assertion error if the condition is not satisfied.

The verification of any BPEL process is the key idea to verify each service in a SOA suite. The details of BPEL processes are explained above with details. In this work the verification of a system is targeted, so this software block can be thought as a closed box that will be explored in detail by state space searches using the SPIN tool. The BPEL activities covered in this work and covered by [9] and [8] that are listed in the references section are shown below.

The scope and the activities mapped by this work can be seen in table 4.1. Invoke activity has a very wide concept in BPEL processes, since a process might invoke a file adapter, database adapter or a web service implemented in java. In the scope of this work, BPEL to BPEL invocations are implemented and verified. The mapping of wait and onAlarm activities which are related with timings creates problems. Those time specific actions are mapped to PROMELA undeterministic decision branching mechanism. Wait is modeled to skip directive in PROMELA. Even a wait activity does nothing except waiting for a period of time, it might have a source and target specification for a link inside a flow activity. So, all activities have to be implemented

Table 4.1. BPEL activities

BPEL Activities	covered	covered previously[9]	covered previously[8]
Sequence	+	+	+
Assign	+	+	+
Switch	+	-	+
Empty	+	-	+
While	+	-	+
Terminate	+	-	+
Scope	+	-	+
Flow	+	+	+
Compensate	+	-	Limited[7]
Throw	+	-	+
Invoke	+	+	+
Reply	+	+	+
Receive	+	+	+
Pick	+	-	+
Wait	+	-	+
Compensation Handlers	+	-	+
Fault Handlers	+	-	+
Event Handlers	-	-	Limited[7]

even if they do nothing remarkable like empty and wait activities.

As it can be seen in the table 4.1, previous works done so far are listed and their implementations are summarized too. The previous implementation in [9] tells us that the main goal of the work done there is the communication of processes and very basic activities. The decision making activities and control activities are not implemented in their work.

In the petri net approach [8] shown in table 4.1, it is seen that every BPEL activity is mapped to petri nets. Then this petri net representation is analyzed by the WOFBPEL[7] tool. The limitation of their approach is about the compensation and event handlers. For compensation they assume compensate command shows before its compensation handler in the scope and they assumed that all the compensation handlers should be the same way as it is defined in their example that is stated in their paper. Another limitation is the multiple simultaneous event handler execution is not supported in their petri net generation approach [7].

In petri net approach most of the BPEL activities are mapped to petri nets, but the important part for the verification comes after that conversion. Their tool reads the converted petri net and makes it possible to verify that software in three different ways. It makes reachability analyzes which is the controlling of the unreachable activities in a BPEL process. Secondly it finds competing message-consuming activities which means to find the activities that wait for a message from the same partnerlink, same porttype and the same operation. These kinds of message waiting activities create runtime problems in BPEL processes. Finally WOFBPEL tool finds the garbage collection queued messages, which is the job of finding unread and garbage collected messages in the BPEL process [7].

Note that the petri net approach for verifying BPEL processes create a conversion from BPEL to petri nets and this generated petri net is analyzed with the tool WOFBPEL. But the important point of this approach is that in petri net approach, xpath queries, bpel expressions and the data manipulation are abstracted. WOFBPEL tool analyzes the generated petri net for the three kind of problems. So, in this approach we have a limited verification since we don't have a running model and LTL formula verification in this approach.

4.2. Web Service, WSDL and XSD definitions

Web service is one of the most popular standard ways of software communication protocols that is widely used in most of the programming languages. Web services

make the functionality of a software component available in the internet in a standard way. Web services are constituted of several layers which is also known as the web service stack. This stack is defined for web service implementations in order to make the web service communication standard, so that all web service implementations could communicate without any problems. The standard web service stack defined in [4] is shown in the Figure 4.2.

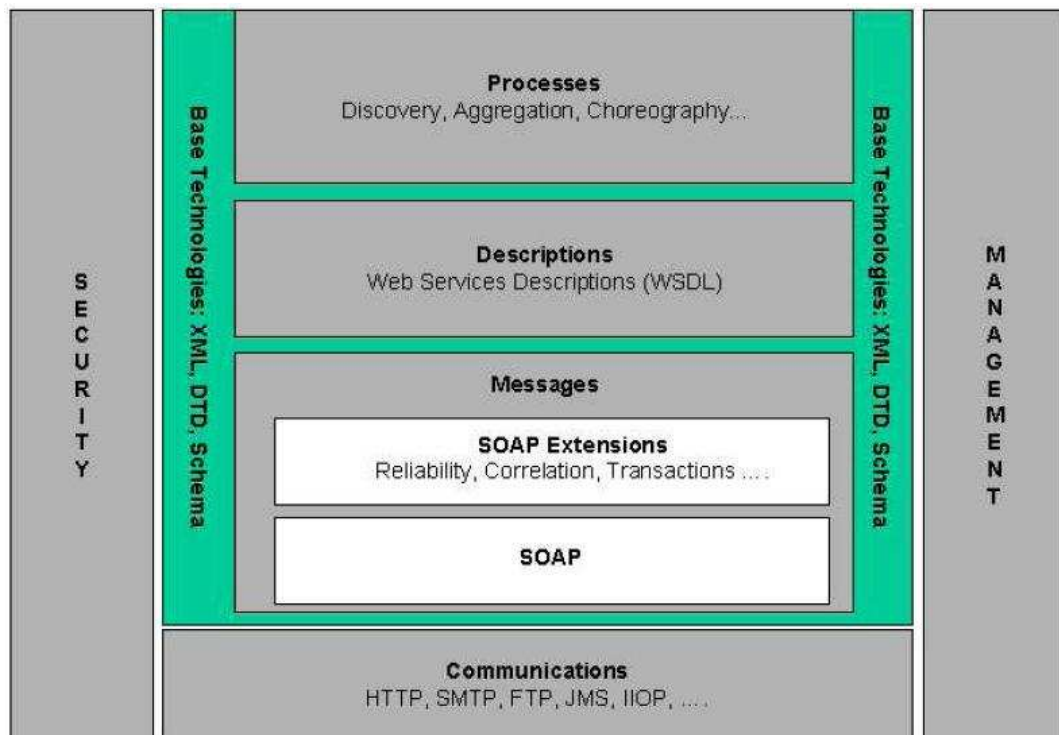


Figure 4.2. Web services architecture stack[4]

The communication layer of the webservice stack is the basic networking layer which is like HTTP, FTP, SMTP, etc. The task of the communication layer is to pass the received or sent data to the appropriate destination or to the upper layer which is messaging layer. In web services the communicating language is XML, which increases the flexibility and extensible data format. On the communication layer, messaging layer includes SOAP and SOAP extensions. SOAP message type has the required information in order to call the appropriate service or the method. In the descriptions layer the definitions of the web services are done. WSDL stands for Web Service Definition Language and WSDLs are used in order to identify web services including its input, output, exposed services and all kind of information to express the web

service. In order to define the data types inside the wsdl, XSD types can be imported or defined in an inline style inside a WSDL file. Briefly a WSDL file is the descriptor of a web service. The highest layer in the web service layer is the processes layer as it can be seen including UDDI discovery.

The relation between BPEL and Web Services is that the interaction of BPEL processes with the outer world is accomplished with web services. As mentioned before, BPEL processes have the partnerlink definitions inside them in order to communicate with other web services. In order to do that there has to be some change in the implementation of the WSDL definitions in order to create a link between web service and BPEL processes. That change is done inside the WSDL files by changing the WSDL extensions definitions inside it and put the partnerlink definitions inside WSDL files. In BPEL process management implementations the request is received and transferred to the upper layers ending with the BPEL process.

4.3. Modeling WSDL and XSD files in PROMELA

Basically a BPEL process is a web service itself which means that the process itself should have a web service definition language like all other web services do. A BPEL process is totally compound by three objects which are its BPEL source code, wsdl files and BPEL descriptor file which holds the properties of the flow and partnerlink definitions.

The start point of modeling a BPEL process into PROMELA is the modeling of the variable declaration and complex types defined in the wsdl and the xsd files that can be imported in the wsdl files. A wsdl file consists of several parts and it is in an xml format. The structure of a wsdl file is composed of multiple sections which are types, message, porttype, binding, port and service.

The brief explanations that are specified by W3C are as given respectively. Types is a container for data type definitions using schema definitions or xsd file imports. Message is the definition of the messages which are used during the communication.

Operation is the description of a web service action which is supported by the web service. Porttype is an abstract set of operations supported by endpoint(s). Binding data format definition of a specific port declaration. Port is just one endpoint described with a binding definition and an address of the endpoint. Service is the collection of the related endpoints [17].

The parts that are important for the variable mapping work, are types and messages parts. The messages part of a wsdl file consists of the message types that are used during the communication of processes. A webservice's input and output messages are also defined in this section. A message consists of several parts and those parts can be all basic data formats like string, integer, bool, etc. or those parts can also reference to the types defined in the types section of a wsdl or in any xsd file imported in the types section of a wsdl. So it is obvious that in order to convert a message type to PROMELA, the type definitions that are used in the message type are also have to be converted somehow to the PROMELA language.

Firstly the BPEL source file is parsed completely and the variables are modeled. During the variable mapping to PROMELA if the variable's type is a type or message defined in the wsdl type than the mapping of the variable type is done and the variable type and variable created together. By this approach the types and messages are converted to PROMELA.

The schema types and messages are modeled in PROMELA by using the typedef declaration of PROMELA. If the variable's type is a basic type like string, integer, etc. then a simple conversion between simple web type defined by W3C is mapped to the PROMELA primitive types. An example of a message and a type declaration with a variable declaration is shown in Figure 4.3 and the PROMELA model corresponding to this variable is shown in Figure 4.4 too.

The types section definitions are mapped with a name trailed with `_type` text in order to indicate that was a type in the wsdl and messages are mapped directly with the same name specified in the wsdl file. All the complex definitions can be mapped

```

<element name = "makcay1ProcessRequest">
  <complexType>
    <sequence>
      <element name = "input" type = "int"/>
    </sequence>
  </complexType>
</element>
<message name = "makcay1RequestMessage">
  <part name = "payload" element = "client:makcay1ProcessRequest"/>
</message>

```

Figure 4.3. A chunk of types and messages section from the wsdl definition

```

typedef input_type{
  int input;
}
typedef makcay1ProcessRequest_type{
  input_type makcay1ProcessRequest;
}
typedef makcay1RequestMessage{
  makcay1ProcessRequest_type payload;
}
makcay1RequestMessage inputVariable;

```

Figure 4.4. PROMELA conversion of the wsdl chunk

to PROMELA by using typedef declarations and the important part of this mapping is that we do not lose the expression power of wsdl xml messages, since all of the xml parts are mapped into PROMELA.

After completing the mapping of wsdl and xsd types to PROMELA another big problem during the modeling of BPEL process into PROMELA will be the xpath expressions that are used in the BPEL process. An xpath expression helps the BPEL engine to query the xml data easily. Since the main purpose of this work is not to fully convert xpaths to PROMELA types, currently basic xpath expressions are allowed that can express the BPEL activity operations, which looks like `"/element1/element2/intVal"` query. According to the xpath specifications declared by W3C, there are many xpath

functions like concat,contains,etc... More than the xpath functions and special xpath extensions, a more detailed definition of xpathes are described in the W3C specifications [18].

4.4. Modeling Expressions and Conditions

As it is stated before a BPEL process's main logics and body are constructed by the BPEL activities. But if those activities are investigated more specifically, it will be seen that inside those activities there are several BPEL expressions and BPEL conditions that are specified. For example the switch activity, it was mentioned that switch consists of case conditions before and those conditions are specified as an expression to the case conditions. Those conditions and all other expressions that are defined in BPEL specifications have to be implemented and also mapped to PROMELA condition and expression mechanisms.

Regarding BPEL expressions, there are expressions that are mandatory for BPEL implementations like bpws:getVariableData() function like expressions. In order to implement those kind of BPEL expressions and conditions in PROMELA, a general parser framework has to be introduced. For this purpose a BPEL expression parser structure has been created within this work. LEX and CUP implementation of LR parsers are used for this purpose. In order to use those languages, the language grammar of the BPEL expressions has to be stated clearly. Also this mechanism will ease adding a new expression to the current implementation easily. The following sections describe the LEX and CUP implementation and grammar specification of the BPEL expressions in detail.

4.4.1. Lex Model

Lex is a program which functions like a Lexical Analyzer. It's a program generator designed for lexical processing of character input streams [19]. It reads a character stream and search user specified regular expressions on that stream and produce matching outputs. Lex can be used in order to scan any type of expressions in order to

generate a new expression or parse. For our BPEL expression scanning task, lex rules and regular expressions are used and the symbols that are used in BPEL expressions are specified in a lex file and scanned from the expression supplied.

The implementation of lex into java is the purpose of the project called jlex [20]. Jlex reads a lex file as input and generates a java source code corresponding to the given lex file. This produced java file can be used as scanner for an input stream and passed to a parser in order to parse input coming from a stream and generate a new representation which is our main purpose for converting BPEL expressions. That's why JLEX and CUP will be used together, jlex will read the tokens and symbols and pass the data to the CUP and it will parse and return the parsed data to the caller of the parser.

Since lex is a scanner for the data coming from a stream, we have to write the regular expressions of our expressions into a lex file. That will be used to scan the symbols in a BPEL expression. Below is the jlex symbol definitions and scanner code is shown with explanations.

```
package bpelExpression;
import java.io.*;
import java_cup.runtime.*;

%%
%line
%char
%state COMMENTS
%cup
```

Figure 4.5. Jlex directives

The directives shown in Figure 4.5 are the jlex directives that tells jlex to keep track of line number, which character of a line, a COMMENTS state like the predefined YYINITIAL state. From YYINITIAL state, this automata can be directed to COMMENTS state and in COMMENTS state expression comments can be parsed. The cup directive enables JLEX the CUP compatibility mode that we will use after

getting the symbols.

```
%{
private ComplexSymbolFactory symbolFactory=new ComplexSymbolFactory();
public Symbol symbol( String name,int tokenType ) {
    return symbolFactory.newSymbol(name, tokenType,ytext());
}
public ComplexSymbolFactory getSymbolFactory(){
    return symbolFactory;
}
%}
```

Figure 4.6. Jlex symbol functions

The source code shown in Figure 4.6 is the java method that will be placed in the produced java class which is used below. We used the complexSymbolFactory which is defined in CUP library and used our own symbol definitions by the help of those methods.

```
%eofval{
    return symbolFactory.newSymbol("EOF",sym.EOF);
}%eofval}
```

Figure 4.7. Jlex end of file symbol

When the end of file symbol is matched, the return of the lex class will be predefined EOF symbol as shown in Figure 4.7.

The definitions shown in Figure 4.8 are the regular expression definitions that can be used in a BPEL expression.

The definitions shown in Figure 4.9 are the jlex matching rules defined with the actions. The dot sign which is the last symbol means any other token that are received except those ones defined. Here in any other token received we tell the jlex to return error, so that any symbol which is forgotten in this implementation can be added easily.

```

ALPHA=[A-Za-z_]
DIGIT=[0-9]
ALPHA_NUMERIC=ALPHA|DIGIT
IDENT=ALPHA(ALPHA_NUMERIC)*
NUMBER=(DIGIT)+
WHITE_SPACE=(|\ \n\r\t\f|)+
CONSTSTRING = (\[^\"'*\')|(\^[\'']*\\)

```

Figure 4.8. Jlex definitions

```

<YYINITIAL> "<|"&lt;" {
    return symbol("LT",sym.LT);
}
<YYINITIAL> ">|"&gt;" {
    return symbol("GT",sym.GT);
}
<YYINITIAL> "<="|"&lt;=" {
    return symbol("LE",sym.LE);
}
<YYINITIAL> ">="|"&gt;=" {
    return symbol("GE",sym.GE);
}
<YYINITIAL> "=" {
    return symbol("EQUAL",sym.EQUAL);
}
...
<YYINITIAL> "bpws:getVariableData" {
    return symbol("bpelgetVar",sym.bpelgetVar);
}
<YYINITIAL> {NUMBER} {
    return symbol("NUMBER",sym.NUMBER);
}
<YYINITIAL> {CONSTSTRING} {
    return symbol("LITERAL",sym.LITERAL);
}
<YYINITIAL> {WHITE_SPACE} { }
<YYINITIAL> . {
    return symbol("error",sym.error);
}

```

Figure 4.9. Chunk of jlex rules

The definition of a rule is not very complex. YYINITIAL is the initial state of the jlex. For example `<YYINITIAL> {WHITE_SPACE} { }` means if in the initial state and white space regular expression is received then do nothing. Also new states can be defined as mentioned before like COMMENTS or any other states.

Jlex is responsible of converting those lexical rules into a java source code which will be used by CUP in order to parse out expressions. Instead of any other hardcoded programming technique, this kind of approach will make the parsing job of BPEL expressions easier to develop. When a new BPEL function or expression is wanted to be added in this project, just adding a new lexical rule will be enough for the lex scanning part.

So far scanning of an expression and finding the appropriate symbols in order to understand what we have is done by using lex. Now the parsing jobs of the found symbols have to be implemented. For this purpose the returned symbols from lex rules defined above has to be captured and parsed somehow. In order to accomplish this task a CUP parser is used in order to keep the general framework maintainable.

4.4.2. CUP Model

CUP is the synonym for Constructor of Useful Parsers. CUP is the system that is used to generate LALR parsers from specified grammars [21]. Basicly it has the same usage with yacc that unix user's are familiar with, but CUP is the java implementation of the same usage. CUP reads a file which is CUP specification of a grammar and then it generated two files. One of those files is the symbols file and the other is the parser class that will do the parsing job.

In order to use CUP parser for BPEL expressions, the LR grammar of the BPEL expressions has to be expressed. First of all terminals and non-terminals of the grammar have to be found. Then the grammar can be implemented with the help of those terminals and non-terminals. Inside the CUP specification file we also have to point to the lexer class generated by lex with the lexical definitions explained above.

For all of those required data we have to create a CUP file and use that file in order to generate the parser and symbols classes. The CUP file used to generate the BPEL expression parser that has the grammar definitions and expressions in it is shown below with the detailed explanations.

```
package bpelExpression;

import java.lang.*;
import java.util.*;
import java_cup.runtime.*;
import java.io.*;
import bpelHandler.*;
action code{
    Hashtable table = new Hashtable();
:};
```

Figure 4.10. CUP action code

The package name and java imports have to be specified for the resulting java files. The user action code part shown in Figure 4.10 is to include code within the CUP\$actions class which is used by the code embedded in the grammar for resolving the appropriate action to perform.

```
parser code{
    private Yylex lexer;
    private InputStream inputst;
    public parser(InputStream instream){
        lexer=new Yylex(instream);
        symbolFactory=lexer.getSymbolFactory();
        inputst=instream;
    }
:};
```

Figure 4.11. CUP parser code

Parser code that is shown in Figure 4.11 is like the action code, but parser code is the user code supplied in order to be placed inside the parser class. Above the Yylex class is instantiated. Yylex is the class that the lex produced in order to return

symbols to the CUP. In the constructor of parser lex class is initialized (lexer) and the symbolFactory is initialized by the returning symbol factory that was formed by the lex implementation before. By this symbol factory implementation we can use our own defined symbols.

```
scan with{  
    return lexer.next_token();  
};
```

Figure 4.12. CUP scan with section

Scan with section shows how the parser should be asking for the next token from the scanner. The return type of the code found in the scan with function should be of type `java_cup.runtime.token` type. As it can be seen in Figure 4.12, we use `lexer` as the token generator for our CUP implementation. It will ask the lexical analyzer implemented before to get the tokens.

```
terminal LPAREN,RPAREN,COMMA;  
terminal ADD,MINUS,MOD,TIMES,DIVIDE;  
terminal NUMBER,LITERAL;  
terminal LT,GT,LE,GE,EQUAL,NEQUAL,AND,OR;  
terminal bpelgetVar;  
  
non terminal expr;
```

Figure 4.13. CUP terminal and non terminal symbols

The terminal symbols in our LR grammar specified in the CUP file specification are shown in Figure 4.13. All the symbols that are used in the lex implementation are terminals for us. Those are the end states and non-terminal expressions are about to be reduced to terminals by using the grammar specified. If the grammar is correct all the BPEL expressions should be converted to terminals according to the grammar rules.

Precedences part specified the priority between the symbols used in the grammar.

```

/* Precedences */
precedence left LT,GT,LE,GE,EQUAL,NEQUAL,AND,OR,ADD,MINUS;
precedence left TIMES,DIVIDE,MOD;

start with expr;

```

Figure 4.14. CUP priorities

There are three types of precedence which are left, right and noassoc. Since we have left to right priorities only precedence left is used for all of the symbols defined in Figure 4.14. The start point of the grammar also be thought as the goal of the grammar. Because at the end of parsing process the result will be returned to the caller which is obtained by applying the grammar rules to the given BPEL expression starting from the start point of the rules which is shown in Figure 4.14.

The grammar rules part of the CUP specifications is the main part and the most important part for the parsing process which is shown in Figure 4.15. For example `bpws:getVariableData('counter')` expression will be parsed by the grammar above as follows. Firstly the lex scanner will scan and find `bpws:getVariableData` text and create a symbol and pass that symbol to the CUP. This symbol is `bpelgetVar` symbol. Then the rule `expr::=bpelgetVar LPAREN LITERAL:l1 RPAREN` will be applied for this expression, which is returning the variable indicated in its action handler. `RESULT` is returned from those expression handlers to the caller of the parser.

By using that LEX and CUP structures together, adding of a new expression to the system is very easy which consists of adding a new line to LEX file and CUP file with the specific handler. The reason why we used lex is more obvious after looking into that grammar section. The character matching part is the job of lex's part and the grammar rule matching and the parsing of the contents of the given expressions is the CUP grammar expression parser's job.

```

expr::= expr:e1 ADD expr:e2
  { : RESULT = e1.toString() + "+" + e2.toString(); :}
  |
  expr:e1 MINUS expr:e2
  { : RESULT = e1.toString() + "-" + e2.toString(); :}
  |
  ...
  |
  NUMBER:n
  { : RESULT = n; :}
  |
  LPAREN expr:e RPAREN
  { : RESULT = "(" + e.toString() + "; :}
  |
  bpelgetVar LPAREN LITERAL:l1 COMMA LITERAL:l2 COMMA LITERAL:l3 RPAREN
  { :   RESULT= activityHandler.varPartQuery2Promela(utilities.removeQuotes(l1.toString()),
utilities.removeQuotes(l2.toString()), utilities.removeQuotes(l3.toString())); :}
  |
  bpelgetVar LPAREN LITERAL:l1 COMMA LITERAL:l2 RPAREN
  { : RESULT= activityHandler.varPartQuery2Promela(utilities.removeQuotes(l1.toString()), null,
utilities.removeQuotes(l2.toString())); :}
  |
  bpelgetVar LPAREN LITERAL:l1 RPAREN
  { : RESULT=utilities.removeQuotes(l1.toString()); :}
  |
  LITERAL:l1
  { : RESULT=l1.toString(); :}
  ;

```

Figure 4.15. CUP grammar rules

4.5. Modeling Activities in PROMELA

4.5.1. Assign

Assign activity in BPEL is consisting of copy rules inside the assign activity. Inside the activity there may be unlimited copy operations, those copy operations all correspond to the variable assignment statements in PROMELA. The tough part for the mapping is the variable, message part and xpath queries mapping. Also, instead of a copy operation from a variable, an expression or an output of a function might be

copied to the destination.

```
<assign name = "Assign_1">
  <copy>
    <from variable = "inputVariable" part = "payload" query = "/client:makcay1ProcessRequest/client:input"/>
    <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
  </copy>
  <copy>
    <from expression = "5"/>
    <to variable = "counter"/>
  </copy>
  <copy>
    <from expression = "bpws:getVariableData('counter')+1"/>
    <to variable = "counter"/>
  </copy>
</assign>
```

Figure 4.16. Example assign activity

```
outputVariable.payload.makcay1ProcessResponse.result = inputVariable.payload.makcay1ProcessRequest.input;
counter=5;
counter=counter+1;
```

Figure 4.17. PROMELA conversion of the assign example in Figure 4.16

During the conversion the copy expressions are extracted and the mapping process is applied for each. And the variable part is the determination part of the activity if the variable part of the from part is empty then this is treated as expression and expression parser which was implemented before is used, otherwise the variable mapping is used.

4.5.2. Sequence

Sequence activity is the activity that keeps a sequence of activities inside. In PROMELA there is not an activity for a sequence itself. Mapping a sequence will mean mapping the activities inside that sequence box to the PROMELA language sequentially. A sequence might have any type of activity inside, so handling a sequence means handling all possible activities.


```

<sequence name = "Sequence_3">
  <assign name = "Assign_3">
    <copy>
      <from expression = "5"/>
      <to variable = "counter"/>
    </copy>
  </assign>
  <sequence name = "Sequence_4">
    <terminate name = "Terminate_2"/>
    <empty name = "Empty_6"/>
  </sequence>
</sequence>

```

Figure 4.18. Example sequence activity

```

counter=5;
goto end;
/*empty activity*/
skip;
...
end:skip;

```

Figure 4.19. PROMELA conversion of the sequence in Figure 4.18

In the sequence activity in Figure 4.18 as it is mentioned before, all of the activities are translated into PROMELA language sequentially. The important part is that inside a sequence box there may be also sequence boxes recursively. Those activities should also be handled and converted sequentially as it is shown in Figure 4.19.

4.5.3. Switch

Switch activity of BPEL is the activity which is used for decision making using the conditions supplied to switch activity. Switch activity consists of case conditions and each case block is consisting of one or more activities that are going to be executed when that case is satisfied. The condition string of each case block is a BPEL condition expression which has to be parsed using BPEL expression parser created before.

The switch activity directly corresponds to the if conditional branching in PROMELA. Also each switch has an optional otherwise implementation which is executed if none of the case conditions are satisfied in a switch activity. Otherwise branch of a switch activity is a directly corresponding to the else branch of conditional if branches in PROMELA. In Figure 4.20 and 4.21 an example of a switch and its translation to PROMELA is shown.

```

<switch name = "Switch_1">
  <case condition = "bpws:getVariableData('counter')=5">
    <sequence name = "Sequence_5">
      <assign name = "Assign_4">
        <copy>
          <from expression = "5"/>
          <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
        </copy>
      </assign>
      <empty name = "Empty_2"/>
    </sequence>
  </case>
  <case condition = "bpws:getVariableData('counter')=1">
    <sequence name = "Sequence_6">
      <assign name = "Assign_5">
        <copy>
          <from expression = "2"/>
          <to variable = "counter"/>
        </copy>
      </assign>
      <empty name = "Empty_1"/>
    </sequence>
  </case>
  <otherwise>
    <empty name = "Empty_3"/>
  </otherwise>
</switch>

```

Figure 4.20. Example switch activity

As it is seen in Figure 4.21, switch, case and otherwise activities are directly mapping to the if structure in PROMELA with the expression parsing and else implementation. Even each case block might have a sequence or a scope declaration inside, all of those activities will be mapped to PROMELA as shown in Figure 4.21.

```

if
:: counter==5 -> outputVariable.payload.makcay1ProcessResponse.result=5;
/*empty activity*/
skip;
::else->
if
:: counter==1 -> counter=2;
/*empty activity*/
skip;
::else->
/*empty activity*/
skip;
fi;
fi;

```

Figure 4.21. PROMELA translation of the switch activity in Figure 4.20

4.5.4. While

While is the loop activity in BPEL that gets a condition expression as an attribute and loops while that condition holds. Each time this expression is reevaluated and when the result is false the loop is broken. The behavior of while activity is not unexpected according to the other structural programming languages. It is similar to the while concept in any other programming language.

```

<while name = "While_1" condition = "bpws:getVariableData('counter')<10">
  <sequence name = "Sequence_1">
    <assign name = "Assign_2">
      <copy>
        <from expression = "bpws:getVariableData('counter')+1"/>
        <to variable = "counter"/>
      </copy>
    </assign>
    <empty name = "Empty_4"/>
  </sequence>
</while>

```

Figure 4.22. While activity example

Note that in PROMELA we don't have while or for loops. Instead there is a

```

do
  :: counter<10 -> counter=counter+1;
  /*empty activity*/
  skip;
  :: else -> break;
od;

```

Figure 4.23. PROMELA translation of the while activity in Figure 4.22

generic loop mechanism which can be used as for or while like it is in other structural programming languages. The generic do loop is getting the guard statements and the action statements for implementing conditional branching. For while we use the break directive of PROMELA when the condition of while is not holding anymore.

4.5.5. Scope

Scope activity is the activity that enables us to create an inner scope inside a BPEL process. Most of the programming languages have local scopes whose variables and declarations are only valid within that scope. BPEL scope activity is also similar to creating a local scope. A scope might have its own variables, exception handlers, compensation handlers, etc. which actually makes scope a small process.

The obvious problem of mapping scopes to PROMELA is the translation of local variables to PROMELA. Because the variables of the process had been converted before, now we have to append new declarations and create local variables of those type definitions if they have to be done. For the variables of scope the necessary typedef definitions are done globally, since PROMELA allows only global type definitions. In BPEL those variable types are globally defined in wsdl messages too. So the mapping of variable types and variable instances are mapped directly as it is in BPEL too. An example scope activity and its translation is shown in Figures 4.24 and 4.25.

Another problem of scope is that it has its own fault handlers. Since fault handlers are appended at the end of processes as fault handler labels and appropriate goto state-

ment is created during the exception throwing model in PROMELA, same procedure have to be repeated for this inner scope in PROMELA too. And in case of unhandled exception coming from an activity inside the scope, this exception has to be thrown again, for the upper faulthandler catch blocks. If there is no upper faulthandler for this error this exception has to be put in the unhandled exception channel as it is explained in the modeling of throw activity.

```
<scope name = "Scope_1">
  <variables>
    <variable name = "scopeVar1" messageType = "client:makcay1RequestMessage"/>
    <variable name = "scopeVar2" messageType = "client:makcay1ResponseMessage"/>
    <variable name = "scopeVar3" type = "xsd:int"/>
  </variables>
  <faultHandlers>
    <catch faultName = "bpws:selectionFailure">
      <empty name = "Empty_15"/>
    </catch>
    <catch faultName = "bpws:conflictingReceive">
      <sequence name = "Sequence_10">
        <empty name = "Empty_16"/>
        <throw name = "Throw_2" faultName = "bpws:joinFailure"/>
      </sequence>
    </catch>
  </faultHandlers>
  <sequence name = "Sequence_2">
    <terminate name = "Terminate_1"/>
    <empty name = "Empty_5"/>
  </sequence>
</scope>
```

Figure 4.24. Example BPEL scope activity

Note that in PROMELA there is no local scopes, in PROMELA there are two variable scopes which are global scope and process local scope [22]. So actually in the produced code above inside a process we create a scope with braces, but those are not meaningful at all. But those braces and comments increase the readability of the generated code which also means those variables defined above might also be called outside the braces in this scope translation within the process scope of this code part.

Another challenging part during the implementation of scope activity is the com-

```

makcay1RequestMessage scopeVar1;
makcay1ResponseMessage scopeVar2;
int scopeVar3;
Scope_1_scopeSnapshot_context Scope_1_scopeSnapshot_var;
Scope_2_scopeSnapshot_context Scope_2_scopeSnapshot_var;
...
...
/*scope name=Scope_1*/
{
    goto faultHandler_conflictingReceive;
    goto end_makcay1;
    skip;
    goto end_Scope_1;

    faultHandler_selectionFailure_2:
        skip;
        goto end_Scope_1;
    faultHandler_conflictingReceive:
        skip;
        goto faultHandler_joinFailure;
        goto end_Scope_1;
    end_Scope_1:skip;
    /*Scope Snapshot will be saved to channel*/
    ch_makcay1_processContext??<eval(_pid),procContext>;
    Scope_1_scopeSnapshot_var.payload.makcay1ProcessRequest.input=scopeVar1.payload.makcay1ProcessRequest
.input;
    Scope_1_scopeSnapshot_var.payload.makcay1ProcessResponse.result=scopeVar2.payload.makcay1ProcessResponse
.result;
    Scope_1_scopeSnapshot_var.scopeVar3=scopeVar3.scopeVar3;
    atomic{
        ch_Scope_1_scopeSnapshot!!ch_Scope_1_scopeSnapshot_session,Scope_1_scopeSnapshot_var;
        ch_Scope_1_scopeSnapshot_session=ch_Scope_1_scopeSnapshot_session-1;
    }
}

```

Figure 4.25. PROMELA translation of the scope activity above

pensation handling of a scope. Implementation of compensation handlers are explained in the compensation handler part. Another important part of the compensation mechanism of a scope activity is the term explained in [23] and referred as "scope snapshot". Scope snapshot is the snapshot of the successfully completed scope. During the compensation those scope snapshots are retrieved. Inside the scope snapshots the variables of the scope will be saved and the values of the structures inside the scope are used

while compensating the scope. Note that if a scope is placed inside a while activity, then for each iteration of the loop, there has to be one scope snapshot and each scope iteration has to be compensated separately while compensating this scope.

4.5.6. Terminate

Terminate is one of the simplest BPEL activities that can be used in a process. It looks like the exit routine of any programming language. But the problem of PROMELA translation for terminate is that we don't actually have a terminate directive in PROMELA. That's why another workaround solution for terminate activity mapping is introduced. We can use goto and labels in order to terminate a process. An example of terminate activity and its translation is shown in Figures 4.26 and 4.27.

```
<sequence name = "Sequence_4">
  <terminate name = "Terminate_2"/>
  <empty name = "Empty_6"/>
</sequence>
```

Figure 4.26. Example terminate BPEL activity

```
goto end;
/*empty activity*/
skip;
...
end:skip;
}
```

Figure 4.27. PROMELA translation of the terminate activity in Figure 4.26

When the code reaches to the terminate activity's translation which is goto directive of PROMELA, the process will branch to the label named end and end is the label that is placed at the end of the automatic PROMELA process code generation step. This workaround solution will accomplish the same task of terminate activity.

4.5.7. Flow

Flow is one of the most complex activities in BPEL. Flow activity enables parallel processing in a BPEL process. A flow activity consists of multiple child activities that will be run in parallel during the execution of the flow. The biggest problem of flow activity is the links of the flow activity. Flow enables not only parallel execution, but also control dependencies of the parallel executing activities. The links created in a flow might be used within the activities inside the flow in order to determine the dependencies. An activity might refer to the links defined in a flow as a source or as a target. If an activity A is the source of a link L, and another activity B is the target of link L, then the link L creates a dependency between A and B.

The evaluation of links that are references as sources inside the activities should be done with the rules defined in the BPEL specifications [23]. When an activity A is completed without any exceptions, the link statuses of all the links that are defined in activity A as source (outgoing link statuses) should be evaluated. The status of a link might be true, false or unset. The status of the link will be evaluated using the evaluation of the boolean expression defined in the source definition of the activity. If the transition condition is empty then the link status is set to true by default.

The link status is evaluated for the use of the activity having the target definition for the same link. If an activity B has the target definition for a link, before executing the activity it has to be joined with the link. In order to do the joining of the activity with the links, all the links that defined in activity B as target has to be evaluated which means all the link statuses have to be true or false (not unset). If the evaluation of all the links that are defined as target in activity B are completed, then the join condition of the activity B has to be evaluated. If the join condition evaluates to true then the activity B is executed.

If the join condition is not defined in an activity the default join condition for an activity is the logical or of all the link statuses coming to the activity (defined as target). This indicates that if any of the link statuses is set to true then the activity is

able to execute. If the join condition evaluates to false, then a joinFailure exception is thrown. Another important part of an activity is the suppress join failure flag of the flow activity. If suppress join failure is set to yes then the join failure is now thrown instead the activity B is silently skipped by evaluating all of its source links to false which is for dead path elimination. If suppress join failure is not defined in the activity then this value is inherited from the parent activities (process definition at the top). The default value of the suppress join failure flag is set to no.

In order to create the dependency checks for each activity inside the flow, a new procedure is created which makes the parallel execution possible. All those procedures have to be terminated in case of any exception, the unhandled exception channels created for unhandled BPEL faults are used again here. Each PROMELA procedure will have unless statement which causes the PROMELA process to stop if the condition inside unless becomes true. In the main process which originated the parallel processes those parallel processes have to be waited to end, since flow activity ends when all of its parallel branches are over. In order to do this, the created processes pids are polled in the main process and when the processes over the unhandled exception channels are checked if there is an exception. If any exception occurs the exception is thrown from the flow activity.

The pid values of parent and the child are used in order to make sure that an exception is coming from the process what we are watching. Otherwise the same pid might be used in some other process after its execution, but when its parent still alive and asking for the messages related to its id there will be no misleading messages by combining both pid and parentid values. That is why in this implementation all the processes get a parent pid as an input and then the caller checks if it threw any exception during this pid and parent pid execution.

4.5.8. Throw

Throw is the activity in order to throw an exception occurred during the execution of BPEL process. A throw activity has one required parameter which is the fault

```

<flow name = "Flow_links">
  <links>
    <link name = "seq1"/>
  </links>
  <sequence name = "Sequence_7">
    <empty name = "empty1"/>
    <empty name = "empty2">
      <source linkName = "seq1" transitionCondition = "1=1"/>
    </empty>
    <throw name = "Throw_3" faultName = "bpws:selectionFailure"/>
  </sequence>
  <sequence name = "Sequence_7">
    <empty name = "empty3" joinCondition = "2=2">
      <target linkName = "seq1"/>
    </empty>
    <empty name = "empty4"/>
  </sequence>
  <sequence>
    <empty>
      <source linkName = "seq1"/>
    </empty>
  </sequence>
</flow>

```

Figure 4.28. Example flow activity

name to throw. In BPEL all faults are identified by fault names that are consisting of namespace and the name of the fault in that namespace. Namespace and name forms the fault name together that can be thrown. In order to model throw activities the fault handlers have to be modeled, since a throw results the process to branch on a fault handler of a scope and end that scope. The fault handlers are mapped with the labels that can be branched from anywhere within the PROMELA process. Since the expected behavior of a throw activity is to cause the process to branch on the appropriate fault handler or to pass the exception one higher scope, an appropriate goto statement to the appropriate fault handler is the PROMELA conversion of the throw activity. If an appropriate fault handler cannot be found, then an unhandled exception channel is created for the PROMELA process and the faultname if not exists. And that unhandled exception is pushed on the channel. When the process ends the caller will check the unhandled exception channels belonging to the called process

```

link_seq1_eval=false;
pid_3=run proc_Flow_links_Sequence_7_FlowProcess(_pid);
pid_4=run proc_Flow_links_Sequence_7_FlowProcess_2(_pid);
pid_5=run proc_Flow_links_FlowProcess(_pid);
do
::else->skip;
::(enabled(pid_3)==false && enabled(pid_4)==false && enabled(pid_5)==false)->break;
od;
if
::unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_selectionFailure??eval(pid_3),eval(_pid)->
goto faultHandler_selectionFailure;
::(empty(unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_selectionFailure))->skip;
fi;

```

Figure 4.29. PROMELA model of the flow in Figure 4.28

and unhandled exceptions can be understood. In Figure 4.32 one handled and one unhandled exception is shown.

The generated PROMELA conversions of the three exceptions in Figure 4.31 are shown. First and the third exceptions are handled since they are branching to the appropriate fault handler of a scope whose execution will end after running the fault handler. The second throw activity is an unhandled throw activity for the PROMELA process it was placed in. In order to notify that the caller an exception has occurred, the faultvariable which is inputVariable in that example is sent to the channel. The poller of this channel can handle this thrown exception or it can throw the exception too. If no process handles this exception eventually this fault will be passed to init process and init process will generate an assertion.

4.5.9. Pick

Pick activity is one of the selection activities that can change the execution path of a process. Pick activity might have two kinds of elements which are onMessage and onAlarm elements. Pick activity waits for the occurrence of the events that are specified inside the pick and it branches to the part whose message arrives first. Pick activity

```

proctype proc_Flow_links_Sequence_7_FlowProcess(int _parentId){
    int tmpSession;
    int tmpPid;
    {
        skip;
        skip;
        link_seq1=(1==1);
        link_seq1_eval=true;
        atomic{
            unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_selectionFailure!_pid,_parentId;
            goto end_Sequence_7;
        }
    }
    unless {false || unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_selectionFailure?[_eval(_parentId)]};
    end_Sequence_7:skip;
}

```

Figure 4.30. The model of the first block of the flow in Figure 4.28

```

<throw name="Throw_3" faultName="bpws:selectionFailure"/>
<throw name="Throw_5" faultName="bpws:conflictingReceive" faultVariable="inputVariable"/>
<throw name="Throw_6" faultName="bpws:joinFailure" faultVariable="inputVariable"/>

```

Figure 4.31. Example throw activity

is used for selective event processing [23]. OnMessage is actually same with a receive activity waiting on a partnerlink to get a message. onAlarm is used to create timeout or wait until a specific time. Inside a pick several onMessages and several onAlarms can be specified, but there should be at least one onMessage branch specified inside the pick activity. Pick chooses the first occurred event and branches that activity.

The problem with pick activity is the user interactivity with onAlarm timeouts, since in PROMELA there is no waiting function and it is pointless to wait a process in verification. The aim of the verification of the BPEL process is to make a full state space search and enter in all states and generate an error or assertion if it can occur in a way and to find out that way that causes the verification to fail. So we can create an undeterministic branching for onalarm events. Since PROMELA acts

```

goto faultHandler_selectionFailure;
atomic{
    tmpSession=unhandledEx_makay1_conflictingReceive_session+1;
    unhandledEx_makay1_conflictingReceive!tmpSession,inputVariable;
    goto end_makay1;
}
goto faultHandler_joinFailure;

```

Figure 4.32. PROMELA model of the throw activities in Figure 4.31

undeterministically, when it has more than one true statement inside an if, it selects one of them and jumps one of them. And since timeout can occur at any time we can create a true condition jump for onAlarm events. SPIN will decide undeterministically to pick onAlarm or onMessage if any message is available.

4.5.10. Empty

Empty activity is the activity that can be used to comply with BPEL standards in some cases. It has no meaning as an activity. The PROMELA translation of the empty activity is skip directive of the PROMELA. Even though empty has no meaning it can have source or target of the links inside a flow activity.

4.5.11. Wait

Wait activity is the activity that causes the BPEL process to wait for a time period or until a specific time. During the verification of a BPEL process, wait activity is not meaningful. Since SPIN verifies all possible states undeterministically, time domain and user interaction is not important for verification purposes. So, wait activity is also mapped to skip directive as it is in empty activity. Again wait can have source and target implementations too inside a flow activity.

```

<pick name = "Pick_1">
  <onMessage portType = "ns1:chargeCallback" operation = "onResult" variable =
"OnMessage_onResult_InputVariable" partnerLink = "charge">
    <assign name = "successfullInvocation">
      <copy>
        <from variable = "OnMessage_onResult_InputVariable" part = "payload" query =
"/ns1:chargeProcessResponse/ns1:result"/>
        <to variable = "outputVariable" part = "payload" query = "/client:serviceProcessResponse/client:result"/>
      </copy>
    </assign>
  </onMessage>
  <onAlarm for = "'P1DT30S'">
    <assign name = "invocationTimeout">
      <copy>
        <from expression = "1"/>
        <to variable = "outputVariable" part = "payload" query = "/client:serviceProcessResponse/client:result"/>
      </copy>
    </assign>
  </onAlarm>
</pick>

```

Figure 4.33. Example pick activity

4.5.12. Invoke

Invoke activity is the way of communication of a BPEL process with the outer BPEL processes. In BPEL engine implementations, there are lots of platforms that an invoke can communicate. In our implementation we assumed a BPEL process can invoke another BPEL process through message channels, since message channels are the best way of communicating processes in PROMELA. Inside an invoke activity the partnerlinktype, partnerlink roles and operation parameters are used in order to find the appropriate communication channel. With the same parameters a receive activity will be waiting for a message on the same channel. Another important part of the invoke activity is that invoke activity might have its own fault catch blocks. In case of an exception coming from the remote process, those faults can be handled in those handlers. An example implementation of the invoke activity is shown in Figures 4.35 and 4.36.

```

do
  /*onMessage*/
  ::ch_service_processContext??eval(_pid),procContext->
    ch_service_processContext!_pid,procContext;
  if
    ::ch_charge_chargeRequester_onResult_IN ?
  charge_onResult_session,procContext.OnMessage_onResult_InputVariable->
    atomic{
      ch_service_processContext??eval(_pid),procContext;
      procContext.outputVariable.payload.serviceProcessResponse.result =
procContext.OnMessage_onResult_InputVariable.payload.chargeProcessResponse.result;
      ch_service_processContext!_pid,procContext;
    }
    break;
    ::true->skip;
  fi;
  /*onAlarm*/
  ::true->
    atomic{
      ch_service_processContext??eval(_pid),procContext;
      procContext.outputVariable.payload.serviceProcessResponse.result=1;
      ch_service_processContext!_pid,procContext;
    }
    break;
  /*waiting behaviour*/
  ::true->skip;
od;

```

Figure 4.34. PROMELA model of the pick activity in Figure 4.33

Note that invocation is not a one way operation on synchronous web service invocations, since it also has an outputvariable parameter which is used to receive the reply from the partner and copy it in the output variable specified. The problem of those sending and receiving messages is that how can we be sure that the reply we get from the reply channel of the partner process is the reply to our request. In BPEL engine implementations this part is handled by TCP/IP's network layers automatically. A connection is established and the received reply from the partner process is sent to the socket to the calling process. But in PROMELA we have to solve this problem by introducing sessionIds for all the communication channels. When sending a data a unique sessionId has to be generated with the help of a global integer variable, and the partner process's reply queue is polled with that sessionId. Also the partner process

```

...
<partnerLink name="makcay2" partnerRole="makcay2Provider" partnerLinkType="ns1:makcay2"/>
...

<invoke name = "Invoke_1" partnerLink = "makcay2" portType = "ns1:makcay2" operation = "process"
inputVariable = "Invoke_1_process_InputVariable" outputVariable = "Invoke_1_process_OutputVariable">
  <correlations>
    <correlation initiate = "yes" set = "CorrelationSet_1" pattern = "out-in"/>
  </correlations>
</invoke>

```

Figure 4.35. Example invoke activity with its partnerlink definition

must return with the same sessionId, that it received at the beginning of the process. So, the partner process should have a temporary variable to hold the sessionId received and send the same sessionId back to the caller at the end. The synchronous invocation is also a blocking action it waits till the partner replies. In order to map this behavior in PROMELA, `enabled()` function is used. `Enabled` checks the pid specified in it and it returns false when the process specified ends. With this implementation the caller process will wait till the remote process ends, then it will check the exception channels and reply channel.

4.5.13. Receive

Receive activity is the blocking activity for communicating through a partnerlink, which causes the process wait till a message arrives. Receive activity waits the message from the specified partnerlink, myrole definition and the operation specified. In PROMELA there is not a blocking receive, if a message channel is empty then it returns timeout error. For an invoked process we know the message will show in the channel eventually. So blocking message receiving is done by a do loop as shown in Figure 4.38 in the example. The same channel structure that is used in invoke activity is used now for the retrieving of the messages that are put by the appropriate invoke activities. Another problem with invocation and receive activities that is handled in BPEL is the correlation sets in BPEL. Correlation sets are used in order to distinguish the messages that are passing from the same partnerlink. In order to solve this


```

makcay2_process_CorrelationSet_1_session=ch_makcay2_makcay2Provider_process_IN_session+1;
atomic{
  ch_makcay1_processContext??eval(_pid),procContext;
  ch_makcay2_makcay2Provider_process_IN !
makcay2_process_CorrelationSet_1_session,procContext.Invoke_1_process_InputVariable;
  ch_makcay1_processContext!_pid,procContext;
}
makcay2_processContext makcay2_procContext;
atomic{
  pid_4=run makcay2(_pid);
  ch_makcay2_processContext!pid_4,makcay2_procContext;
}
do
  ::enabled(pid_4)==false->break;
  ::else->skip;
od;
atomic{
  ch_makcay1_processContext??eval(_pid),procContext;
  if
    ::ch_makcay2_makcay2Provider_process_OUT ??
eval(makcay2_process_CorrelationSet_1_session),procContext.Invoke_1_process_OutputVariable->skip;
  fi;
  ch_makcay1_processContext!_pid,procContext;
}

```

Figure 4.36. PROMELA translation of the invoke in Figure 4.35

problem, in the message channel implementation correlation sets are used to besides partnerlinktype, role and operation. A receive activity and the PROMELA translation of a receive activity is shown in the example in Figure 4.38.

As it can be seen in Figure 4.38, receive activity is translated into a do loop, that checks the channel till a message arrives, otherwise if the channel is empty it returns back to the loop with a skip directive not to cause a timeout error in PROMELA. The important part of the receive activity is the sessionId part which is the first data in the message. This has to be saved into a variable and this variable has to be used in the reply implementation, since the caller waits the same sessionId.

```

...
<partnerLink name="client" partnerLinkType="client:makcay1" myRole="makcay1Provider"/>
...
<receive name="receiveInput" partnerLink="client" portType="client:makcay1" operation="process"
variable="inputVariable" createInstance="yes"/>
...

```

Figure 4.37. Example receive activity with its partnerlink

```

do
  ::ch_makcay1_makcay1Provider_process_IN?[_,_]->
    atomic{
      ch_makcay1_processContext??eval(_pid),procContext;
      ch_makcay1_makcay1Provider_process_IN?client_process_session,procContext.inputVariable;
      ch_makcay1_processContext!_pid,procContext;
    }
    break;
  ::empty(ch_makcay1_makcay1Provider_process_IN)->skip;
od;

```

Figure 4.38. PROMELA translation of the example receive activity

4.5.14. Reply

Reply activity is the activity to send the response of a web service invocation to its caller that is blocked waiting for a reply. After finding the appropriate message channel by using the partnerlink type, role and operation specified to reply activity, the variable specified in reply will be sent to the channel. An example reply activity and its translation in PROMELA is shown in Figures 4.39 and 4.40. Correlation sets are again considered during the implementation of reply activity.

```

...
<partnerLink name="client" partnerLinkType="client:makcay1" myRole="makcay1Provider"/>
...
<reply name="replyOutput" partnerLink="client" portType="client:makcay1" operation="process"
variable="outputVariable"/>
...

```

Figure 4.39. Example reply activity with its partnerlink

```

atomic{
  ch_makcay1_processContext??eval(_pid),procContext;
  ch_makcay1_makcay1Provider_process_OUT!client_process_session,procContext.outputVariable;
  ch_makcay1_processContext!_pid,procContext;
}

```

Figure 4.40. PROMELA translation of the example in Figure 4.39

Reply operation is rather more easier according to the receive and invoke operations that operate on the same channels. Reply just sends the data to the channel. As it was stated before, the most important part during communication implementation in this work is the sessionId implementation. The same sessionId that was received by the receive activity of the current process has to be saved and returned back to the caller in the reply activity. Above the sessionId variable received from the appropriate receive activity is sent back as it can be seen.

4.5.15. Compensate

Compensate activity is the activity that causes all inner scopes to start compensation in default order which is the reverse order of the successfully completion of the scopes as described in modeling compensation handlers part too. If a specific scope is specified to the compensate activity, then compensate activity starts compensating specified inner scope. Since compensation handlers are implemented by using inline functions, finding the appropriate compensation handler and calling it will be the translation of compensate activity. The default compensation order is calculated by finding the scopes inside a scope in reverse order and for inner scopes those compensation handlers are found recursively. After finding the scopes that are to be compensated in default order those inline functions are called respectively as shown in Figure 4.42.

Compensate activity finds the scopes that will be compensated according to the default order and makes the calls to those inline functions. Since in the example in Figure 4.41 the compensate activity was placed inside the fault handler, the compensate function calls are made inside the fault handler catch blocks at the end of the process.

```

...
<faultHandlers>
  <catchAll>
    <compensate name = "Compensate_1"/>
  </catchAll>
</faultHandlers>
<sequence name = "main">
  <receive name = "receiveInput" partnerLink = "client" portType = "client:makcay1" operation = "process"
variable = "inputVariable" createInstance = "yes"/>
  <scope name = "Scope_1">
    <variables>
      <variable name = "scopeVar1" messageType = "client:makcay1RequestMessage"/>
      <variable name = "scopeVar2" messageType = "client:makcay1ResponseMessage"/>
      <variable name = "scopeVar3" type = "xsd:int"/>
    </variables>
    <compensationHandler>
      <empty name = "Empty_2"/>
    </compensationHandler>
    <empty name = "Empty_1"/>
  </scope>
  <reply name = "replyOutput" partnerLink = "client" portType = "client:makcay1" operation = "process" variable
= "outputVariable"/>
</sequence>

```

Figure 4.41. Compensate activity example

The inline function will be invoked even if the scope that has the compensation handler hasn't been successfully completed. Since in that case, the message channel will be empty the compensation handler of the scope will be invoked but it will break the do loop. Because, after completion of each scope, snapshots of those scopes are saved to the scope snapshot channels that are retrieved in the compensation handlers. If the scope snapshot channels are empty then the compensation handler functions will break the loop. This structure provides that each compensation handler for a scope snapshot should not run more than once.

4.6. Modeling Fault Handlers In PROMELA

Fault handlers are the exceptions handlers that are very similar to the exception handling mechanism in any programming language. Fault handlers can be defined for

```

...
goto end_makcay1;

faultHandler:
  Scope_1_compHandler();
  goto end_makcay1;
end_makcay1:skip;
}

```

Figure 4.42. PROMELA translation of the example in Figure 4.41

a BPEL process, scope activity and invoke activity. If an exception cannot be handled inside a scope then scope terminates and the exception is passed to one higher level to be handled. Eventually if the process's fault handlers cannot handle the fault, then an unhandled exception occurs and the process terminates with a fault and an optional fault variable. Fault handlers section of a scope consists of catch blocks in order to catch a specific fault, and if none of the catch blocks works for a fault there is a catchall block which is designed to run in that case. The invoke activity also has catch and catchall blocks which is specifically for invocation faults. Since fault handlers intended to run in exceptional cases and after the execution of the fault handler the scope terminates, this process can be accomplished by branching to a fault handler at the end of the scope and then branch again the end of the scope. By applying this rule to all fault handlers, any fault handler section can be implemented at the end of each scope in a BPEL process.

```

<faultHandlers>
  <catch faultName = "bpws:selectionFailure">
    <empty name = "Empty_15"/>
  </catch>
  <catch faultName = "bpws:conflictingReceive">
    <sequence name = "Sequence_10">
      <empty name = "Empty_16"/>
      <throw name = "Throw_2" faultName = "bpws:joinFailure"/>
    </sequence>
  </catch>
</faultHandlers>

```

Figure 4.43. Example fault handlers section

```

goto end_Scope_1;

faultHandler_selectionFailure_2:
    skip;
    goto end_Scope_1;
faultHandler_conflictingReceive:
    skip;
    goto faultHandler_joinFailure;
    goto end_Scope_1;
end_Scope_1:skip;

```

Figure 4.44. PROMELA model of the fault handlers in Figure 4.43

Note that if the code doesn't branch to any fault handler in front of the fault handlers there is a branching statement to the end of the scope for successful completion of the scope. Otherwise the scope ends after executing the appropriate fault handler.

4.7. Modeling Compensation Handlers In PROMELA

Compensation handlers are the handlers that can be used in order to rollback a successfully completed scope. If a compensation handler is not defined for a scope, then a default compensation handler is used which has a compensate activity inside it. Compensation handlers can be invoked from another compensation handler or a fault handler with compensate activity. If there are several successfully completed scopes in a process, they are compensated in the default compensation order defined in [23]. The default order is the reverse order of the successfully completed scopes, which requires a LIFO implementation. In order to create the LIFO behaviour the compensation of the inner scopes are done from end to start and for each scope the last inserted scope snapshot is fetched first by the help of sorted message channels in PROMELA.

In order to implement scope snapshots, scope context type definitions and variables are created as stated in the implementation of scope. In Figure 4.45 a scope snapshot context, message channel and the sessionid variable in order to create LIFO behaviour is shown. At the successful end of each compensation the variables of the scope will be saved to the scope snapshot channel with the session id. After each saving process the

session variable will be decreased by one, so that the sorting of the message channels causes a LIFO order. A scope snapshot saving example is shown in Figure 4.25.

```
typedef Scope_1_scopeSnapshot_context{
    makcay1RequestMessage scopeVar1;
    makcay1ResponseMessage scopeVar2;
    int scopeVar3;
}

int ch_Scope_1_scopeSnapshot_session=1000;
chan ch_Scope_1_scopeSnapshot=[10] of int,Scope_1_scopeSnapshot_context;
```

Figure 4.45. PROMELA definitions of a scope snapshot

The variables inside a scope are included inside the scope context. With this data structure, compensation of a scope inside a loop is possible. If a scope is inside a loop, all of the running instances of the scope has to be compensated with reverse order too. Compensation of a scope can be thought as the continuation of the scope and rolling back the activities completed inside the scope. For this purpose compensation handlers can be implemented by using inline function implementation in PROMELA. Since inline functions are added to the calling point, the continuation of the scope can be done with the help of inline functions. The important part here is the starting of those inline functions have to be polling the scope snapshot channels and assign those to the current working process context, no matter how many times any scope has been completed, it will compensate all of them in the order they are placed in the message channel. In Figures 4.46 and 4.47 an example compensation handler and its PROMELA translation is shown.

4.8. Verification Process

After generating the model of BPEL processes, SPIN can be used to simulate the model and create a verifier to search the model for verification as discussed before. Another tool that can be used during the simulation and verification process is XSPIN which runs independently from SPIN. Briefly XSPIN generated the appropriate SPIN commands in order to run SPIN and then passes those parameters to the SPIN. So,

```

<scope name="Scope_1">
  <variables>
    <variable name="scopeVar1" messageType="client:makcay1RequestMessage"/>
    <variable name="scopeVar2" messageType="client:makcay1ResponseMessage"/>
    <variable name="scopeVar3" type="xsd:int"/>
  </variables>
  <compensationHandler>
    <empty name="Empty_2"/>
  </compensationHandler>
  <empty name="Empty_1"/>
</scope>

```

Figure 4.46. Compensation handler example

XSPIN also requires SPIN executable, an ANSI C Compiler and WISH (tcl/tk) in order to run [24].

Gui of the XSPIN is shown above, it has a built in editor and it is a very useful tool during the development of the model and the understanding of the runtime behavior of the PROMELA model by using its simulation mode. XSPIN generated sequence diagrams of the simulation modes it supports which are random, guided and interactive simulation modes as discussed previously. An example simulation of two bpel processes with an invocation timeout behavior is shown in Figure 4.49.

In the example simulation in Figure 4.49 two BPEL processes are invoked and the second process invoked two parallel executing tasks created by the flow activity inside the second BPEL process which caused the termination of the flow processes and the second process with unhandled faults. Note that if a flow activity throws an exception, all of the branches stop execution and the fault is escalated to the upper level. In this example the flow activities exception is not handled even by the process, so the second process stopped executing too. Those steps are shown above with a sequence diagram that makes the understanding of the behavior of the current PROMELA model more obvious during the development of the model.

After the simulations are done with the generated model, the behavior of the gen-


```

inline Scope_1_compHandler(){
  do
    ::ch_Scope_1_scopeSnapshot?[_ ,Scope_1_scopeSnapshot_var]->
      ch_Scope_1_scopeSnapshot?[_ ,Scope_1_scopeSnapshot_var;
        procContext.scopeVar1.payload.makcay1ProcessRequest.input=Scope_1_scopeSnapshot_var.scopeVar1.payload
        .makcay1ProcessRequest.input;

procContext.scopeVar2.payload.makcay1ProcessResponse.result=Scope_1_scopeSnapshot_var.scopeVar2.payload
.makcay1ProcessResponse.result;
      procContext.scopeVar3=Scope_1_scopeSnapshot_var.scopeVar3;
      skip;
    ::else-> break;
  od;
}

```

Figure 4.47. PROMELA translation of the compensation handler in Figure 4.46

erated PROMELA model can be seen clearly. The next step will be the verification of the created model. XSPIN also has an interface to pass to SPIN's verifier system and verify the model. A PROMELA model can be verified by assertion checks, invalid end-state checks, non-progress, acceptance cycles, never claim, unreachable code detection. The XSPIN verification parameter setting screen is shown in Figure 4.50.

The verifications options that will be searched during the verification is selected in Figure 4.50 then the verification starts with the appropriate parameters and a C source file is generated, then the code is executed and the output is shown by the tool. Note that XSPIN has also an LTL property verification interface in order to create never claim statements from LTL formulas. It is also possible to select if the never claim statement generated by LTL property manager must hold for all cases (desired behavior) or it should never hold (error behavior). LTL property manager will create the required definitions too.

In Figure 4.51 there is a verification result that is run as described above. The verification run in the figure resulted with an unreachable code check error. After searching all states in the generated model, SPIN gives unreachable code line numbers and the codes as it can be seen in Figure 4.51.

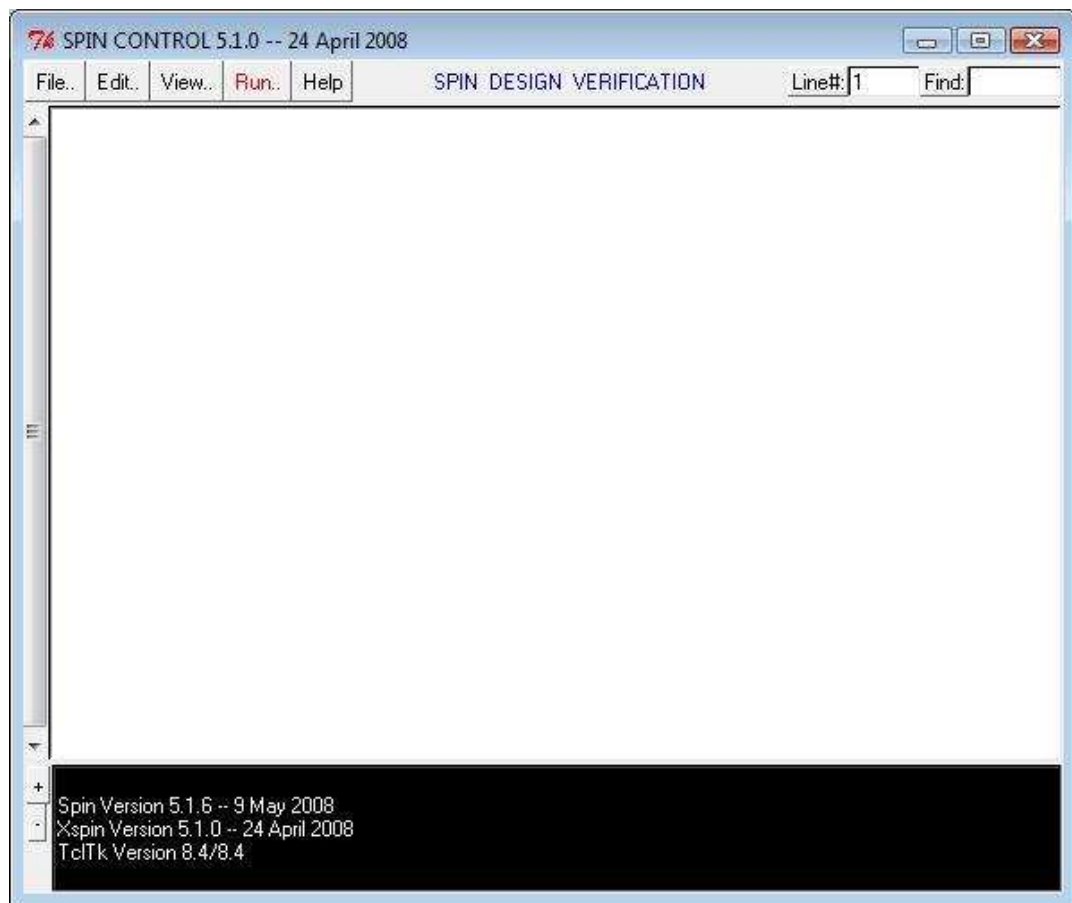


Figure 4.48. Overview of XSPIN

4.9. Example Scenario

One of the most powerful verification method of SPIN is the never claim verification based on LTL specification of the generated model. In Figure 4.52 a BPEL process example scenario is shown. Service process gets two user inputs which are credit card and charge flag as a boolean. If the charge flag is false then, the request is served without any charge, otherwise service process invokes the charge BPEL process which is an asynchronous web service. Then the result is received by using pick activity. The specification of the service process is that, if charge flag is false user should never be charged, otherwise the user has to be served and charged. If there exists any case that violate this specification, SPIN will find the counter example for this specification.

Service process is implemented with a switch activity on charge flag, and depending

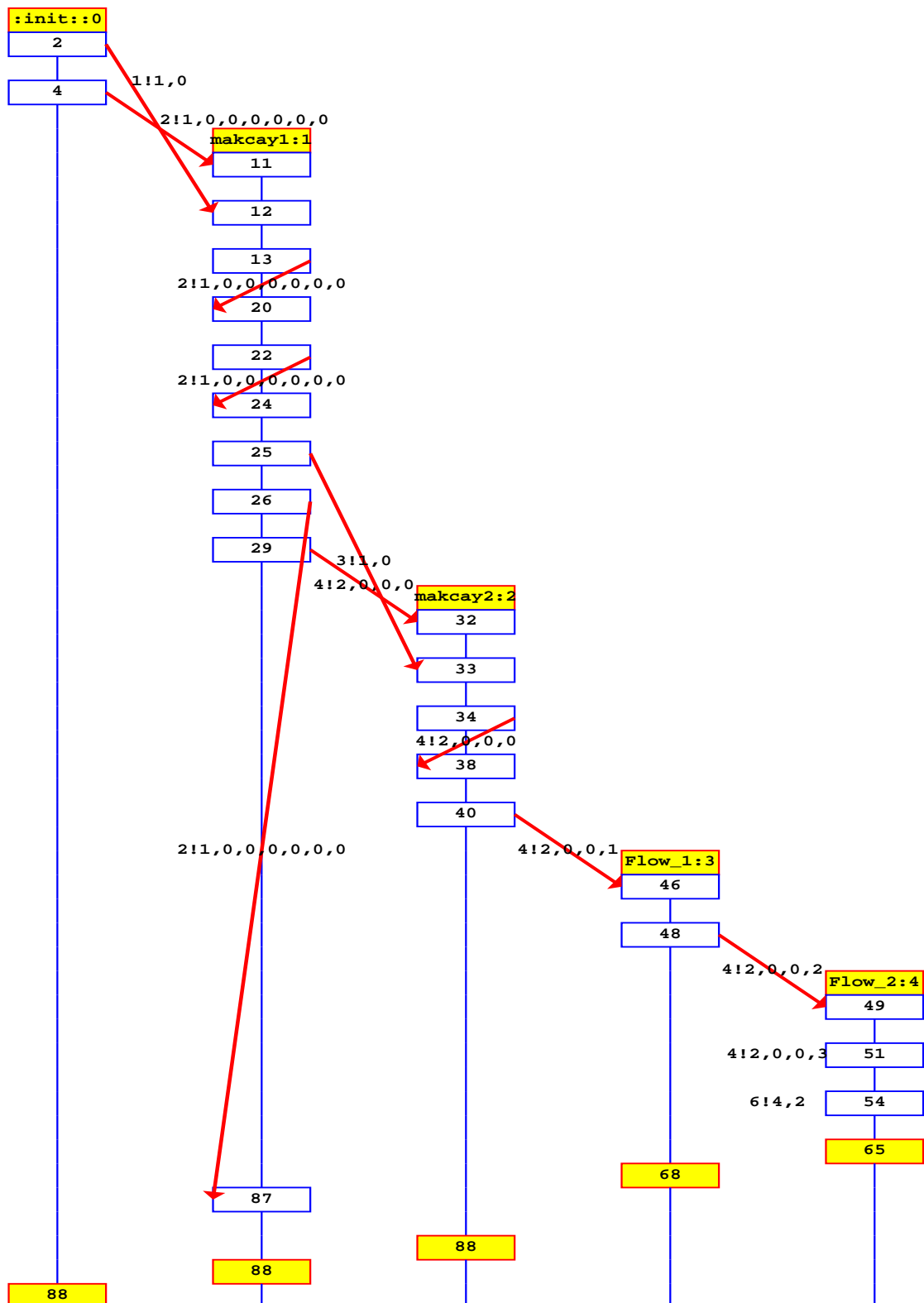


Figure 4.49. XSPIN simulation example

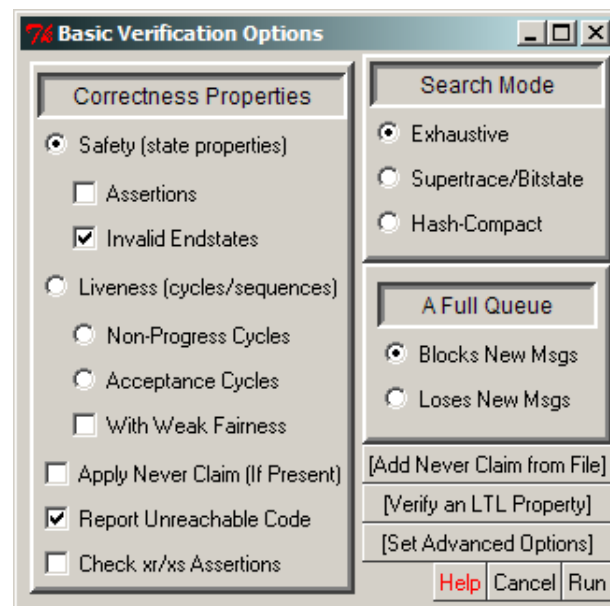


Figure 4.50. XPIN verification settings

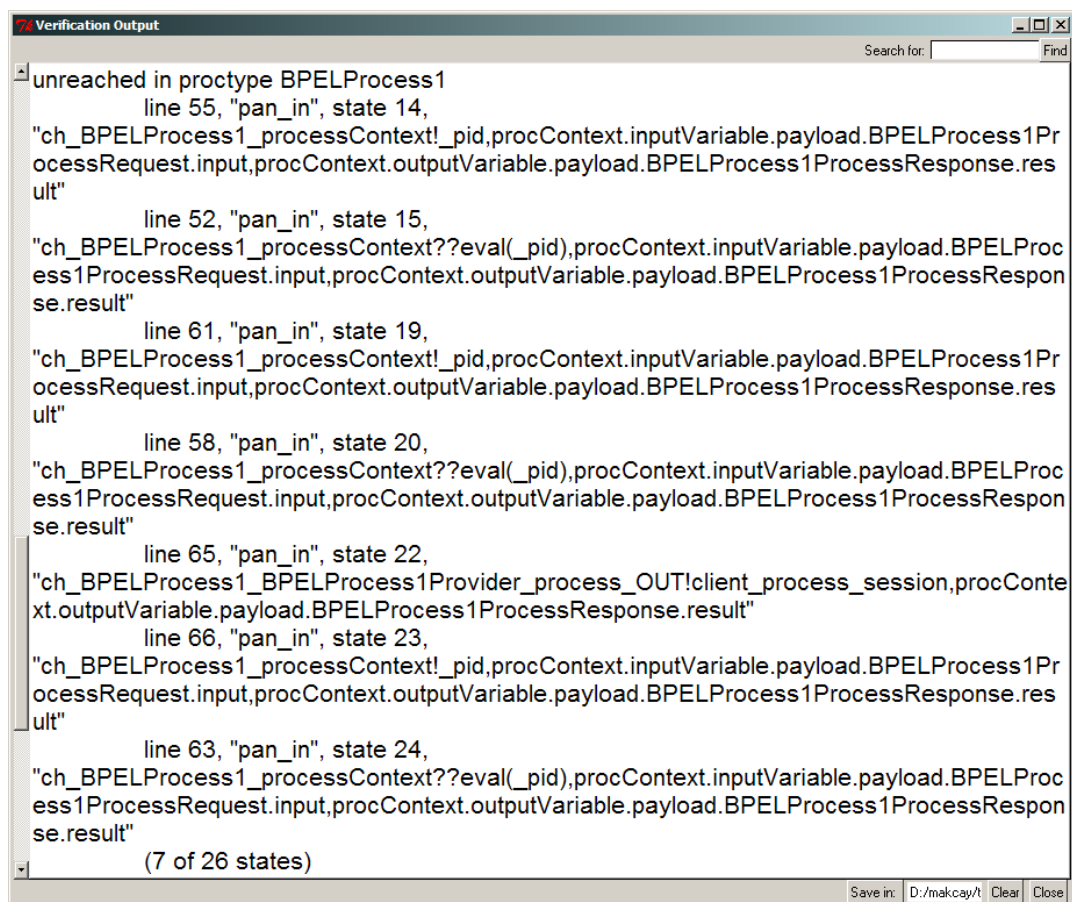


Figure 4.51. Unreachable code verification output

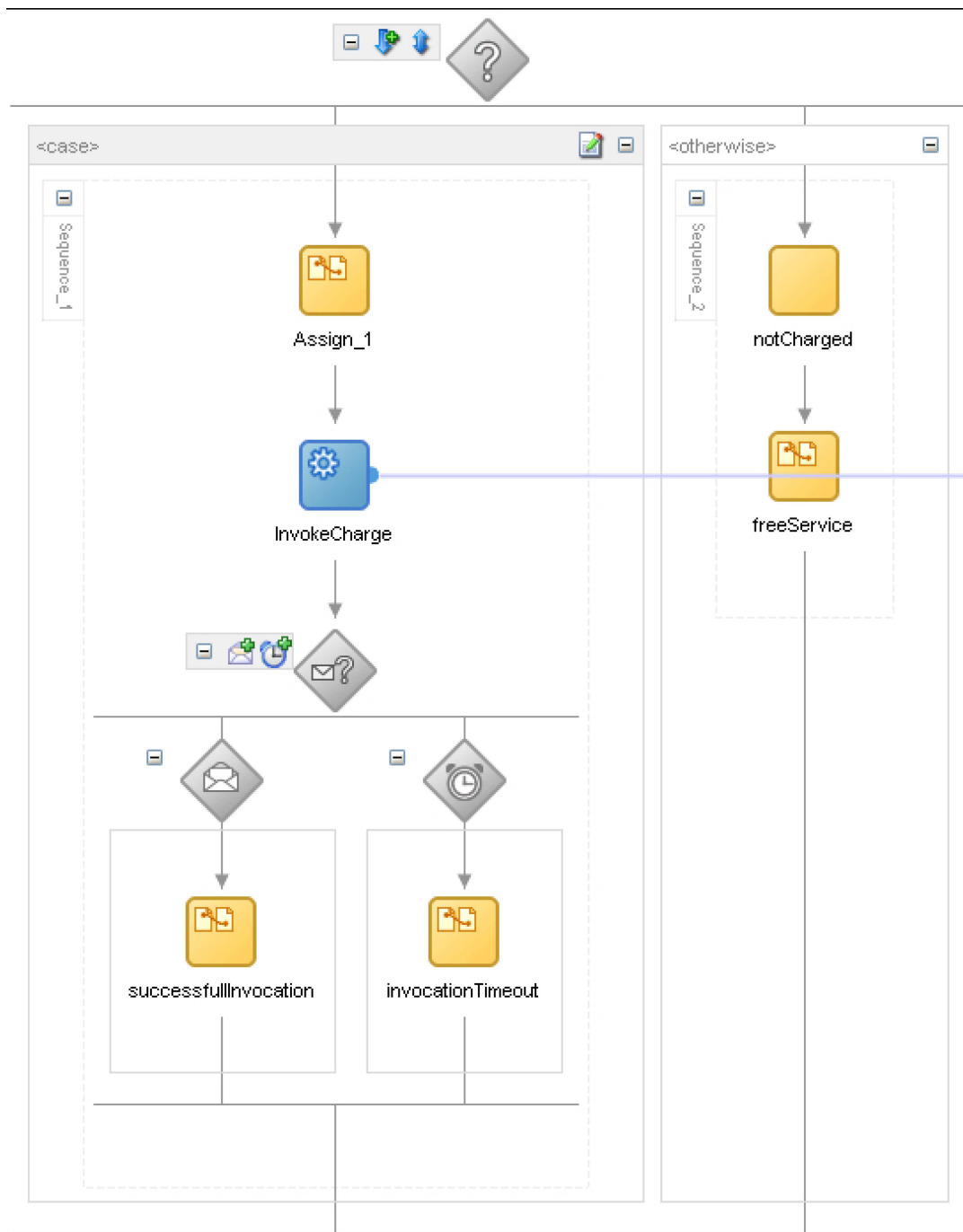


Figure 4.52. Service BPEL process

on the charge flag's value the user is charged or the user is not charged. If the user is charged, then the result from the charge service is received with a pick activity. In order to verify this process to check if everything works correctly, the specification of this process has to be written as an LTL[25] formulae and given to SPIN as a never claim statement. According to the service process specification, the user should

never be charged when the user is not serviced. The same claim can be written as “*if(chargeFlag = true and serviceProcessEnded = true and userServed = false) – >userCharged = false*” which has to hold for all states. This condition should hold in the normal behavior of the service process for all states. Lets assume $p = \text{chargeFlag}$, $q = \text{userServed}$ and $r = \text{userCharged}$, then the LTL definition of the statement above will be:

$$\Box((\text{serviceEnded} \ \&\& \ p \ \&\& \ !q) \rightarrow !r).$$

```

Verification Output
Search for: Find

pan: claim violated! (at depth 94)
pan: wrote pan_in.trail

(Spin Version 5.1.6 -- 9 May 2008)
Warning: Search not completed

Full statespace search for:
    never claim          +
    assertion violations + (if within scope of claim)
    acceptance cycles   + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 1044 byte, depth reached 94, errors: 1
    56 states, stored
    14 states, matched
    70 transitions (= stored+matched)
    27 atomic steps
hash conflicts:      0 (resolved)

Stats on memory usage (in Megabytes):
    0.057 equivalent memory usage for states (stored*(State-vector + overhead))
    0.569 actual memory usage for states (unsuccessful compression: 1004.39%)
           state-vector as stored = 10630 byte + 16 byte overhead
    2.000 memory used for hash table (-w19)
    0.305 memory used for DFS stack (-m10000)
    2.696 total actual memory usage

Save in: D:/makcay/t Clear Close

```

Figure 4.53. Verification output of the service process

The never claim LTL property that is specified to the SPIN should be the negation of the system requirements LTL definition, in this case it will be:

$$\Box((\text{serviceEnded} \ \&\& \ p \ \&\& \ !q) \rightarrow !r)$$

The verification result of the generated model for the service process with the never claim statement generated for the LTL property above is shown in the Figure 4.53. Since SPIN tries to find a counter example by using a never claim, the LTL formulae given to specified model has a negation operator "!" at the beginning of the expression, to find a counter example that doesn't comply to the specification.

In Figure 4.54 the guided simulation output of the counter example found by SPIN verifier is shown. The found counter example shows that init process invokes the service process with chargeFlag equal to true, and as expected service process calls the charge asynchronous web service and waits for the response with a pick activity. The counter example shows that it is possible that the onAlarm branch might be executing before the charge process's response which results the service process return a failure code and not to serve the request. But the charge process might have charged the user even the user is not served by the process according to the counter example case found by SPIN.

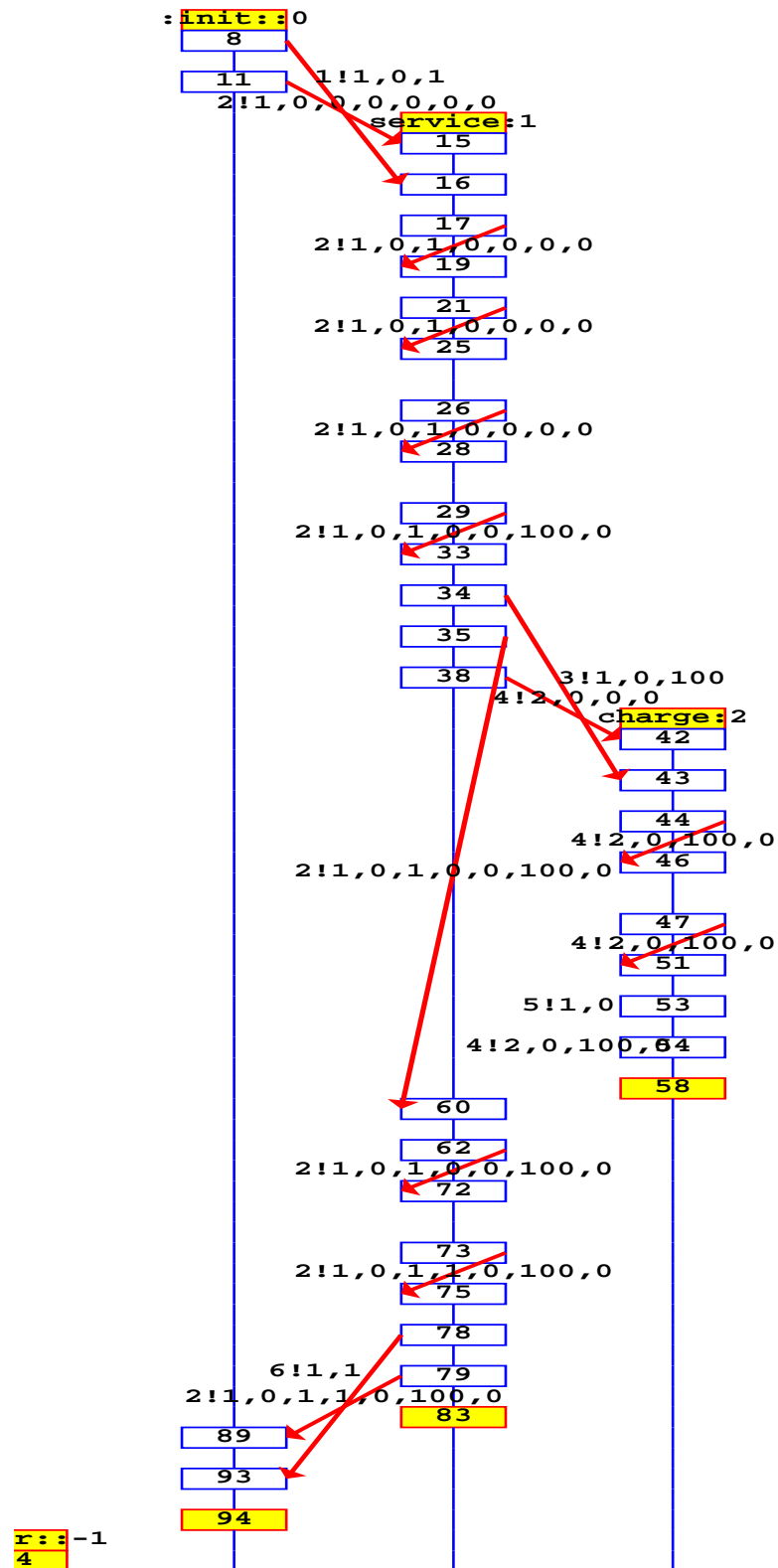


Figure 4.54. Counter example for service process

5. CONCLUSIONS

Verification of BPEL specifications have been studied by several researchers since verification of BPEL specifications is important, because BPEL is a language commonly used for business flows and business rules of a process. Regarding the standard software development cycle waterfall model[26], software development cycle consists of main steps consisting of analysis, design, implementation and testing. Any kind of mistake in the business requirement might also affect analysis, design, development and testing steps. It is also possible to change all the development done previously, if a requirement error is caught during testing phase. That's why verification of BPEL specifications are important to minimize that cost. Also verification of software specifications ensures that the given software specifications are certainly correct which is very important for systems that cannot tolerate any runtime errors. Any runtime error in the business flow of any software could result crucial results whose effects could trigger unacceptable behaviors in some cases.

Verification of BPEL specifications has been studied with different group of people so far. One of the important approaches is the petrinet approach which is based on a conversion from BPEL language to petri net[27] representation. In this approach, all control flows in the BPEL are translated into petri net with the help of the tool[7] created for the mapping between BPEL and petri net. With the wofbpeel tool used in this approach, it is possible to find unreachable activities, potential conflicting message receipts and discarded messages which are never picked from the message queue [8]. But this tool lacks of getting multiple processes and analyzing the communication of multiple BPEL processes. Also, this approach makes an abstraction of xpath queries, bpel expressions and the data manipulation part of a BPEL process, the scope of their work is the analyzing of the control flows [6].

Another approach for the verification of BPEL processes is the guarded automata approach introduced by Xiang Fu[9] in his phd thesis. In this approach a mapping from BPEL processes to guarded automata is done. After the conversion, the generated

guarded automata representation is converted into the PROMELA language by the tool `wsat`[10]. Xpath queries are also implemented in this approach with the help of a new intermediate language for schema types which is called MSL representation. After the translation from guarded automata to PROMELA, the verification of BPEL specification is done by the help of SPIN tool. But the missing part of this approach is that a very limited subset of BPEL activities are implemented in this approach and also it focuses on the communication overhead of the BPEL processes.

In this work, it is shown that any BPEL process that uses the activities discussed in this paper can be translated into PROMELA language and can be verified by using SPIN in order to find the design time errors of the processes. A PROMELA translation for BPEL specifications are mentioned to verify a BPEL process for unreachable codes, assertions and LTL claims. It is also shown how `bpel` specifications can be verified by specifying never claims to SPIN with the generated PROMELA model of the tool `BPEL2PML` and an example scenario is simulated whose results are shown in the previous section. With the approaches discussed in previous chapters, it is shown that any kind of BPEL processes consisting of assign, switch, sequence, empty, while, terminate, scope, flow, throw, invoke, reply, receive, pick, wait, fault handlers and compensation handlers can be verified with the given system specifications using the tool `BPEL2PML` which created for the translation from BPEL to PROMELA.

6. FUTURE WORK

Currently, only the basic XPATH functions and expressions are modeled in order to focus on the verification of BPEL source code mainly. Since the main scope of this project is not modeling the whole XPATH library, as a future direction the whole XPATH library defined in [18] can be implemented and integrated with this work. Since the full XPATH library is very big, only the most crucial ones are modeled in this work. Also in currently proposed method, xml data has to have one unique name for each child node. It is possible to have xml data which has same attribute name in the same hierarchy, which means that there is not a restriction of the occurrence of an attribute in an xml data. But the translations of WSDL and XSD files proposed in this work does not support those kind of variable and xpath queries currently. This bounded implementation can be improved by mapping all the child nodes with a different notation than its original name or some advanced tree structures can be used in order to represent unbounded xml data.

Another future work can be extending the invocation activity translation proposed in this work. The invocation activity of BPEL language and partnerlink definitions are explained in previous chapters. Partnerlinks are the standard naming convention of the outside interaction mechanisms in BPEL. In the WSDL representation, the partnerlink definitions are created and those web service could be called from BPEL using this defined partnerlink definitions. But in most of the real BPEL engine implementations, this partnerlink definitions are extended more than just web services. For example Oracle supports database adaptors, file adaptors...etc. Currently during the invocation translation to PROMELA, it is assumed that the partnerlinks being invoked from a BPEL process will be another BPEL process. This behavior can be extended, so that other kind of partnerlinks could be received by the invoke activity as a further direction like other web services not implemented in BPEL. The external systems could be thought as black box and all possible outcomes could be interpreted by the SPIN as a different behavior. The partnerlink parsing part, invocation, receiving, sending data parts of BPEL processes can be improved in order to be more generic.

APPENDIX A: BPEL Source

A complete BPEL source of containing all possible BPEL Activities.

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!--//////////////////////////////////////
Oracle JDeveloper BPEL Designer

Created: Mon Apr 21 13:39:04 EEST 2008
Author: NIRVANA
Purpose: Synchronous BPEL Process
//////////////////////////////////////
-->
<process
  name = "makcay1"
  targetNamespace = "http://xmlns.oracle.com/makcay1"
  xmlns = "http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpws = "http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:xp20 = "http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.Xpath20"
  xmlns:ids = "http://xmlns.oracle.com/bpel/services/IdentityService/xpath"
  xmlns:ldap = "http://schemas.oracle.com/xpath/extension/ldap"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:client = "http://xmlns.oracle.com/makcay1"
  xmlns:ora = "http://schemas.oracle.com/xpath/extension"
  xmlns:hwf = "http://xmlns.oracle.com/bpel/workflow/xpath"
  xmlns:ns1 = "http://xmlns.oracle.com/makcay2"
  xmlns:ehdr = "http://www.oracle.com/XSL/Transform/java/oracle.tip.esb.server.headers.ESBHeaderFunctions"
  xmlns:ns2 = "http://xmlns.oracle.com/makcay1/correlationset"
  xmlns:bpelx = "http://schemas.oracle.com/bpel/extension"
  xmlns:orcl = "http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.ExtFunc">
  <!--//////////////////////////////////////
PARTNERLINKS
List of services participating in this BPEL process
//////////////////////////////////////
-->
<partnerLinks>
  <!--The 'client' role represents the requester of this service. It is
  used for callback. The location and correlation information associated
  with the client role are automatically set using WS-Addressing.
  -->
  <partnerLink name = "client" partnerLinkType = "client:makcay1" myRole = "makcay1Provider"/>
  <partnerLink name = "makcay2" partnerRole = "makcay2Provider" partnerLinkType = "ns1:makcay2"/>
</partnerLinks>
<!--//////////////////////////////////////
VARIABLES
```

List of messages and XML documents used within this BPEL process

////////////////////////////////////

->

<variables>

<!--Reference to the message passed as input during initiation -->

<variable name = "inputVariable" messageType = "client:makcay1RequestMessage"/>

<!--Reference to the message that will be returned to the requester-->

<variable name = "outputVariable" messageType = "client:makcay1ResponseMessage"/>

<variable name = "counter" type = "xsd:int"/>

<variable name = "Invoke_1_process_InputVariable" messageType = "ns1:makcay2RequestMessage"/>

<variable name = "Invoke_1_process_OutputVariable" messageType = "ns1:makcay2ResponseMessage"/>

<variable name = "OnMessage_process_InputVariable" messageType = "client:makcay1RequestMessage"/>

</variables>

<correlationSets>

<correlationSet name = "CorrelationSet_1" properties = "ns2:prop1"/>

</correlationSets>

<faultHandlers>

<catch faultName = "bpws:selectionFailure" faultVariable = "outputVariable">

<empty name = "Empty_13"/>

</catch>

<catch faultName = "bpws:joinFailure">

<empty name = "Empty_14"/>

</catch>

<catchAll>

<empty name = "Empty_15"/>

</catchAll>

</faultHandlers>

<!--////////////////////////////////////

ORCHESTRATION LOGIC

Set of activities coordinating the flow of messages across the

services integrated within this business process

////////////////////////////////////

->

<sequence name = "main">

<!--Receive input from requestor. (Note: This maps to operation defined in makcay1.wsdl) -->

<receive name = "receiveInput" partnerLink = "client" portType = "client:makcay1" operation = "process"

variable = "inputVariable" createInstance = "yes"/>

<!--Generate reply to synchronous request -->

<assign name = "Assign_6">

<copy>

<from expression = "5"/>

<to variable = "Invoke_1_process_InputVariable" part = "payload" query =

"/ns1:makcay2ProcessRequest/ns1:input"/>

</copy>

</assign>

<invoke name = "Invoke_1" partnerLink = "makcay2" portType = "ns1:makcay2" operation = "process"

inputVariable = "Invoke_1_process_InputVariable" outputVariable = "Invoke_1_process_OutputVariable">

<correlations>

```

    <correlation initiate = "yes" set = "CorrelationSet_1" pattern = "out-in"/>
  </correlations>
</invoke>
<assign name = "Assign_1">
  <copy>
    <from variable = "inputVariable" part = "payload" query = "/client:makcay1ProcessRequest/client:input"/>
    <to variable = "outputVariable" part = "payload" query = "/client:makcay1ProcessResponse/client:result"/>
  </copy>
  <copy>
    <from expression = "5"/>
    <to variable = "counter"/>
  </copy>
  <copy>
    <from expression = "bpws:getVariableData('counter')+1"/>
    <to variable = "counter"/>
  </copy>
</assign>
<flow name = "Flow_links">
  <links>
    <link name = "seq1"/>
  </links>
  <sequence name = "Sequence_7">
    <empty name = "empty1"/>
    <empty name = "empty2">
      <source linkName = "seq1" transitionCondition = "1=1"/>
    </empty>
  </sequence>
  <sequence name = "Sequence_7">
    <empty name = "empty3" joinCondition = "2=2">
      <target linkName = "seq1"/>
    </empty>
    <throw name = "Throw_7" faultName = "bpws:selectionFailure"/>
    <empty name = "empty4"/>
  </sequence>
  <sequence>
    <empty>
      <source linkName = "seq1"/>
    </empty>
    <assign name = "Assign_7">
      <copy>
        <from variable = "inputVariable" part = "payload" query =
"/client:makcay1ProcessRequest/client:input"/>
        <to variable = "counter"/>
      </copy>
    </assign>
  </sequence>
</flow>
<switch name = "Switch_1">

```

```

<case condition = "bpws:getVariableData('counter')=5">
  <sequence name = "Sequence_5">
    <assign name = "Assign_4">
      <copy>
        <from expression = "5"/>
        <to variable = "outputVariable" part = "payload" query =
"/client:makcay1ProcessResponse/client:result"/>
      </copy>
    </assign>
    <empty name = "Empty_2"/>
  </sequence>
</case>
<case condition =
"bpws:getVariableData('inputVariable','payload','/client:makcay1ProcessRequest/client:input')='makcay'">
  <sequence name = "Sequence_6">
    <assign name = "Assign_5">
      <copy>
        <from expression = "2"/>
        <to variable = "counter"/>
      </copy>
    </assign>
    <empty name = "Empty_1"/>
  </sequence>
</case>
<otherwise>
  <sequence name = "Sequence_9">
    <empty name = "Empty_3"/>
    <throw name = "true" faultName = "bpws:selectionFailure"/>
  </sequence>
</otherwise>
</switch>
<while name = "While_1" condition = "bpws:getVariableData('counter')<10">
  <sequence name = "Sequence_1">
    <assign name = "Assign_2">
      <copy>
        <from expression = "bpws:getVariableData('counter')+1"/>
        <to variable = "counter"/>
      </copy>
    </assign>
    <empty name = "Empty_4"/>
  </sequence>
</while>
<pick name = "Pick_1">
  <onMessage portType = "client:makcay1" operation = "process" variable =
"OnMessage_process_InputVariable" partnerLink = "client">
    <empty name = "Empty_16"/>
  </onMessage>
  <onAlarm for = "PT30S">

```

```
        <empty name = "Empty_17"/>
    </onAlarm>
</pick>
    <throw name = "Throw_5" faultName = "bpws:conflictingReceive" faultVariable = "inputVariable"/>
    <reply name = "replyOutput" partnerLink = "client" portType = "client:makcay1" operation = "process"
variable = "outputVariable"/>
</sequence>
</process>
```



```

<?xml version = "1.0" encoding = "UTF-8" ?>
<!--//////////////////////////////////////
Oracle JDeveloper BPEL Designer

Created: Tue Apr 29 20:39:20 EEST 2008
Author: NIRVANA
Purpose: Synchronous BPEL Process
//////////////////////////////////////
-->
<process
  name = "makcay2"
  targetNamespace = "http://xmlns.oracle.com/makcay2"
  xmlns = "http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpws = "http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:xp20 = "http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.Xpath20"
  xmlns:hwf = "http://xmlns.oracle.com/bpel/workflow/xpath"
  xmlns:ids = "http://xmlns.oracle.com/bpel/services/IdentityService/xpath"
  xmlns:ldap = "http://schemas.oracle.com/xpath/extension/ldap"
  xmlns:ehdr = "http://www.oracle.com/XSL/Transform/java/oracle.tip.esb.server.headers.ESBHeaderFunctions"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:bpelx = "http://schemas.oracle.com/bpel/extension"
  xmlns:client = "http://xmlns.oracle.com/makcay2"
  xmlns:ora = "http://schemas.oracle.com/xpath/extension"
  xmlns:orcl = "http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.ExtFunc">
  <!--//////////////////////////////////////
PARTNERLINKS
List of services participating in this BPEL process
//////////////////////////////////////
-->
<partnerLinks>
  <!--The 'client' role represents the requester of this service. It is
  used for callback. The location and correlation information associated
  with the client role are automatically set using WS-Addressing.
  -->
  <partnerLink name = "client" partnerLinkType = "client:makcay2" myRole = "makcay2Provider"/>
</partnerLinks>
<!--//////////////////////////////////////
VARIABLES    List of messages and XML documents used within this BPEL process
//////////////////////////////////////
-->
<variables>
  <!--Reference to the message passed as input during initiation -->
  <variable name = "inputVariable" messageType = "client:makcay2RequestMessage"/>
  <!--Reference to the message that will be returned to the requester-->
  <variable name = "outputVariable" messageType = "client:makcay2ResponseMessage"/>
  <variable name = "counter" type = "xsd:int"/>
</variables>
<!--//////////////////////////////////////

```

ORCHESTRATION LOGIC Set of activities coordinating the flow of messages across the services integrated within this business process

```

////////////////////////////////////
->
<sequence name = "main">
  <!--Receive input from requestor. (Note: This maps to operation defined in makcay2.wsdl) -->
  <receive name = "receiveInput" partnerLink = "client" portType = "client:makcay2" operation = "process"
variable = "inputVariable" createInstance = "yes"/>
  <!--Generate reply to synchronous request -->
  <assign name = "Assign_1">
    <copy>
      <from expression = "1"/>
      <to variable = "counter"/>
    </copy>
  </assign>
  <flow name = "Flow_1">
    <sequence name = "Sequence_1">
      <assign name = "Assign_3">
        <copy>
          <from expression = "bpws:getVariableData('counter')+1"/>
          <to variable = "counter"/>
        </copy>
      </assign>
    </sequence>
    <sequence name = "Sequence_1">
      <assign name = "Assign_2">
        <copy>
          <from expression = "bpws:getVariableData('counter')+1"/>
          <to variable = "counter"/>
        </copy>
      </assign>
    </sequence>
  </flow>
  <reply name = "replyOutput" partnerLink = "client" portType = "client:makcay2" operation = "process"
variable = "outputVariable"/>
</sequence>
</process>

```

APPENDIX B: PROMELA Conversion

The PROMELA conversion of the BPEL process specified above containing all possible BPEL activities. The translation of all the activities can be seen in the translation below.

```

typedef makcay1ProcessRequest_type_seq{
    mtype input;
}
typedef makcay1ProcessRequest_type{
    makcay1ProcessRequest_type_seq makcay1ProcessRequest
}
typedef makcay1RequestMessage{
    makcay1ProcessRequest_type payload;
}
typedef makcay1ProcessResponse_type_seq{
    mtype result;
}
typedef makcay1ProcessResponse_type{
    makcay1ProcessResponse_type_seq makcay1ProcessResponse
}
typedef makcay1ResponseMessage{
    makcay1ProcessResponse_type payload;
}
typedef makcay2ProcessRequest_type_seq{
    mtype input;
}
typedef makcay2ProcessRequest_type{
    makcay2ProcessRequest_type_seq makcay2ProcessRequest
}
typedef makcay2RequestMessage{
    makcay2ProcessRequest_type payload;
}
typedef makcay2ProcessResponse_type_seq{
    mtype result;
}
typedef makcay2ProcessResponse_type{
    makcay2ProcessResponse_type_seq makcay2ProcessResponse
}
typedef makcay2ResponseMessage{
    makcay2ProcessResponse_type payload;
}

typedef makcay1_processContext{

```

```

makcay1RequestMessage inputVariable;
makcay1ResponseMessage outputVariable;
int counter;
makcay2RequestMessage Invoke_1_process_InputVariable;
makcay2ResponseMessage Invoke_1_process_OutputVariable;
makcay1RequestMessage OnMessage_process_InputVariable;
}
typedef makcay2_processContext{
    makcay2RequestMessage inputVariable_2;
    makcay2ResponseMessage outputVariable_2;
    int counter_2;
}

chan terminatedProcesses=[10] of int,int;
int ch_makcay1_makcay1Provider_process_IN_session=0;
chan ch_makcay1_makcay1Provider_process_IN=[10] of int,makcay1RequestMessage;
int ch_makcay1_makcay1Provider_process_OUT_session=0;
chan ch_makcay1_makcay1Provider_process_OUT=[10] of int,makcay1ResponseMessage;
int ch_makcay2_makcay2Provider_process_IN_session=0;
chan ch_makcay2_makcay2Provider_process_IN=[10] of int,makcay2RequestMessage;
int ch_makcay2_makcay2Provider_process_OUT_session=0;
chan ch_makcay2_makcay2Provider_process_OUT=[10] of int,makcay2ResponseMessage;
chan unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure=[10] of int,int;
chan ch_makcay1_processContext=[10] of int,makcay1_processContext;
chan ch_makcay2_processContext=[10] of int,makcay2_processContext;

bool link_seq1;
bool link_seq1_eval=false;

mtype = m_makcay;

proctype makcay1(int _parentId){
    int client_process_session;
    int makcay2_process_CorrelationSet_1_session;
    int pid_1;
    int pid_2;
    int pid_3;
    int pid_4;
    int tmpSession;
    int tmpPid;

    makcay1_processContext procContext;

    do
        ::ch_makcay1_makcay1Provider_process_IN?[_pid,procContext.inputVariable]->
        atomic{
            ch_makcay1_processContext??eval(_pid),procContext;
            ch_makcay1_makcay1Provider_process_IN?client_process_session,procContext.inputVariable;

```

```

        ch_makcay1_processContext! _pid,procContext;
    }
    break;
    ::empty(ch_makcay1_makcay1Provider_process_IN)->skip;
od;
atomic{
    ch_makcay1_processContext??eval(_pid),procContext;
    procContext.Invoke_1_process_InputVariable.payload.makcay2ProcessRequest.input=5;
    ch_makcay1_processContext! _pid,procContext;
}
makcay2_process_CorrelationSet_1_session=ch_makcay2_makcay2Provider_process_IN_session+1;
atomic{
    ch_makcay1_processContext??eval(_pid),procContext;
    ch_makcay2_makcay2Provider_process_IN !
makcay2_process_CorrelationSet_1_session,procContext.Invoke_1_process_InputVariable;
    ch_makcay1_processContext! _pid,procContext;
}
makcay2_processContext makcay2_procContext;
atomic{
    pid_4=run makcay2(_pid);
    ch_makcay2_processContext!pid_4,makcay2_procContext;
}
do
    ::enabled(pid_4)==false->break;
    ::else->skip;
od;
atomic{
    ch_makcay1_processContext??eval(_pid),procContext;
    if
        ::ch_makcay2_makcay2Provider_process_OUT ??
eval(makcay2_process_CorrelationSet_1_session),procContext.Invoke_1_process_OutputVariable->skip;
    fi;
    ch_makcay1_processContext! _pid,procContext;
}
atomic{
    ch_makcay1_processContext??eval(_pid),procContext;
    procContext.outputVariable.payload.makcay1ProcessResponse.result =
procContext.inputVariable.payload.makcay1ProcessRequest.input;
    ch_makcay1_processContext! _pid,procContext;
}
atomic{
    ch_makcay1_processContext??eval(_pid),procContext;
    procContext.counter=5;
    ch_makcay1_processContext! _pid,procContext;
}
atomic{
    ch_makcay1_processContext??eval(_pid),procContext;
    procContext.counter=procContext.counter+1;
}

```

```

    ch_makcay1_processContext!_pid,procContext;
}
link_seq1_eval=false;
pid_1=run proc_Flow_links_Sequence_7_FlowProcess(_pid);
pid_2=run proc_Flow_links_Sequence_7_FlowProcess_2(_pid);
pid_3=run proc_Flow_links_FlowProcess(_pid);
do
    ::else->skip;
    ::(enabled(pid_1)==false && enabled(pid_2)==false && enabled(pid_3)==false)->break;
od;
if
    ::terminatedProcesses??_,_pid->
        goto end_makcay1;
fi;
if
    ::unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure??eval(pid_2),eval(_pid)->
        goto faultHandler_selectionFailure;
    ::(empty(unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure))->skip;
fi;

if
    :: procContext.counter==5 ->
        atomic{
            ch_makcay1_processContext??eval(_pid),procContext;
            procContext.outputVariable.payload.makcay1ProcessResponse.result=5;
            ch_makcay1_processContext!_pid,procContext;
        }
        skip;
    :: else->
        if
            ::procContext.inputVariable.payload.makcay1ProcessRequest.input==m_makcay ->
                atomic{
                    ch_makcay1_processContext??eval(_pid),procContext;
                    procContext.counter=2;
                    ch_makcay1_processContext!_pid,procContext;
                }
                skip;
            ::else->
                skip;
                goto faultHandler_selectionFailure;
        fi;
fi;
do
    :: procContext.counter<10 -> atomic{
        ch_makcay1_processContext??eval(_pid),procContext;
        procContext.counter=procContext.counter+1;
        ch_makcay1_processContext!_pid,procContext;
    }

```

```

        skip;
        :: else -> break;
    od;
    atomic{
        ch_makcay1_processContext??eval(_pid),procContext;
        if
            ::ch_makcay1_makcay1Provider_process_IN ?
client_process_session,procContext.OnMessage_process_InputVariable->
            ch_makcay1_processContext!_pid,procContext;
            skip;
            ::true->
            ch_makcay1_processContext!_pid,procContext;
            skip;
        fi;
    }
    goto faultHandler;
    atomic{
        ch_makcay1_processContext??eval(_pid),procContext;
        ch_makcay1_makcay1Provider_process_OUT!client_process_session,procContext.outputVariable;
        ch_makcay1_processContext!_pid,procContext;
    }

    goto end_makcay1;

faultHandler_selectionFailure:
    skip;
    goto end_makcay1;
faultHandler_joinFailure:
    skip;
    goto end_makcay1;
faultHandler:
    skip;
    goto end_makcay1;
end_makcay1:skip;
}

proctype makcay2(int _parentId){
    int client_process_session;
    int pid_1;
    int pid_2;
    int tmpSession;
    int tmpPid;

    makcay2_processContext procContext;

    do
        ::ch_makcay2_makcay2Provider_process_IN?[_,procContext.inputVariable_2]->
            atomic{

```

```

    ch_makcay2_processContext??eval(_pid),procContext;
    ch_makcay2_makcay2Provider_process_IN?client_process_session,procContext.inputVariable_2;
    ch_makcay2_processContext!_pid,procContext;
}
break;
::empty(ch_makcay2_makcay2Provider_process_IN)->skip;
od;
atomic{
    ch_makcay2_processContext??eval(_pid),procContext;
    procContext.counter_2=1;
    ch_makcay2_processContext!_pid,procContext;
}
pid_1=run proc_Flow_1_Sequence_1_FlowProcess(_pid);
pid_2=run proc_Flow_1_Sequence_1_FlowProcess_2(_pid);
do
    ::else->skip;
    ::(enabled(pid_1)==false && enabled(pid_2)==false)->break;
od;
if
    ::terminatedProcesses??_,_pid->
        goto end_makcay2;
fi;
if
    ::skip;
fi;

atomic{
    ch_makcay2_processContext??eval(_pid),procContext;
    ch_makcay2_makcay2Provider_process_OUT!client_process_session,procContext.outputVariable_2;
    ch_makcay2_processContext!_pid,procContext;
}

end_makcay2:skip;
}

proctype proc_Flow_links_Sequence_7_FlowProcess(int _parentId){
    int tmpSession;
    int tmpPid;

    makcay1_processContext procContext;

    {
        skip;
        skip;
        link_seq1=(1==1);
        link_seq1_eval=true;
    }
}

```



```

    unless terminatedProcesses?[_,_parentId] ||
unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure?[_eval(_parentId)];
end_Sequence_7:skip;
}

proctype proc_Flow_links_Sequence_7_FlowProcess_2(int _parentId){
    int tmpSession;
    int tmpPid;

    makcay1_processContext procContext;

    {
        do
            ::link_seq1_eval==true->break;
        od;
        if
            ::(link_seq1==true)->
                skip;
            ::else->assert(link_seq1==true);
        fi;
        atomic{
            unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure!_pid,_parentId;
            goto end_Sequence_7_2;
        }
        skip;
    }
    unless terminatedProcesses?[_,_parentId] ||
unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure?[_eval(_parentId)];
end_Sequence_7_2:skip;
}

proctype proc_Flow_links_FlowProcess(int _parentId){
    int tmpSession;
    int tmpPid;

    makcay1_processContext procContext;

    {
        skip;
        link_seq1=true;
        link_seq1_eval=true;
        atomic{
            ch_makcay1_processContext??eval(_parentId),procContext;
            procContext.counter=procContext.inputVariable.payload.makcay1ProcessRequest.input;
            ch_makcay1_processContext!_parentId,procContext;
        }
    }
}

```

```

    unless terminatedProcesses?[_,_parentId] ||
unhandledEx_proc_Flow_links_Sequence_7_FlowProcess_2_selectionFailure?[_eval(_parentId)];
end:skip;
}

proctype proc_Flow_1_Sequence_1_FlowProcess(int _parentId){
    int tmpSession;
    int tmpPid;

    makcay2_processContext procContext;

    {
        atomic{
            ch_makcay2_processContext??eval(_parentId),procContext;
            procContext.counter_2=procContext.counter_2+1;
            ch_makcay2_processContext!_parentId,procContext;
        }
    }
    unless terminatedProcesses?[_,_parentId];
end_Sequence_1:skip;
}

proctype proc_Flow_1_Sequence_1_FlowProcess_2(int _parentId){
    int tmpSession;
    int tmpPid;

    makcay2_processContext procContext;

    {
        atomic{
            ch_makcay2_processContext??eval(_parentId),procContext;
            procContext.counter_2=procContext.counter_2+1;
            ch_makcay2_processContext!_parentId,procContext;
        }
    }
    unless terminatedProcesses?[_,_parentId];
end_Sequence_1_2:skip;
}

init {
    int tmpSession;

    int _session_0;
    makcay1RequestMessage makcay1RequestMessage_inst;
    _session_0=ch_makcay1_makcay1Provider_process_IN_session+1;
    ch_makcay1_makcay1Provider_process_IN!_session_0,makcay1RequestMessage_inst;
    int pid_0;
    makcay1_processContext makcay1_procContext;

```

```

atomic{
    pid_0=run makcay1(_pid);
    ch_makcay1_processContext!pid_0,makcay1_procContext;
}
makcay1ResponseMessage makcay1ResponseMessage_inst;
do
    ::enabled(pid_0)==false->
        ch_makcay1_processContext??eval(pid_0),makcay1_procContext;
        break;
    ::else->skip;
od;
if
    ::ch_makcay1_makcay1Provider_process_OUT??eval(_session_0),makcay1ResponseMessage_inst->skip;
fi;

int _session_2;
makcay2RequestMessage makcay2RequestMessage_inst;
_session_2=ch_makcay2_makcay2Provider_process_IN_session+1;
ch_makcay2_makcay2Provider_process_IN!_session_2,makcay2RequestMessage_inst;
int pid_2;
makcay2_processContext makcay2_procContext;
atomic{
    pid_2=run makcay2(_pid);
    ch_makcay2_processContext!pid_2,makcay2_procContext;
}
makcay2ResponseMessage makcay2ResponseMessage_inst;
do
    ::enabled(pid_2)==false->
        ch_makcay2_processContext??eval(pid_2),makcay2_procContext;
        break;
    ::else->skip;
od;
if
    ::ch_makcay2_makcay2Provider_process_OUT??eval(_session_2),makcay2ResponseMessage_inst->skip;
fi;

}

```

REFERENCES

1. Linthicum, D., “Building a SOA”, *SOA World Magazine*, Vol. Volume 7, Issue 5, Jun 2, 2007.
2. Katoen, J.-P., “Concepts, Algorithms, and Tools for Model Checking Lecture Notes of the Course "Mechanised Validation of Parallel Systems" (course number 10359)”, Universität Erlangen-Nürnberg, 1999.
3. Korovin, K., “CS2142 Lecture Notes, Chapter 14, Linear Temporal Logic, pages 212-215”, The University of Manchester, School of Computer Science, 2006.
4. Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, *Web Services Architecture*, Tech. rep., W3C, 11 February 2004.
5. Ouimet, M., *Formal Software Verification: Model Checking and Theorem Proving*, Tech. rep., Embedded Systems Laboratory Massachusetts Institute of Technology, Cambridge, MA, 02139, USA, 2005.
6. Servais, F., *Verifying and Testing BPEL Processes*, Master’s thesis, Université Libre de Bruxelles, 2007.
7. Ouyang, C., E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas and A. H. M. ter Hofstede, “Wofbpele: A tool for automated analysis of bpm processes”, C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas and A. H. M. ter Hofstede (Editors), *Proceedings Third International Conference on Service Oriented Computing (ICSOC 2005)* 3826, ICSOC, pp. 484–489, Springer Berlin / Heidelberg, Amsterdam, The Netherlands, 2005.
8. Ouyang, C., E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas and A. H. M. ter Hofstede, “Formal semantics and analysis of control flow in WS-BPEL”, *Sci. Comput. Program.*, Vol. Vol. 67, No. 2-3, pp. 162–198, 2007.

9. Fu, X., *Formal Specification and Verification of Asynchronously Communicating Web Services*, Ph.D. thesis, UNIVERSITY of CALIFORNIA, Santa Barbara, 2004.
10. Fu, X., T. Bultan and J. Su, “WSAT: A Tool for Formal Analysis of Web Services”, *Computer Aided Verification*, Vol. Volume 3114/2004 of *Lecture Notes in Computer Science*, pp. 510–514, Springer Berlin / Heidelberg, 2004.
11. Oracle, *Oracle Bpel Process Manager Developer’s Guide 10g Release 2 (10.1.2) Part No. B14448-01*, 2005.
12. Heljanko, K., *Model Checking based Software Verification*, Tech. rep., Laboratory for Theoretical Computer Science Department of Computer Science and Engineering Helsinki University of Technology, Helsinki, Finland, 2006.
13. Mukund, M., “Linear-Time Temporal Logic and Büchi Automata”, SPIC Mathematical Institute Winter School on Logic, 92 G N Chetty Road Madras 600 017, India, 1996.
14. Holzmann, G. J. (Editor), *The SPIN Model Checker*, Addison-Wesley, 2003.
15. Newcomer, E. and G. Lomow, *Understanding SOA with Web Services (Independent Technology Guides)*, Addison-Wesley Professional, 2004.
16. Clarke, E. M., O. Grumberg and D. A. Peled (Editors), *Model checking*, Vol. ISBN 0262032708, MIT Press, Heidelberg, 1999.
17. Erik Christensen, M., I. R. Francisco Curbera, M. Greg Meredith and I. R. Sanjiva Weerawarana, *Web Services Description Language (WSDL) 1.1*, Tech. rep., W3C, 2001.
18. Clark, J. and S. DeRose, *XML Path Language (XPath) Version 1.0*, Tech. rep., W3C, 1999.

19. Lesk, M. E. and E. Schmidt, *Lex - a lexical analyzer generator*, Tech. Rep. Computing Science Technical Report No.39, Bell Telephone Laboratories, 1975.
20. Berk, E., *JLex: A lexical analyzer generator for Java*, Department of Computer Science, Princeton University, Version 1.2, May 5, 1997, Last updated September 6, 2000 for JLex 1.2.5.
21. Hudson, S., F. Flannery and C. S. Ananian, *CUP User's Manual*, Georgia Institute of Technology, Atlanta, Georgia, 1999.
22. Holzmann, G. J., "The Model Checker SPIN", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Vol. Vol. 23, No. 5, pp. 279–295, May 1997.
23. Andrews, T., F. Curbera, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, *Business Process Execution Language for Web Services Version 1.1*, Tech. rep., BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, 2003.
24. Ruys, T. C., "Xspin/Project - Integrated Validation Management for Xspin", *Theoretical and Practical Aspects of SPIN Model Checking*, Lecture Notes in Computer Science Volume 1680/2008, pp. 108–119, Springer Berlin / Heidelberg, 1999.
25. Alur, R. and T. A. Henzinger, "A really temporal logic", *Journal of the ACM*, Vol. Volume 41, Issue 1, pp. 181–203, 1994.
26. Davis, A. M., H. Bersoff and E. R. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models", *IEEE Trans. Softw. Eng.*, Vol. 14, No. 10, pp. 1453–1461, 1988.
27. Peterson, J. L., "Petri Nets", *Computing Surveys*, Vol. Vol. 9, 1977.
28. van der Aalst, W., M. Beisiegel, K. van Hee, D. König and C. Stahl, "A

- SOA-Based Architecture Framework”, F. Leymann, W. Reisig, S. R. Thatte and W. van der Aalst (Editors), *The Role of Business Processes in Service Oriented Architectures*, No. 06291 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
29. Kusy, B. and S. Abdelwahed, *FTSP Protocol Verification using SPIN*, Tech. rep., Institute for Software Integrated Systems Vanderbilt University, Nashville, Tennessee, 37235, 2006.
 30. Holzmann, G. J. and M. H. Smith, “Software Model Checking”, *FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pp. 481–497, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1999.
 31. Holzmann, G. J., “The Model Checker Spin”, *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Vol. Vol. 23, 1997.
 32. Laboratory, P. R. G. N. J. P. and D. G. J. H. C. S. R. B. Labs, “Using SPIN Model Checking for Flight Software Verification”, *IEEE Aerospace Conference Proceedings, 2002*, Vol. Vol. 1 of *Aerospace Conference Proceedings*, pp. 105–113, IEEE, 2002.
 33. Merz, S., *Model Checking: A Tutorial Overview*, Institut für Informatik, Universität München, München, 2000.
 34. Kand, K., M. Das and A. Yiu, “A Close Look at BPEL 2.0”, *SOA World Magazine*, Oct. 2007.
 35. Abrams, M. D., S. Jajodia and H. J. Podell (Editors), *Information Security: An Integrated Collection of Essays*, chap. 8, pp. 170–186, IEEE Computer Society

Press, Los Alamitos, CA USA, 1995.

36. Cao, H., S. Ying and D. Du, “Towards Model-based Verification of BPEL with Model Checking”, *CIT '06: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, p. 190, IEEE Computer Society, Washington, DC, USA, 2006.
37. Jackson, D., *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.