

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

— * —

ĐỒ ÁN
TỐT NGHIỆP ĐẠI HỌC

NGÀNH CÔNG NGHỆ THÔNG TIN

XÂY DỰNG PHƯƠNG PHÁP KIỂM ĐỊNH
TIẾN TRÌNH BPEL

Sinh viên thực hiện : **Bùi Hoàng Đức**

Lớp CNPM – K50

Giáo viên hướng dẫn: PGS.TS.

Huỳnh Quyết Thắng

HÀ NỘI 6-2010

HANOI UNIVERS HANOI UNIVERSITY OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

— * —

A METHOD OF BPEL PROCESS VERIFICATION

SUBMITTED JUNE/2010 IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF ENGINEER OF SCIENCE IN
COMPUTER SCIENCE

Student : **Bùi Hoàng Đức**
Software Engineering – K50
Supervisor : A/Prof. Ph.D.
Huỳnh Quyết Thắng

PHIẾU GIAO NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP

1. Thông tin về sinh viên

Họ và tên sinh viên: Bùi Hoàng Đức

Điện thoại liên lạc: 0972347051

Email: ducbuihoang@gmail.com

Lớp: CNPM A – K50

Hệ đào tạo: Đại học chính quy

Đồ án tốt nghiệp được thực hiện tại: Bộ môn CNPM

Thời gian làm ĐATN: Từ ngày 21 / 12 /2010 đến 28 / 05 /2010

2. Mục đích nội dung của ĐATN

Đề xuất và xây dựng thuật toán cùng công cụ để kiểm định các dịch vụ web phức hợp được tích phối bởi tiến trình BPEL, cụ thể là dịch chuyển các tiến trình BPEL sang dạng đồ thị luồng điều khiển có dán nhãn, rồi dịch chuyển sang chương trình viết bằng ngôn ngữ Promela. Để minh họa tạo ra công cụ cài đặt các thuật toán trên nhằm kiểm định các tiến trình BPEL với các thuộc tính thông dụng.

3. Các nhiệm vụ cụ thể của ĐATN

- (1) Nghiên cứu kiến trúc phần mềm hướng dịch vụ, dịch vụ web và ngôn ngữ thực thi quy trình nghiệp vụ (BPEL).
- (2) Nghiên cứu các phương pháp hình thức trong phát triển phần mềm, đặc biệt là phương pháp kiểm định mô hình cùng các trình kiểm tra mô hình.
- (3) Xây dựng phương pháp kiểm định các tiến trình đặc tả bằng BPEL sử dụng trình kiểm tra mô hình Spin.
- (4) Xây dựng các cấu trúc dữ liệu để xử lý các tiến trình BPEL và đồ thị luồng điều khiển có dán nhãn và sinh mã nguồn của ngôn ngữ Promela một cách có hệ thống.
- (5) Xây dựng công cụ thực hiện các phương pháp và giải pháp trên.
- (6) Đánh giá phương pháp và thử nghiệm công cụ trên.

4. Lời cam đoan của sinh viên:

Tôi – *Bùi Hoàng Đức* - cam kết ĐATN là công trình nghiên cứu của bản thân tôi dưới sự hướng dẫn của *PGS.TS Huỳnh Quyết Thắng*.

Các kết quả nêu trong ĐATN là trung thực, không phải là sao chép toàn văn của bất kỳ công trình nào khác.

Hà Nội, ngày tháng năm

Tác giả ĐATN

Bùi Hoàng Đức

5. Xác nhận của giáo viên hướng dẫn về mức độ hoàn thành của ĐATN và cho phép bảo vệ:

Hà Nội, ngày tháng năm
Giáo viên hướng dẫn

PGS.TS. Huỳnh Quyết Thắng

ABSTRACT OF THESIS

This graduation thesis works towards an effective solution of verifying composite web services composed using Business Process Execution Language (BPEL), which is the language specialized for the composition of distributed web services.

First, the author provides the theoretical foundations for service-oriented architecture, formal methods in software verification, underlying formal model of Promela language and the Spin model checker. After that, the problem of verifying BPEL business process specifications and current research approaches that have been conducted on the world are presented. Basing on these research approaches, the author propose a new solution that includes transformation algorithms.

To illustrate the proposed solution, the author developed a tool called BVT (BPEL Verification Tool) that includes the implementation of transformation algorithms and several metamodels. Finally, some testing results and evaluation of the solution will be presented.

TÓM TẮT NỘI DUNG ĐỒ ÁN TỐT NGHIỆP

Đồ án tốt nghiệp này nghiên cứu giải pháp cho bài toán kiểm định các dịch vụ web phức hợp được tích phối bằng chương trình viết bằng ngôn ngữ thực thi quy trình nghiệp vụ (BPEL), ngôn ngữ chuyên dùng cho tích phối các dịch vụ web phân tán.

Đầu tiên, tác giả trình bày về cơ sở lý thuyết của kiến trúc hướng dịch vụ, các phương pháp hình thức trong kiểm định phần mềm, mô hình hình thức của ngôn ngữ Promela và trình kiểm tra mô hình Spin. Sau đó là mô tả về vấn đề kiểm định các đặc tả quy trình nghiệp vụ BPEL và các hướng nghiên cứu đã được thực hiện trên thế giới. Dựa trên những hướng nghiên cứu này tác giả đề xuất một giải pháp mới cùng các thuật toán dịch chuyển.

Để minh họa cho giải pháp đề xuất, tác giả xây dựng công cụ gọi là BVT (BPEL Verification Tool) cài đặt các thuật toán cùng với các mô hình đối tượng. Cuối cùng là kết quả thử nghiệm, đánh giá giải pháp đề xuất.

LỜI CẢM ƠN

Trước hết, em xin được gửi lời cảm ơn sâu sắc tới các thầy cô giáo trong trường Đại học Bách Khoa Hà Nội nói chung và các thầy cô trong Viện Công Nghệ Thông Tin và Truyền Thông, bộ môn Công nghệ phần mềm nói riêng đã tận tình giảng dạy, truyền đạt cho em những kiến thức, những kinh nghiệm quý báu trong suốt 5 năm học tập và rèn luyện tại trường.

Em xin được gửi lời cảm ơn đến **PGS. TS. Huỳnh Quyết Thắng**, giảng viên bộ môn Công Nghệ Phần Mềm, Viện Công Nghệ Thông Tin và Truyền Thông, trường đại học Bách Khoa Hà Nội đã tận tình giúp đỡ em trong suốt quá trình thực hiện đồ án. Ngoài ra, em xin được chân thành cảm ơn **thạc sĩ Phạm Thị Quỳnh**, giảng viên khoa Công nghệ thông tin trường đại học Sư phạm Hà Nội, nghiên cứu sinh tại Viện Công Nghệ Thông Tin và Truyền Thông, đại học Bách Khoa Hà Nội đã giúp đỡ em về mặt ý tưởng một số thuật toán và nhận xét bài báo trong đề tài này.

Cuối cùng, em xin được gửi lời cảm ơn chân thành tới gia đình, bạn bè đã động viên, chăm sóc, đóng góp ý kiến và giúp đỡ trong quá trình học tập, nghiên cứu và hoàn thành đồ án tốt nghiệp.

Hà Nội, ngày 28 tháng 05 năm 2010

Bùi Hoàng Đức

Sinh viên lớp CNPM A – K50
Viện Công Nghệ Thông Tin và Truyền Thông
Đại học Bách Khoa Hà Nội

Table of Contents

Table of Contents	1
List of Figures	3
List of Tables	4
ABBREVIATIONS TABLE.....	5
PREFACE	6
Chapter 1. INTRODUCTION	7
1.1. Service-oriented Architecture and Web Services.....	7
1.2. Web Service Composition and BPEL	11
1.3. Formal Methods and Model Checking	15
1.4. The Spin Model Checker.....	16
1.4.1. Introduction.....	16
1.4.2. Underlying models and verification process	17
1.5. Promela Language	23
CHAPTER SUMMARY	29
Chapter 2. BPEL PROCESSES VERIFICATION PROBLEM AND PROPOSED SOLUTION	30
2.1. BPEL Processes Verification Problem and Current Research Trends	30
2.2. Proposed Solution Architecture.....	34
2.2.1. Labeled Control Flow Graph	36
2.3. Proposed Algorithms	36
2.3.1. Algorithm of transforming from BPEL documents to labeled flow control graphs	36
2.3.2. Algorithm of transforming from labeled flow control graphs to Promela programs	40
CHAPTER SUMMARY	50
Chapter 3. IMPLEMENTATION.....	51
3.1. Tool Architecture.....	51
3.2. Metamodels in BVT	52
3.2.1. BPEL metamodel.....	52
3.2.2. LCFG model	53

3.2.1. Promela metamodel	54
3.3. Transformer core module – implementation of algorithms.....	57
3.4. Other features and techniques	59
CHAPTER SUMMARY	60
Chapter 4. TOOL TESTING AND METHOD EVALUATION	61
4.1. A Transformation Test.....	61
4.2. Some Screenshots of BVT.....	67
4.3. Evaluations	69
4.3.1. Evaluation of proposed algorithms.....	69
4.3.2. Evaluation of BPEL verification tool	71
CHAPTER SUMMARY	71
CONCLUSIONS AND FUTURE WORKS	72
Established Achievements.....	72
Future Works	72
Conclusion.....	73
Appendix A. Paper (Vietnamese)	74
Appendix B. Some Tools And Libraries Used In The Thesis	84
B.1. JAXB	84
B.2. JGraphT	84
B.3. JGraph.....	85
B.4. Jspin	85
Appendix C. Javadoc of Main Packages in BVT.....	85
Bibliography.....	87

List of Figures

Figure 1-1 Basic elements of SOA	9
Figure 1-2 Web service triangle architecture	10
Figure 1-3 Web service composition in orchestration style.....	11
Figure 1-4 Web service composition in choreography style	11
Figure 1-5 BPEL document structure	14
Figure 1-6 Transition relation for the sample model in Listing 1.1	20
Figure 1-7 State diagram of if-statement	26
Figure 2-1 Transformation process from BPEL to Promela and verification result.	35
Figure 2-2 Elements in a LCFG	36
Figure 3-1 BPEL Verification Tool architecture	52
Figure 3-2 Classes in model.graph package.....	55
Figure 3-3 Classes in package transformer.bl	58
Figure 3-4 Classes in package transformer.lp	59
Figure 4-1 LCFG for loanApproval process	61
Figure 4-2 BVT screenshot – opening a BPEL document.....	67
Figure 4-3 BVT screenshot – exporting a graph in many file formats	67
Figure 4-4 BVT screenshot - transformation from a BPEL process into a Promela program	68
Figure 4-5 BVT screenshot - saving the generated Promela program to a text file..	68
Figure 4-6 BVT screenshot - verifying the generated Promela program with a property	69
Figure B-1 JAXB Architecture	84

List of Tables

Table 1.1 Comparison of traditional and service-oriented programming	9
Table 1.2 Basic activities descriptions	13
Table 1.3 Structured activities descriptions	13
Table 1.4 Numerical data types in Promela	23
Table 2.1 Current research trends	33
Table 2.2 Mapping structured activities and <process> element to LCFG constructs	40
Table 2.3 Resolved mapping rules for BPEL elements	41
Table 2.4 XML-Promela Data Type Conversion Rules.....	44
Table 2.5 Mapping rules for LCFG constructs that represent structured activities ..	48
Table 3.1 Types of nodes in LCFG	54
Table 4.1 Generated Promela program for loanApproval process	64
Table 4.2 Default verification of loanApproval Promela program with Spin	65
Table 4.3 Verification result for a property of loanApproval process	66
Table 4.4 Solutions comparison	70

ABBREVIATIONS TABLE

No.	Abbreviation	Full name	Explanation
1.	BPEL	Business Process Execution Language	XML-based language, designed for composing web services
2.	BVT	BPEL Verification Tool	Tool that realizes the verification method and data structures
3.	DOT		
4.	JAXB	Java Architecture for XML Binding	
5.	JAXP	Java API for XML Processing	
6.	LCFG	Labeled Control Flow Graph	Directed graph, representing the flow of BPEL process
7.	LTL	Linear Temporal Logic	A kind of logic, can be used to represent properties of a system
8.	Promela		Programming language for specifying models
9.	SOA	Service-oriented Architecture	
10.	SOAP	Simple Object Access Protocol	Protocol for web services interaction
11.	Spin		A popular model checker
12.	WSDL	Web Service Description Language	XML-based language, designed to describe provided services
13.	XML	Extensible Markup Language	

PREFACE

Today, service-oriented architecture (SOA) is advanced software architecture SOA and includes a set of principles to build flexible distributed applications more efficiently. In SOA implementation technologies, web services are supported by most of major vendors. Web services can be composed using existing web services and BPEL is the de-facto language to build those composite web services. The composition of web services using BPEL processes is error-prone so its correctness needs to be verified.

In this thesis, I propose a method for verifying BPEL processes. The method can be integrated into SOA software development process to find errors at early stage and create highly reliable composite web services.

The objectives of this thesis are as follows:

- (1) Studying Service-oriented Architecture, Web Service and Business Process Execution Language (BPEL).
- (2) Studying formal methods in software development, especially model checking method and model checkers.
- (3) Developing a method for verifying business processes written in BPEL using the Spin model checker.
- (4) Developing data structures for processing BPEL processes and labeled control flow graph and generating Promela language source code in a systematic manner.
- (5) Developing a tool that realizes the above method and data structures.
- (6) Evaluating the above method and testing the tool.

Chapter 1. INTRODUCTION

In this chapter

- Service-oriented architecture and web services.
- Overview of formal methods and model checking.
- Main components in Promela language.
- The Spin model checker and its underlying model and verification process.

In modern fast changing business environment, enterprise application need to be more flexible so it can easily change when business process changes. Moreover, when the software industry develops, many standards and technologies emerge. Software developed on different technologies is often not compatible to each other. The integration of enterprise applications which are developed on different platforms or technologies becomes a significant challenge.

The service oriented software architecture provides interoperability for software development. SOA can be implemented using any service-based technology such as CORBA, EJB and Web Service. In those technologies, Web Service, which is an open standard, is widely supported and adopted.

Another essential requirement of software development is the correctness and reliability. Applying formal methods which are mathematics and theoretical models to software development lifecycle can create very high quality software. Model checking is a formal method in a software verification that systematically checks whether a property holds on a model of software.

1.1. Service-oriented Architecture and Web Services

Service-oriented Architecture is a set design principles in which the building elements are services. Each service exposes a set of functionalities that other services can use.

According to [1], there are many differences between traditional programming (procedure, object-oriented programming methodologies). The key differences are shown in the table below.

	Traditional programming	Service-oriented programming
Comparisons on software development strategies		
Goal	Know programming language constructs and apply them for problem solving	Know the overall application architecture and how to compose applications using existing component services.

Focus	On hardware and software interface, system interaction, low-level programming techniques, and low-level reusability (rather than applications).	On service specification, application composition, human/computer interactions, system interaction, and software modules, applications domains, and high-level reusability.
Contents	The syntax of the programming language, with an emphasis on the construction of program modules.	The service-oriented computing principles and the use of existing service to compose applications.
Acquiring order	Learn programming language constructs, followed by architecture design.	Learn software architecture design followed by workflows and services.
First development essay	Develop application from scratch.	Develop applications by composition using existing services in a service-oriented computing infrastructure.
Comparisons on the target software characteristics		
Overall process	For example, object-oriented design by first identifying data, classes, or associated methods.	Software development by identifying loosely coupled services and composing them into executable applications.
Level of abstraction and cooperation	Application development is software delegated to a single team responsible for the entire lifecycle of the application. Developers need to have programming knowledge and some domain knowledge.	Development is delegated to three independent parties: application builder, service provider, and service broker. Builders understand application logic and may now know how services are implemented. Providers develop services but may not know the applications. Brokers know and broadcast a large number of services.
Code sharing and reuse	Code reuse through inheritance of class members and through library functions. Often these are platform dependent.	Code reuse at service level. Services have standard interfaces and are published on repositories on the Internet. They are platform-independent and can be searched and remotely accessed. Service brokers enable systematic sharing of services.
Dynamic binding and	Associating a name to a method at runtime. The method	Binding a service request to a service can be done at the design time or at

decomposition	must have been linked to the executable code before the application is deployed	runtime. The services can be discovered after the application has been deployed. This feature allows an application to be composed (and re-composed) at runtime.
Methodology	Application development by identifying tightly coupled classes. Application architecture is hierarchical based on the inheritance relationships.	Application development by identifying loosely coupled services and composing them into executable applications.
System maintenance	Users need to upgrade their software regularly. The application has to be stopped to perform the upgrading.	The service code resides on service providers' computers. Services can be updated without users' involvement.

Table 1.1 Comparison of traditional and service-oriented programming

There are 3 basic elements in SOA:

- Service Broker.
- Service Consumer.
- Service Provider.

An execution scheme should be: First, a service provider advertises its service contract information to service brokers. A service consumer finds needed service providers by contacting service brokers. After finding necessary providers, the consumer will use the services from the provider by interacting with it directly.

Figure 1-1 (from [2]) illustrates this scheme.

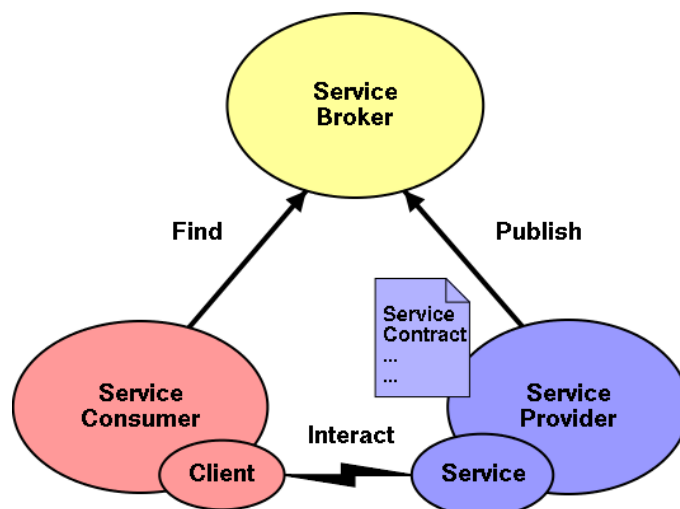


Figure 1-1 Basic elements of SOA

There are many technologies for SOA implementation. Some examples are CORBA, EJB, and Web service. The most popular and flexible technology is the Web service. Some reasons for the popular of Web service technology are open standards, interoperability and platform independence.

Definition 1.1 Web Service

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Figure 1-2 (from [3]) illustrates the Web Service architecture with WSDL and SOAP: A service provider provides its service contract in a WSDL document and advertises it to a discovery agency using UDDI_save_xxx action of UDDI protocol. A service consumer will find compatible service using UDDI_find_xxx action of UDDI protocol. After finding needed web services, the consumer will interact with the web service by XML documents over SOAP protocol.

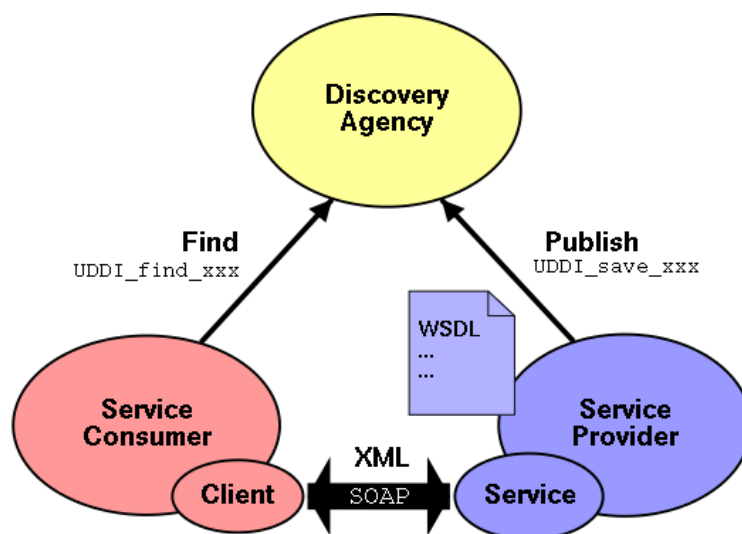


Figure 1-2 Web service triangle architecture

Web services are accessed using open standard text-based protocols such as Hypertext Transfer Protocol (HTTP) and XML. SOAP is a simple XML-based protocol to let applications exchange information over HTTP. Specification of SOAP can be found at [4]

1.2. Web Service Composition and BPEL

A web service can be created by either developing from scratch or composing other existing web services following some coordination plan. “A coordination plan represents the coordinated execution of services.” [1]

There are 3 main approaches for composition of services: Orchestration, Choreography and Collaboration.

In orchestration, web services are controlled by a central coordinator following a business process. Each web service taking part in the process does not need to know about its role in the process. In choreography, each involved web service knows the time it operates and which service it interacts with. Figure 1-3 and Figure 1-4 illustrate that approach of web service composition.

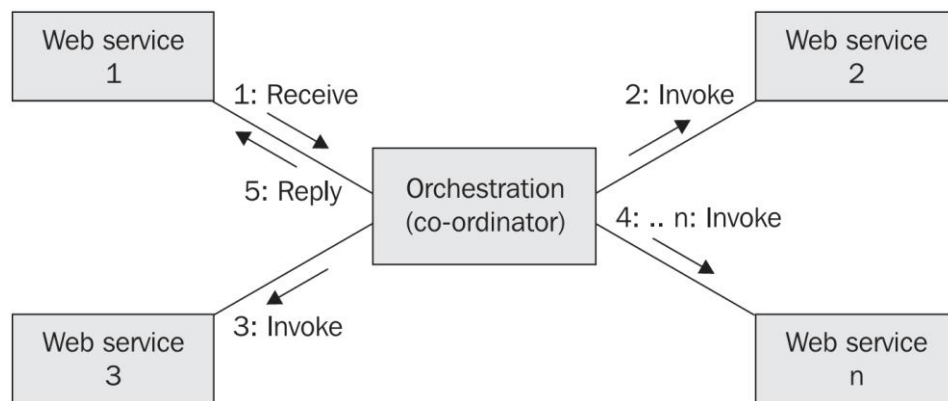


Figure 1-3 Web service composition in orchestration style

The third type of web service composition is collaboration. In this style, web services involving in a business process partly knows its role in the process but are still controlled by a central coordinator. This kind is a hybrid of the orchestration and choreography styles. However, this style has not been fully developed and supported by software vendors. [1]

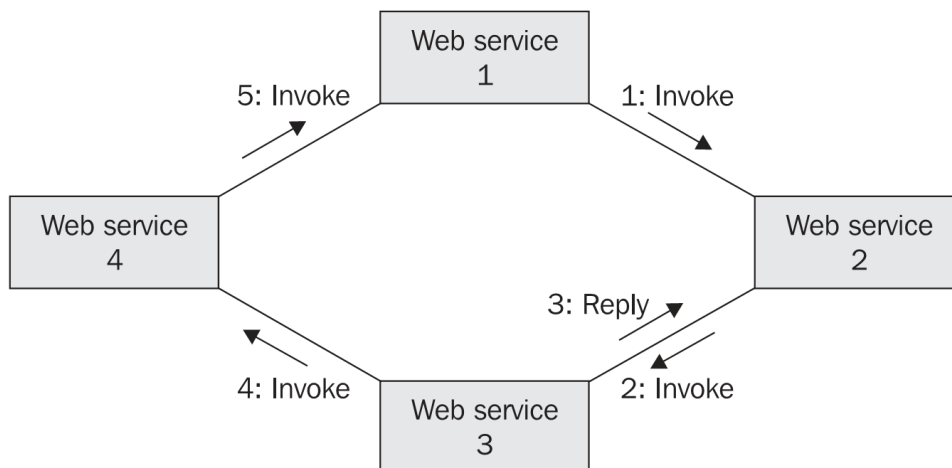


Figure 1-4 Web service composition in choreography style

In those approaches, the orchestration approach with the BPEL language is the most popular in today's real-life applications. Many major vendors such as Oracle, IBM and SAP produce BPEL engines such as Oracle BPEL Process Manager, IBM Process Manager [5]

Business Process Execution Language (BPEL) is an XML language for the composition of web services. The program written in BPEL is exposed as a Web service to the outside world. The involved web services are controlled by a central engine.

Every BPEL process is specified in an XML document, commonly with “.bpel” extension.

Figure 1-5 shows main parts of a BPEL documents: extensions, imports, partnerLinks, messageExchanges, variables, correlationSets, faultHandlers, eventHandlers and activity.

A BPEL process consists of activities. There are 2 types of activities: basic activities and structured activities. [6]

There are 11 basic activities: `invoke`, `receive`, `reply`, `assign`, `throw`, `rethrow`, `wait`, `empty`, `exit`, `validate` and `extensionActivity`

There are 7 structured activities: `sequence`, `if`, `while`, `repeatUntil`, `pick`, `flow` and `forEach`.

Basic activity	Brief explanation
<invoke>	Call an operation of a web service specified in a partner link and receive feedback from it.
<receive>	Receive data from a web service specified in a partner link.
<reply>	Send a response to a request previously accepted through an inbound message activity such as the <receive> activity.
<assign>	Copy data from one variable to another, to construct and insert new data using expressions and to copy endpoint references to and from partnerLinks.
<throw>	Signal an internal fault explicitly.
<rethrow>	Propagate faults, used in fault handlers to rethrow the fault they caught.
<wait>	Specifies a delay for a certain period of time or until a certain deadline is reached.
<empty>	Does nothing.
<exit>	Immediately end the business process instance.

<validate>	Validate the values of variables against their associated XML and WSDL data definition.
<extensionActivity>	Extend WS-BPEL by introducing a new activity type.

Table 1.2 Basic activities descriptions

Structured activity	Brief explanation
<sequence>	Define a collection of activities to be performed sequentially in lexical order.
<if>	Select exactly one activity for execution from a set of choices.
<while>	Define that the child activity is to be repeated as long as the specified <condition> is true.
<repeatUntil>	Define that the child activity is to be repeated until the specified <condition> becomes true.
<pick>	Wait for one of several possible messages to arrive or for a timeout to occur.
<flow>	Specify one or more activities to be performed concurrently.
<forEach>	Iterates its child scope activity exactly N+1 times where N equals the <finalCounterValue> minus the <startCounterValue>.

Table 1.3 Structured activities descriptions

Partner links

Partner links in BPEL model services with that the business process interacts. Each partner link belongs to a partner link type. Each partner link type defines roles for the process and the interacting service. Each role contains a port type and each port type defines some operations. Thus, the actions of sending and receiving data to and from other web services are performed over partner link. Each partner link can be considered as a contract between the business process and an outside service.

Accessing variable with XPath expressions

The query and expression language used in BPEL is specified by the `queryLanguage` and `expressionLanguage` attributes of `process` element. BPEL specification uses XPath 1.0 as the default query and expression language and the default value for the above attributes is `urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0`

“An enclosing element is defined as the parent element in the WS-BPEL process definition that contains the Query or Expression.” [6].

The expressions written in Query/Expression language should be only able to access the objects in the scope of the Enclosing Element’s enclosing activity.

The result of the evaluation of a WS-BPEL expression or query will be one of the following:

- A single XML infoset item
- A collection of XML infoset items
- A sequence of Character Information Items for simple type data
- A variable reference

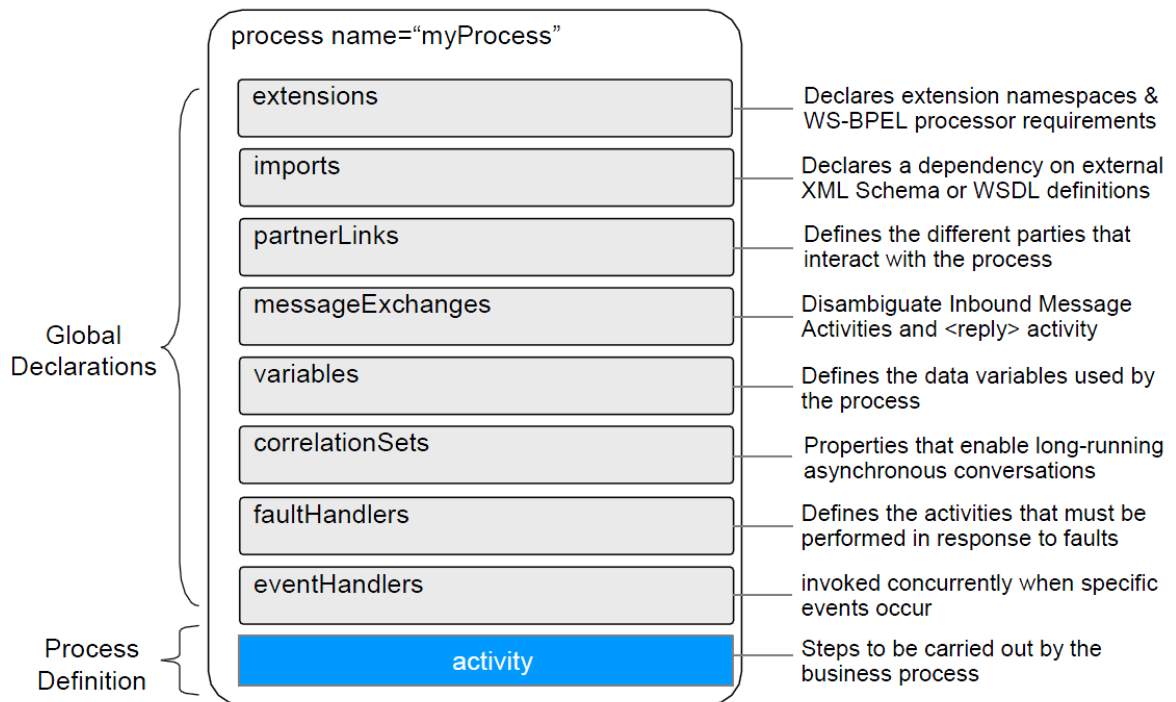


Figure 1-5 BPEL document structure

To access BPEL variables, we will use XPath variable bindings. For variables declared using an element, we can access the variable using XPath as an XML document with the document element be the element type of the variable. For variables declared using a complex type, we can access it as a node-set XPath variable with one member node containing the anonymous document element that contains the actual value of the variable. The XPath variable binds to the document element. For WS-BPEL `messageType` variable will be expressed as a series of variables, each of which corresponds to a part in the `messageType`. To access to a part, because WSDL message parts are always defined using either an XSD element, an XSD complex type or a XSD simple type, we will use the part name followed by a dot "." and the variable name, then access each part as a "normal" element. Simple type variables are accessed directly as either an XPath string, boolean or float object. `xsd:boolean` variable are manifested as an XPath boolean object; `xsd:float`, `xsd:int` and `xsd:unsignedInt` are manifested as an XPath float object; other XML Schema types must be manifested as an XPath string object.

Synchronization dependencies

In `<flow>` activity, synchronization dependencies can be specified by `<links>` between inner activities. Each link is a tri-stated flag which can hold one of the 3 values: *true*, *false* or *unset*. Each link points from an activity which is the *source* of the link into another activity which is the *target* of the link.

Each inner activity is ready to start whenever the flow activity starts. However, if the activity is the targets of one or more links, it will not be executed until the `joinCondition` which specifies a logical expression attribute is evaluated to true. If the `joinCondition` is omitted, its logical expression is the OR-logic of all incoming links i.e. the activity will be executed if at least one incoming link is *true*. If the `joinCondition` is evaluated to false, a fault will be thrown unless the `suppressJoinCondition` attribute is *yes*. The default value of `suppressJoinCondition` is *yes*. The thrown fault will be handled by a fault handler.

1.3. Formal Methods and Model Checking

Formal methods in software engineering can be defined as follows:

Definition 1.1 Formal methods [7]

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner.

A method is formal if it has a sound mathematical basis, typically given by a formal specification language.

According to [7], traditional approaches have some deficiencies such as: contradictions, ambiguities, vagueness, incompleteness and mixed levels of abstraction. The use of mathematics in software development can solve those problems. The formal syntax of a specification language such as notations of set or logic theory makes requirements to be interpreted in one way, so it eliminates ambiguity of natural languages. A system specification specified in a formal specification language can be checked automatically by software to find contradictions. We also are able to prove statements like theorems or system's properties mathematically. Therefore, consistency is ensured and correctness can be verified.

A disadvantage of formal methods is that they require trained staff that is able to work with mathematical notations, so development cost may increase. Formal methods do not replace all traditional development approaches. We also should include documents in natural language next to the formally specified system

requirements to help readers comprehend the system. Software testing and software quality assurance must continue to ensure the quality of the results.

Simulation, testing, deductive verification, and model checking are the principal validation for complex systems [8]. Simulation is performed on an abstraction or a model of the system while testing is conducted on a real instance of the system. Deductive verification refers to techniques of proving the correctness of systems using axioms and proof rules. Model checking uses an exhaustive search of the state space of the system to verify that if some specification is true or not. One benefit of model checking over deductive verification is that it can be performed automatically.

Applying model checking to a design consists of several tasks [8]:

- **Modeling:** System design must be expressed in a formal language that is accepted by a model checking tool. One of the challenges of model checking is to construct a model that is practical to verify but sufficient and faithful to represent the verified program [9].
- **Specification:** Properties that the design must satisfy must be stated. It is common to use temporal logic that can assert the state of the system over time. A significant issue in specification is completeness because we cannot determine whether the given specification covers all the properties that the system should satisfy.
- **Verification:** The result of a verification process is whether the properties in specification are held by the model. In case of a negative result, an error trace will be provided. Users will analyze this trace to find out the cause which may be in the original design, in modeling process or incorrect specification.

1.4. The Spin Model Checker

1.4.1. Introduction

Spin is a very popular model checker. It verifies properties of a system using model checking method. Systems to be verified are described in Promela (Process Meta Language), which supports modeling of asynchronous distributed algorithms as non-deterministic automata.

To verify properties of a system, users use assertions or Linear Temporal Logic (LTL) formulae. An assertion is a statement consisting of the keyword **assert** followed by an expression. LTL formulae are more general than assertions. It adds to the propositional calculus temporal operators: always [], eventually \diamond and until U; so LTL formulae express propositions with time factor. Spin will check if the properties. In simulation mode, if an assertion is evaluated to false, the program will terminate and Spin will show an error message. In verification mode, Spin will

show whether LTL formulae or assertions are violated and the depth that the violation happens. A trail of the computation is also recorded.

In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. There are 3 modes of simulation: random, guided by a trail file and interactive with users.

- In **interactive simulation mode**, users have to choose one of the statements to execute.
- In **random simulation mode**, Spin will choose which statements to execute, i.e. run in a random path. With only a filename as an argument and no option flags, Spin performs a random simulation of the model specified in the text file.
- In **guided simulation mode**, Spin will use a trail file that contains an encoded sequence of transitions to decide which statements will be executed and by default has the name of the model file with *trail extension*. This mode is useful particularly when the verification process fails and we want to find out exactly what happened.

Users can specify the level of output detail in the arguments of the spin.

In order to verify a Promela model, we perform 3 steps:

- Generate a verifier which is a C program in file pan.c from the Promela model.
- Compile the verifier using a C compiler with POSIX compliant C library.
- Run the executable verifier. The result is a report that there is no error found or else that some computation contains an error.

One of the goals of a model checker is to support system engineer to find where an error happens. Spin does it by maintaining data structures that are used to reconstruct a computation that leads to an error. The data is saved into a trail file and users can use it in guided simulation mode of Spin to analyze the cause of the error.

1.4.2. Underlying models and verification process

Spin model checker searches through state space of a model for a counterexample to the correctness specifications, following model checking method. “The state space of a program is the set of states that can possibly occur during a computation.” [9]. The set of values of variables and location counters makes a state of a program. A computation is a sequence of states that begins with an initial state and continues

with the states that occur when each statement is executed. The correctness specifications can be expressed in assertions or linear time logic formulae.

Automata theory

Formally, Spin considers state space as a Büchi automaton. Automata theory is essential to understand properties such as safety and liveness and what actually Spin is built on. With that thought in mind, in this section, I provide formal definitions of finite-state automata and Büchi automaton.

I will use some convention symbols:

- $\text{Pow}(Q)$ is the set of all subsets of Q .
- $\text{Inf}(r) = \{r \in \Sigma \mid \forall i \in \mathbb{N}, \exists j > i \in \mathbb{N}, r_j = r\}$ is the set of symbols that occur infinitely often in r .

Definition 1.1 Finite State Automaton [10]

An automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite input alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is the transition function mapping $Q \times \Sigma$ to Q . That is $\delta(q, a)$ is a state for each state q and input symbol a .

Definition 1.2 ω -Language

A ω -language is a subset of all words of infinite length over an alphabet Σ : $A \subseteq \Sigma^\omega$

Here, we may interpret ω as repeating infinitely often.

Definition 1.3 Finite ω -automaton

A finite ω -automaton M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set of states.
2. Σ is a finite alphabet.
3. δ : transition function
4. $q_0 \in Q$ is the initial state.
5. $F \subseteq Q$ is the set of final states

Definition 1.4 Determinism and non-determinism

Let $M = (Q, \Sigma, \delta, q_0, Acc)$ be a finite state automaton
 M is non-deterministic if and only if: $\delta : Q \times \Sigma \rightarrow \text{Pow}(Q)$
 M is deterministic if and only if: $\delta : Q \times \Sigma \rightarrow Q$

Definition 1.5 Run

Let $M = (Q, \Sigma, \delta, q_0, Acc)$ be a finite ω -automaton, let $\alpha \in \Sigma^\omega$. A run of M on $\alpha_1 \alpha_2 \alpha_3 \dots \in \Sigma^\omega$ is an infinite sequence of states $r = r_0 r_1 r_2 \dots \in Q^\omega$ such that

(i) $r_0 = q_0$

(ii) $r_{i+1} \in \delta(q_i, \alpha_{i+1}), \forall i = 0, 1, \dots$ - in case of nondeterministic.
 $r_{i+1} = \delta(q_i, \alpha_{i+1}), \forall i = 0, 1, \dots$ - in case of deterministic.

Definition 1.6 Büchi acceptance

We say M accepts a ω -word $\alpha \in \Sigma^\omega$ if and only if there exists a run r of M on α satisfying $\text{Inf}(r) \cap F \neq \emptyset$ i.e. at least one accept state in F has to be visited infinitely often during the run r .

The Definition 1.2 to Definition 1.6 are based on [11]

Besides the Büchi acceptance, there are other types of acceptance such as Muller, Rabin and Street acceptance.

So we can ends up with the definition of a Büchi Automaton.

Definition 1.7 Büchi Automaton

A Büchi Automata is a finite state automaton that accepts infinite strings with Büchi acceptance conditions.

Roughly, a Büchi automaton has a finite number of states and an input that makes it to perform a run of infinite number of steps can be accepted. The main difference between a ω -automaton and a finite state automaton is that the ω -automaton can accept an infinite input.

In terms of Büchi automata, safety and liveness of a system (program) can be defined as:

- Safety: The execution of corresponding Büchi automaton of the program never reaches any invalid state.
- Liveness: The corresponding Büchi automaton of the program will execute a run through valid state infinitely often.

Reachability graph and Promela semantics rules

In Spin, the state space or Büchi automaton corresponds to a global reachability graph. Spin systematically analyze all parts of it to find paths that lead program to states that are invalid or violate correctness specifications. By default, Spin stops as soon as one error path is found because the existence of one counterexample is usually enough to prove the incorrectness of a program.

Promela semantics rules determine the structure of a reachability graph. Each node in that graph represents a possible state of the model and each edge represents a single possible execution step by one of the processes in the model. Promela is deliberately designed to make the generated graph be finite, so in principle, “the complete graph can always be built and analyzed in a finite amount of time”.

Each Promela proctype corresponds to a finite state automaton (S, s_0, L, T, F). The set of states S corresponds to the possible points of control within the proctype. Transition relation T defines the flow of control. The set of final states F

corresponds to the set of valid end states in Promela model. The set of labels L is the set of basic statements in Promela which contains just six elements: assignments, assertions, print statements, send or receive statements, and Promela's expression statement.

Listing 1.1 and Figure 1-6 illustrate a proctype and the corresponding automaton model. [9]

Listing 1.1 Sample Promela Model

```

active proctype not_euclid(int x, y)
{
    if
    :: (x > y) -> L: x = x - y
    :: (x < y) -> y = y - x
    :: (x == y) -> assert(x!=y); goto L
    fi;
    printf(";%d\n", x)
}

```

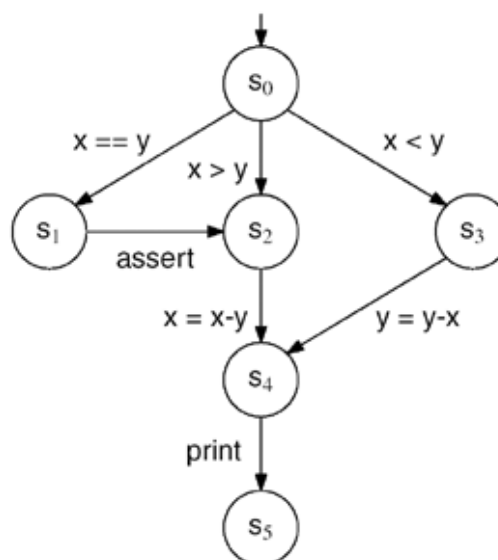


Figure 1-6 Transition relation for the sample model in Listing 1.1

There are 6 basic statements: *assert*, *assign*, *condition*, *printf*, *receive* and *send*; and the semantics engine executes the system in a stepwise manner: selecting and executing one basic statement at a time.

Spin uses a semantics engine to simulate the execution of a model. The semantics rules used in the engine determines the structure of reachability graph. The semantics engine operates on abstract objects that correspond to elements in Promela language.

To explain how the semantics engine operate, I extract definitions in [12]

Definition 1.8 Variable

A variable is a tuple $(name, scope, domain, inival, curval)$ where
name is an identifier that is unique within the given *scope*,
scope is either global or local to a specific process.
domain is a finite set of integers.
inival, the initial value, is an integer from the given *domain*, and
curval, the current value, is also an integer from the given *domain*.

Definition 1.9 Message

A message is an ordered set of variables

Definition 1.10 Message Channel

A channel is a tuple $(ch_id, nslots, contents)$ where
ch_id is a positive integer that uniquely identifies the channel,
nslots is an integer, and
contents is an ordered set of messages with maximum cardinality *nslots*.

Definition 1.11 Process

A process is a tuple
 $(pid, lvars, lstates, initial, curstate, trans)$ where
pid is a positive integer that uniquely identifies the process,
lvars is a finite set of local variables, each with a *scope*
that is restricted to the process with instantiation number *pid*.
lstates is a finite set of integers (see below),
initial and *curstate* are elements of set *lstates*, and
trans is a finite set of transitions on *lstates*.

Definition 1.12 Transition

A transition in process *P* is defined by a tuple
 $(tr_id, source, target, cond, effect, prty, rv)$ where
tr_id is a non-negative integer,
source and *target* are elements from set *P.lstates* (i.e., integers),
cond is a boolean condition on the global system state,
effect is a function that modifies the global system state,

prty and rv are integers.

Definition1.13 System State

A global system state is a tuple of the form

(gvars,procs,chans,exclusive,handshake,timeout,else, stutter) where

gvars is a finite set of variables with global scope,

procs is a finite set of processes

chans is a finite set of message channels,

exclusive, and handshake are integers,

timeout, else, and stutter are booleans.

Basically, semantics engine execute in a stepwise manner. It choose one in a set of executable statements of the model, if there is no handshake (rendezvous), it will apply the effect of the statement on the model and change the state of the process that contains the selected statement; otherwise, it will handle the handshake. The detail of the algorithm is in the chapter 7 of [12].

Search algorithms

In verification mode, after constructing the reachability graph, the Spin will use graph search algorithms, typically, depth-first search or breath-first search. Each algorithm has its own drawback but in typical application, the depth-first search is the most effective choice and is chosen by default.

I will not describe the depth-first and breath-first search algorithms applied in Spin model here because they are like basic algorithms. If you want to know more about them, please refer to [12].

It's proved that the depth-first search algorithm on Spin models always terminates within a finite number of steps.

Because the search algorithms visit every reachable state, so we can use them to check safety property quite simply by embed a checking instruction into the algorithm. It can identify all possible deadlocks and assertion violations. The main issue is what operations that the algorithm should perform when a violated state property is found. The Spin implementation saves the execution trace from the initial state the violated states. Though the depth-first algorithm need not to find the shortest possible counterexamples, it is sufficient for engineers to look at last few steps in the execution sequence to find out the nature of a property violation.

The basic depth-first algorithm can be extended to check the liveness properties that are expressed in linear temporal logic. It is proved that an acceptance cycle in the graph exists if and only if there exists one reachable accepting state from the initial state and one of the accepting states is reachable from itself [12]. To check the liveness property, the Spin verifier first finds a reachable accepting state and then finds a cycle that contains that state.

1.5. Promela Language

Input models of the Spin model checker are described in Promela language. Promela is a language for system modeling so it is like other programming languages (e.g. C and Pascal) but has some components that exist in others and also lack of some familiar components.

Promela grammar rules are described formally in [13]. Those rules will be used in this thesis to construct Promela code generator.

According to [9], the main elements in a Promela program are as follows:

1. Built-in data types

There are data types in Promela. They can be grouped into numerical data types (bit, bool, byte, short, int, unsigned), channel data type (chan), user-defined data types (mtype, typedef), 1-dimensional array data type.

The value range of numerical data types is shown in Table 1.4

Type	Values	Size (bits)
bit, bool	0, 1, false, true	1
byte	0.. 255	8
short	-32768.. 32767	16
int	$-2^{31} .. 2^{31}-1$	32
unsigned	$0..2^{n-1}$	≤ 32

Table 1.4 Numerical data types in Promela

The bit and bool data types are equivalent data types and are intended to make programs more readable.

Promela does not have some familiar data types such as character, string, floating-point data types. Floating-point numbers are not needed because the exact values are not important in models. However, we can use them in embedded segments of C code.

All variables are initialized by default to zero.

There is no explicit data type conversion in Promela. Arithmetic operations are always performed by first implicitly converting all values to int; upon assignment, the value is implicitly converted to the type of the variable. When the value is larger than the range of the assigned variable, a truncation will occur and an error message will be printed but the program may continue because PIN leaves it up to user to decide whether the truncated value is meaningful or not.

`_pid` variable indicates the id of the process that contains that variable. `_pid` of a process begins from 0.

The following operators and functions can be used to build expressions:

+	-	*	/	%	>
>=	<	<=	==	!=	!
&		&&		~	>>
<<	^	++	--		
len()	empty()	nempty()		nfull()	full()
run	eval()	enabled()		pc_value()	

2. User-defined types

A user-defined structured data type can be created by using the following syntax:

```
typedef structured_data_type_name{
    subfield1_type subfield1_name;
    subfield2_type subfield2_name;
    ...
}
```

To declare a structured variable we use the syntax: `structured_data_type_name variable_name;`

In order to access to a subfield of the variable, we write variable's name and subfield's name, separated by a dot: `variable_name.subfield1`.

A typedef type must be prior to any use of that type; otherwise, a syntax error will occur.

3. Control statements

There are **five control statements**: sequence, selection, repetition, jump and unless. The semicolon is the separator between statements rather than a command terminator.

In order to understand the Promela semantics, I'd like to explain the concept of control point. A control point is an address of an instruction. For examples, in the following sequence of statements:

```
int x=1;
int y=2;
z=x+y
```

there are 3 control points, one before each statement, and the program counter (pc) that indicates the current execution point of the program can be at any one of them.

The syntax of **if-statement** is as following:

```
if
:: guard1 → stmt1; stmt2
:: guard2 → stmt3; stmt4; ...
:: ... → ...
:: else → ...
fi
```

An if-statement starts with the key word **if** and ends with the reserved **fi**. Between the key words, there are one or more *alternative* or *option sequence*, each consisting a double colon “;” and a sequence of statements in which the first statement is called *guard*.

If there is only one guard is evaluated to true then the followed statements will be executed. If there is more than one guard being true, then one of those alternatives will be selected randomly as the program is converted into a non-deterministic omega-automaton by SPIN. The else guard will be selected if and only if all other guards are evaluated to false. In the case that all guards are false, the if-statement is blocked until there exists one guard become true.

The sequence of statements following a guard can be empty. SPIN provides user the **skip** keyword that always evaluates to true like **true** or (1).

The if-statement is not atomic, so interleaving is possible between the guard and the followed sequence of statements.

The arrow in an option sequence has the same meaning with the semi colon “;” as a statement separator and is intended to emphasize the role of the guard.

The Figure illustrates the semantics of an if-statement.

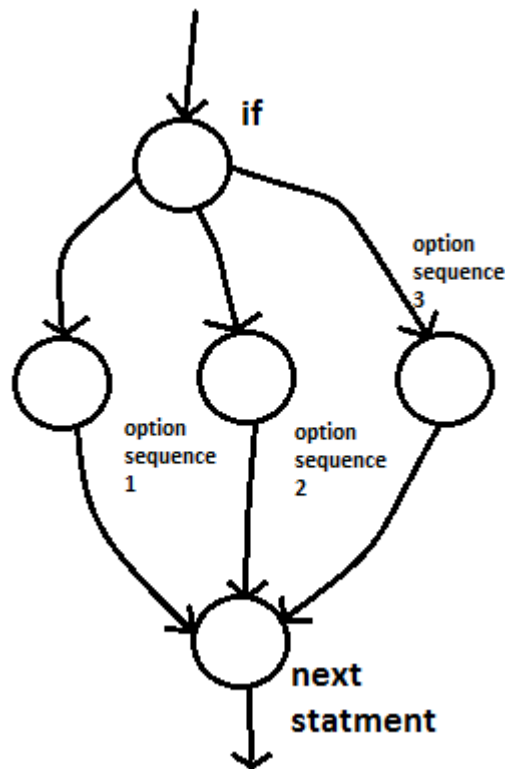


Figure 1-7 State diagram of if-statement

4. Repetitive statements

Do-statement is the only one repetitive statement in Promela. The syntax of the do-statement is almost the same as that of the if-statement:

```

do
:: guard1 → stmt1; stmt2; break
:: guard2 → stmt3; stmt4; ...
:: ... → ...
:: else → ...
od
  
```

If there is only one guard evaluating to true, the followed sequence of statements is executed. If there are two or more true guards, SPIN will choose one of them randomly and run the followed option sequence. If there is no guard being true then the do-statement is block until there is at least one guard become true.

After the option sequence is executed, the program counter will move to the point right before the **do** keyword. In order to exit from the do-statement, we will use the **break** keyword, which is not a statement but an indication that the program counter will move from current location to the statement following the **od**.

5. Jump statements

Users can use **goto**-statement to move the program counter to any label that is an identifier followed by a colon and points to a control point. There is no control point in front of an alternative in an **if**- and **do**-statement but before **if** or **do** keyword.

6. Array

Users can define an one-dimensional array by using the syntax:

```
Element_data_type array_name[number_of_elements]
```

An array is indexed from 0 to (number_of_elements-1). An error will occur in simulation and verification process if the index is out of an array's bounds.

7. The preprocessor

SPIN calls a text-based preprocessor before processing the Promela source file. *Text-based* means that the processor treats source code as pure text without taking into account any language-related element.

By using preprocessor, we can include a file: **#include** "some_file.h" and to declare a symbolic constant: **#define** CONSTANT 4;

#define is also used to declare expressions which are used in the correctness specification: **#define** accepted (approval.accept==true)

Promela provides users means to define reusable segment of codes by using Inline construct. The inline construct and macros are almost identical.

8. Channels

Channels in Promela are a structure that is not associated with processes. A process can send and receive information to and from a channel. A channel is declared by using an initializer that contains the channel capacity and the types of fields in messages.

```
chan channel_name = [capacity] of { typename, ..., typename }
```

The capacity must be a non-negative integer constant. The message type can be considered as a structure of fields: The type of each field is declared as a typename and the number of fields is the number of type names. An array cannot be a message field but we can include the array into a structure (typedef) and use it as a field in the message type.

All channel variables have the type **chan** and must be initialized before using. A channel variable refers to a channel that is created by an initializer, so it can appear in assignment statements or parameters to a **proctype** or **inline**.

After some experiments, I concluded that messages are transferred via channels as a structure of numerical types: Structured variables in a sent message will be decomposed into numerical fields, if there is no value for the field, it will be assigned to zero; and numerical values in received message will be composed into structures and assigned into received variables according to the channel's declaration.

If the capacity is zero, we have *rendezvous* or *synchronous* channels; if the capacity is more than zero, we have *buffered channels*.

To send and receive a message to and from a channel, we use the following syntax:

```
channel_name!expression, expression,... (send statement)
channel_name?variable, variable, ... (receive statement)
```

The number and types of expressions in the send statement, i.e. the structure of the sending message, *should* match those in the channel's declaration. If the expressions in send statement are variables, they will be evaluated first and their values will be transferred through the channel. In the receive statement, the received values are assigned into the listed variables. A receive statement will be blocked until a message is available on the channel.

If the number and types of expression in send and receive statement are not match with the channel's declaration, an unexpected result will occur.

A **buffered channel** contains a queue whose size is the capacity of the channel. The send statement is executable if and only if the buffer is not full. The execution of the statement will put a message into the tail of the buffer. The receive statement is executable if and only if the buffer is not empty. The execution of the statement will remove a message at the head of the buffer and assign its value to the variable in the receive command.

In my algorithms, I used mostly **rendezvous channels** so I will explain more about them. A rendezvous channel has a buffer with size of 0. Data transfer on rendezvous channels is synchronous and is executed as a single atomic operation. Suppose that there are one sender (process that contains a send operation) and one receiver (process that contains a receive operation) sharing a rendezvous channel. When the location pointer (process pointer) of the sender is at the send statement, the sender is said to offer a rendezvous. If there is no matching receive statement, the send statement will be blocked, and the same situation happens with receive statement with no matching send statement. When both location pointers of the sender and receiver are at matching send and receive operations, the rendezvous that is offered by the sender is accepted by the receiver: the location pointers of both processes

move to next statements and the values in the send statement are transferred to the corresponding variables in the receive statement. There is no operation that could happen between the execution of the send statement and the receive statement.

CHAPTER SUMMARY

In this chapter, I have presented key concepts needed for the following chapters: Service-oriented architecture, web service, web service composition; main components of Business Process Execution Language and Promela language; the Spin model checker – its features and underlying models.

Chapter 2. BPEL PROCESSES VERIFICATION PROBLEM AND PROPOSED SOLUTION

In this chapter:

- Description of the problem of verifying BPEL processes.
- Overview of the current approaches done by researchers and professors on the world.
- Proposed solution architecture for the problem.
- Algorithms in the architecture.

2.1. BPEL Processes Verification Problem and Current Research Trends

The general problem of verifying BPEL processes can be described in detail as following:

- Let B is a BPEL process which is written in BPEL language.
- Let P is a property specified on the process B.
- Check whether B satisfies the property P; if no, provide a path to the state that does not satisfy the property P

Some common properties are:

- (1) General properties such as: Safety (the corresponding Büchi automaton does not move to invalid state) or Liveness (the execution of the corresponding Büchi automaton visit final states infinitely often).
- (2) Particular properties such as: For an arbitrary loan request, does there exist two cases in which the request is approved in one case and the request is refused in the other case? In model checking method, properties of this type are described in Linear Temporal Logic formulae.

Researches about model checking of BPEL processes have been conducted in different approaches by researchers around the world. Table 2.1 gives us an overview of current search approaches.

In the table, the symbol “ \rightarrow ” can be read as “is transformed to” or “is translated to”.

Basing on [14] and Table 2.1, common approaches are:

- Using Petri net theory: A BPEL process is translated into a Petri net (that includes workflow net and colored Petri net) and then perform verification on it.

- Using model checker:
 - A BPEL process is translated into a guarded automaton, extended finite-state automaton or a finite state machine; and the automaton is then translated into an input program for a model checkers such as SMV [15] and NuSMV [16].
- Using process algebra: a BPEL process is translated into a process algebra structure, calculus for communication systems, LOTOS or pi-calculus.
- Using abstract state machine theory: a BPEL process is translated into an abstract state machine.
- Using automaton theory: BPEL processes are transformed into annotated deterministic finite automata.

No.	Name	Author	Institution	Country	Year	Published in	Methodology Description
1	Model-based Verification of Web service Compositions [17]	Howard Foster, Sebastian Uchitel, Jeff Magee and Jeff Kramer	Department of Computing, Imperial College London.	United Kingdom	2003	Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE)	BPEL is transform to Finite State Process (a kind of process algebra) that can be verified
2	Describing and Reasoning on Web Services using Process Algebra [18]	G.Salaun, L.Bordeaux and M.Schaerf	University of Roma	Italia	2004	Proceedings of IEEE International Conference on Web Services	Using Process Algebra to describe and compose web services at an abstract level.
3	Analysis of Interacting BPEL Web Services [19]	Xiang FU, Tefvik Bultan and Janwen Su	Department of Computer Science. University of California. Santa Barbara	United States	2004	Proceedings of the 13th International World Wide Web Conference	BPEL → Guarded Automaton → Promela Message types are extracted from WSDL files Handle XML based data manipulation using guarded automata with guards expressed as XPath expressions. Each guarded automaton → one Promela process Message type → an MSL type declaration → typedef in Promela Strings: used as constants → mtype
4	Model Checking Interactions of Composite Web Services [20]	Xiang FU, Tefvik Bultan and Janwen Su	Department of Computer Science. University of California.	United States	2005	UCSB Computer Science Department Technical Report 2004-2005	Tackling 2 challenges: 1) asynchronous messaging 2) rich data representation (XML) and data manipulation (XPath) Handling data using Model Schema Language BPEL → Guarded Automaton → Promela
5	Transforming BPEL to Petri nets [21]	S. Hinz, K. Schmidt, and C. Stahl	Humboldt-University of Berlin Institute of Informatik	German	2005	Proceedings of the 3rd International Conference on Business Process Management	Transforming BPEL to Petri nets which is used in LoLA, a petri net model checking tool
6	Verifying Web Services Composition Based on Hierarchical Colored Petri Nets [22]	YanPing Yang, QingPing Tan and Yong Xiao	National University of Defense Technology Changsha, Hunan	China	2005	ACM workshop on Interoperability of Heterogeneous Information Systems (IHIS'05)	Transforming a web service composition specification into a hierarchical colored Petri net which is the input of a colored Petri-nets analysis tool such as CPN tools.
7	Model-Checking Behavioral Specification of BPEL Applications [23]	Shin NAKAJIMA	National Institute of Informatics and SORST Japan Science and Technology Agency	Japan	2006	Electronic Notes in Theoretical Computer Science 151 (2006) 89-105	Extracting behavioral specification from BPEL application program, then representing it in Extended Finite Automaton (EFA), then translating the EFA into PROMELA which is the input modeling language of the Spin model checker Focus on behavioral aspect of a process

8	LTSA-WS: a tool for model-based verification of web service compositions in Eclipse [24]	Howard Foster, Sebastian Uchitel, Jeff Magee and Jeff Kramer	Department of Computing, Imperial College London.	United Kingdom	2006	Proceedings of the 28th International Conference on Software Engineering, pages 771-774, Shanghai, China.	LTSA-WS is an extension of the Labeled Transition System Analyzer (LTSA) that can used to verify properties of composite web services
9	Towards automatic verification of web-based SOA applications [25]	Xiangping Chen, Gang Huang and Hong Mei	Key Laboratory of High Confidence Software Technologies, Ministry of Education. School of Electronics Engineering and Computer Science, Peking University. Beijing, China.	China	2008	Asia-Pacific Web Conference	composition specification → behavior model that uses the notation of UML sequence with formal semantics → Promela
10	A Methodology and a Tool for Model-based Verification and Simulation of Web Services Compositions [26]	AL-GAHTANI Ali, AL-MUHAISEN Badr and DEKDOUK Abdelkader	Department of Information and Computer Science King Fahd University of Petroleum and Minerals	Saudi Arabia	2004	IEEE International Conference on Information & Computer Science, King Fahd University of Petroleum and Minerals	transform BPEL to Promela directly, developed on .NET framework

Table 2.1 Current research trends

2.2. Proposed Solution Architecture

According to the previous section, an approach to verify a BPEL process is to leverage a model checker, i.e. translating the BPEL process into an input program to a model checker over an intermediate representation.

Here, there is some equivalence: the verification of a BPEL process is equivalent to the creation of a model of the process; the modeling process problem is then equivalent to the translation of a BPEL document to a Promela program.

In this thesis, I propose a solution, in which **I transform BPEL processes into Labeled Control Flow Graph and then to Promela programs**. The output Promela program will be the input to the Spin model checker. This is illustrated in Figure 2-1.

There are some reasons of choosing LCFG as intermediate form. It's necessary to preserve control flow is an essential part of BPEL processes into Promela programs. In my algorithms, I decode control flow of a BPEL process into a LCFG and encode it in into a Promela program. The second reason is that although it's quite redundant to bind BPEL element meta-object to LCFG and then extract it, designing algorithm basing on the control flow graph is easier and more effective than working on the tree structure of BPEL documents because LCFG represent the control flow of activities better.

Compared to other model checking tools like NuSMV, Spin is specialized for modeling distributed system which is very appropriate for BPEL because BPEL defines compositions of distributed web services.

Although LCFG is simpler than other formal models such as abstract state machine or automata used by researchers in the world, my solution is as effective as the other solutions.

```

<process name="loanApprovalProcess"
targetNamespace="http://example.com/loan-approval/"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:lns="http://example.com/loan-approval/wsd1/"
suppressJoinFailure="yes">

<import importType="http://schemas.xmlsoap.org/wsdl/"
location="loanServicePT.wsdl"
namespace="http://example.com/loan-approval/wsd1/" />

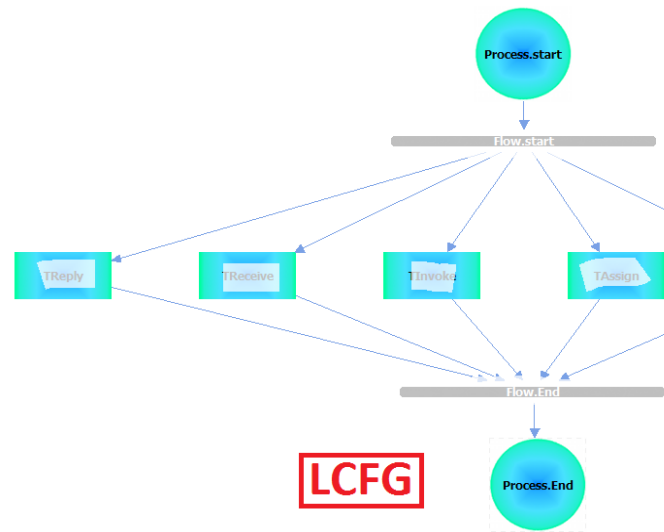
<partnerLinks>
<partnerLink name="customer"
partnerLinkType="lns:loanPartnerLT"
myRole="loanService" />
<partnerLink name="approver"
partnerLinkType="lns:loanApprovallT"
partnerRole="approver" />
<partnerLink name="assessor"
partnerLinkType="lns:riskAssessmentLT"
partnerRole="assessor" />
</partnerLinks>

<variables>
<variable name="request"
messageType="lns:creditInformationMessage" />
<variable name="risk"
messageType="lns:riskAssessmentMessage" />
<variable name="approval"
messageType="lns:approvalMessage" />
</variables>

```

BPEL process

Transformation
from BPEL to LCFG



LCFG

Transformation
from LCFG to Promela

```

mtype = { Lfalse }
mtype = { Ltrue }
mtype = { request }
mtype = { check }
mtype = { low }
mtype = { yes }
mtype = { approve }
mtype = { other }
typedef creditInformationMessage {
byte amount
}
typedef riskAssessmentMessage {
mtype level
}
typedef approvalMessage {
mtype accept
}
chan customerPL_IN = [0] of (mtype, creditInformationMessage)
chan customerPL_OUT = [0] of (mtype, approvalMessage)
chan approverPL_IN = [0] of (mtype, creditInformationMessage)
chan approverPL_OUT = [0] of (mtype, approvalMessage)
chan assessorPL_IN = [0] of (riskAssessmentMessage, mtype)
chan assessorPL_OUT = [0] of (mtype, creditInformationMessage)
creditInformationMessage request_VAR
riskAssessmentMessage risk
approvalMessage approval
mtype receive
mtype receive
mtype approve
mtype assess
mtype setmessage_to_reply
mtype assess_to_approval
proctype loanApprovalProcess () {
run TReceive();
run TInvoke();
run TAssign();
run TReply();
}

```

Promela program

Verified by Spin

Verification result

```

(Spin Version 5.2.4 -- 2 December 2009)
+ Partial Order Reduction

Full state space search for:
im - (none specified)
violations +
cks - (disabled by -DSAFETY)
invalid end states +

State-vector 100 byte, depth reached 306, errors: 0
68104 states, stored
26369 states, matched
94473 transitions (= stored+matched)
1024 atomic steps
hash conflicts: 909 (resolved)

```

Figure 2-1 Transformation process from BPEL to Promela and verification result

2.2.1. Labeled Control Flow Graph

Definition 2.14 Labeled Control Flow Graph (LCFG)

A labeled control flow graph is a directed graph that is a tuple (V, E) , in which V is a set of vertices and E is a set of edges that represent exchanges between vertices.

In set V , there is only one Start and Stop nodes. Other nodes represent activities in BPEL process. These nodes are labeled to describe important information of activities. Edges in set E represent the order of execution of vertices. We can see that Control Flow Graphs are subset of LCFGs because LCFGs allow us to represent concurrent activities.

Figure 2-2 shows elements in a LCFG.

Start and End nodes denote the starts and ends of structured activities. Conditions are denoted by TbooleanExpr nodes and basic activities are denoted by a simple node.

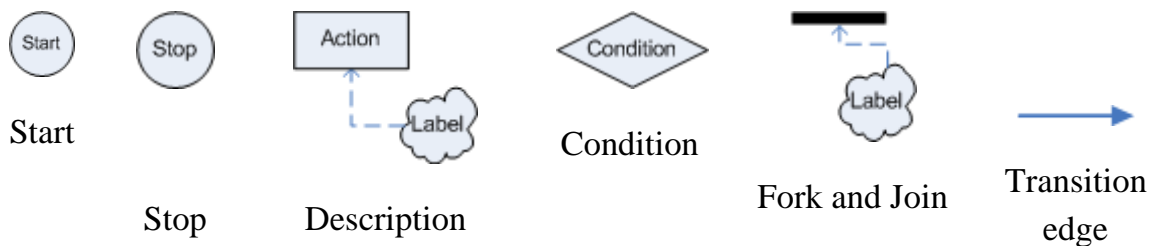


Figure 2-2 Elements in a LCFG

2.3. Proposed Algorithms

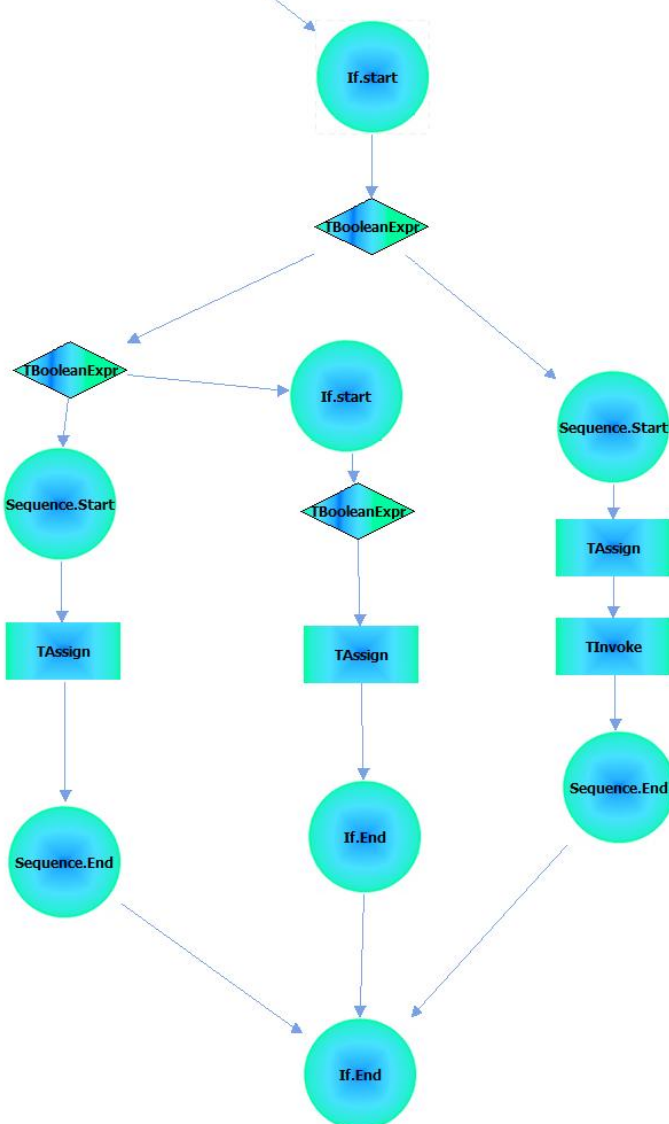
2.3.1. Algorithm of transforming from BPEL documents to labeled flow control graphs

In order to transform BPEL documents to a LCFG, I basically create mapping rules from each activity in BPEL language to a graph construct in LCFG.

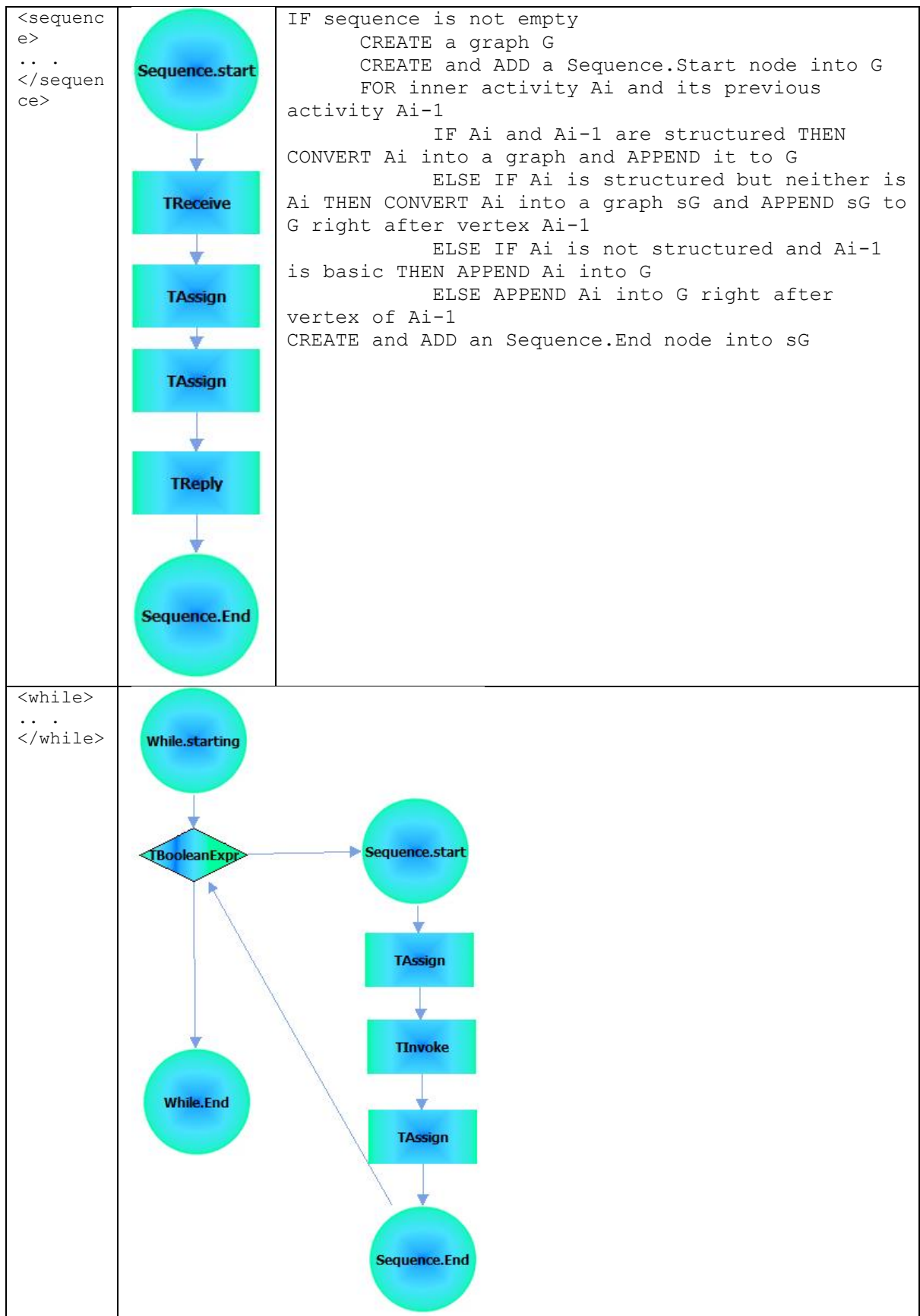
Generally, basic activities in BPEL are directly to LCFG vertices while structured activities which contain other activities are mapped to a set of vertices.

I construct a LCFG graph of a process by converting the activities of a process into a graph recursively with function `cvtStrAct` which converts a structured activity into a LCFG. When call `cvtStrAct` on the outermost activity of a process, it will convert all inner structured activities that includes all basic activities into a LCFG.

The function currently supports 5 structured activities: Sequence, flow, If, While and RepeatUntil. The description of cvtStrAct for each activity and construction of a LCFG from a BPEL process are written in Table 2.2.

BPEL structured activities	Example for mapped LCFG construct and mapping algorithm
<pre><if> ... </if></pre>	 <pre> CREATE graph G CREATE and ADD a If.Start node into G ADD condition of If to the graph CONVERT (if needed) and ADD the inner activity into the graph APPEND If.End node into graph. PARSING all elseif elements APPEND condition of each elseif into G APPEND each activity in elseif element into the graph CREATE an edge from the inner activity to If.End APPEND converted activity in else into G ADD an edge from the else activity with If.End </pre>

<pre><flow> ... </flow></pre>	<pre> graph TD FlowStart[Flow.start] --> S1((Sequence.start)) FlowStart --> S2((Sequence.start)) FlowStart --> S3((Sequence.start)) FlowStart --> S4((Sequence.start)) S1 --> TAssign1[TAssign] S2 --> TAssign2[TAssign] S3 --> TAssign3[TAssign] S4 --> TAssign4[TAssign] TAssign1 --> TInvoke1[TInvoke] TAssign2 --> TInvoke2[TInvoke] TAssign3 --> TInvoke3[TInvoke] TAssign4 --> TInvoke4[TInvoke] TInvoke1 --> SE1((Sequence.End)) TInvoke2 --> SE2((Sequence.End)) TInvoke3 --> SE3((Sequence.End)) TInvoke4 --> SE4((Sequence.End)) SE1 --> FlowEnd[Flow.End] SE2 --> FlowEnd SE3 --> FlowEnd SE4 --> FlowEnd </pre>
	<pre>CREATE a graph G CREATE and ADD a Flow.Start node into G CREATE and ADD a Flow.End node into G FOR EACH inner activity A CONVERT A into a subgraph sG if needed ADD inner activity A into G ADD an edge from the activity A and Flow.End</pre>
<pre><repeatU ntil> ... </repeat Until></pre>	<pre> graph TD Start((RepeatUntil.start)) --> TThrow[TThrow] TThrow --> TBooleanExpr[TBooleanExpr] TBooleanExpr --> End((RepeatUntil.End)) End --> TThrow </pre> <pre>CREATE a graph G CREATE and ADD a RepeatUntil.Start node into G CONVERT inner activity if needed APPEND the inner activity A into G APPEND the condition into G ADD an edge from the condition and A ADD RepeatUntil.End into G ADD an edge from the condition and RepeatUntil.End</pre>




		<pre> CREATE a graph G CREATE and ADD a While.Start node into G CREATE and ADD a condition node into G right after the start node IF inner activity is structured APPEND the converted subgraph from the inner activity into G ELSE APPEND the inner basic node into G CREATE and ADD a While.End node into G right after the condition node </pre>
<pre> <process > . . . </proces s> </pre>	 <pre> graph TD A((Process.start)) --> B((Sequence.start)) B --> C[TReceive] C --> D((Sequence.End)) D --> E((Process.End)) </pre>	<pre> CREATE a graph G CREATE and ADD a Process.Start vertex into G CONVERT the inner activity A if needed APPEND the inner activity into G CREATE and ADD a Process.End vertex into G ADD an edge from inner activity into Process.End vertex </pre>

Table 2.2 Mapping structured activities and <process> element to LCFG constructs

2.3.2. Algorithm of transforming from labeled flow control graphs to Promela programs

The second stage of transforming from BPEL to Promela is to transform LCFG into Promela instructions. There are 2 steps in this stage, the first step is to traverse the LCFG to extract BPEL activities meta-objects and the second step is to map extracted BPEL activities element into Promela instructions.

There are 11 basic activities and 7 structured activities in BPEL processes [6]. So far I have just created mapping rules for part of elements in BPEL document. However, because I'm using model checking, it is not necessary to map all elements of BPEL to Promela language. The table below shows whether elements in BPEL are mapped into Promela constructs.

Categories	Elements in BPEL documents	Resolved mapping rules
Non-activity elements	extensions import partnerLinks messageExchanges variables correlationsSet faultHandlers eventHandlers scope	partnerLinks variables
Basic activities	assign compensate empty exit extensionActivity invoke receive reply rethrow throw validate wait	invoke receive reply assign
Structured activities	sequence if while repeatUntil pick flow forEach	sequence if while repeatUntil flow

Table 2.3 Resolved mapping rules for BPEL elements

I will describe the mapping rules for each of them.

I use some conventions:

- elements are enclosed in “<” and “>”
- attributes are marked by “@” at the beginning.
- [@linkName] means the value of the attribute linkName.
- [<literal>] can be read: the text value of the element <literal>
- We assume that there is **spec** which is an object of model of Promela.

The starting point of generated Promela code is `init` process that will initialize all global variables and create one instance for each process type.

Mapping rules for non-activity elements

i. `<partnerLinks>`

Inner element or attribute (marked by @)	Mapping rule
<code><partnerLink></code>	Yes

`<partnerLink>`

Inner element or attribute (marked by @)	Mapping rule
@name	Yes
@partnerLinkType	No
@myRole	No
@partnerRole	No
@initializePartnerRole	No

A `partnerLink` is mapped into a proctype and 2 rendezvous channels.

```
processPartnerLink(partnerLink, Spec){
  FOR EACH <partnerLink>
    ADD a channel to Spec with the name: [partnerLink's name]"PL_IN"
    ADD a channel to Spec with the name: [partnerLink's name]"PL_OUT"
    ADD a proctype to Spec with the name: [partnerLink's name] /* it is
safe because the pl's name is unique in BPEL document*/
  }
}
```

ii. `<variables>`

Inner element or attribute (marked by @)	minOccurs	maxOccurs	use (for attributes)	Mapping rules
<code><variable></code>	0	unbounded	-	yes

`<variable>`

Inner element or attribute (marked by @)	minOccurs	maxOccurs	use (for attributes)	Mapping rules
<code><from></code>	0	1	-	no
@name	-	-	required	yes
@messageType	-	-	optional	yes
@type	-	-	optional	yes
@element	-	-	optional	yes

```
processVariables(Variables, Spec){
  FOR EACH inner variable element
    IF @messageType was used
      ADD to Spec's typedef list a typedef named [@messageType]
      ADD to Spec a variable declaration: [@messageType] [@name]
    IF @type was used
      ADD a variable declaration: converted_type [@name]
  //the [@type] was converted according to Table 2.4
  /* According to [6], either @messageType or @type is used */
}
```

boolean	bool
string	mtype
short unsignedShort	short
duration dateTime time date	
decimal integer int long negativeInteger nonPositiveInteger unsignedInt unsignedLong positiveInteger nonNegativeInteger float double	
byte unsignedByte	byte
gYearMonth gYear gMonthDay gDay gMonth hexBinary base64Binary anyURI QName NOTATION normalizedString token language NMTOKEN NMTOKENS Name NCName	

ID IDREF IDREFS ENTITY ENTITIES	
---	--

Table 2.4 XML-Promela Data Type Conversion Rules

Because the string values often are used in comparisons so I will add a value in the right side of the comparison and a value ‘other’. This idea was also used in

Other type, I will map to short and consider each integer value as a representation of one value in the value range of the type.

To prevent the explosion of state, I chose the type as small as possible.

Mapping rules for vertices that represent basic activities

For each vertex, I will detach embedded object that represent a basic activity and perform an algorithm on it. In this section, I will describe mapping rules for basic activities.

i. <invoke>

Inner element or attribute (marked by @)	Mapping rule
<correlations>	no
<catchAll>	no
<compensationHandler>	no
<toParts>	no
<fromParts>	no
@partnerLink	yes
@portType	yes
@operation	yes
@inputVariable	yes
@outputVariable	yes

```
processInvoke(invoke element) {
    ADD a mtype whose the name is @operation (the value of attribute
    operation)
    ADD a send statement like this:
    [@partnerLink]PL_OUT![@operation] ([@inputVariable])
    ADD a receive statement like this:
    [@partnerLink]PL_IN![@operation] ([@outputVariable])
}
```

ii. <assign>

Inner element or attribute (marked by @)	Mapping rules
<copy>	yes

<extensionAssignOperation>	no
@validate	no

Mapping inner elements in <assign>:

<copy>

Inner element or attribute (marked by @)	Mapping rules
<from>	yes
<to>	yes
@keepSrcElementName	no
@ignoreMissingFromData	no

<from>

Inner element or attribute (marked by @)	Mapping rules
<documentation>	no
<literal>	yes
<query>	no
@expressionLanguage	no
@variable	yes
@part	yes
@property	no
@partnerLink	no
@endpointReference	no

<to>

Inner element or attribute (marked by @)	Mapping rules
<documentation>	no
<query>	no
@expressionLanguage	no
@variable	yes
@part	yes
@property	no
@partnerLink	no

There are 6 variants for the “from-spec” and “to-spec” which are shown in

Variant	from-spec	Mapping rules	to-spec	Mapping rules
Variable	yes	no	yes	yes
PartnerLink	yes	no	yes	no
Property	yes	no	yes	no
Expression	yes	no	yes	no

Literal	yes	yes	no	no
Empty	yes	no	yes	no

```

processAssign(assign element, Spec, Proctype) {
  IF <from> uses literal variant AND there is no mtype with the same name
    ADD a the value of <literal> as a mtype to Spec
  IF <to> uses variable variant and the type in Promela is typedef
    ADD a field in the typedef declaration: converted_type [@part] to
  Spec
    ADD an assignment statement: [@variable].[@part]=[<literal>]
}

```

iii. <receive>

Inner element or attribute (marked by @)	Mapping rules
<correlations>	No
<fromParts>	No
<sources>	Yes
@partnerLink	Yes
@portType	No
@operation	Yes
@variable	Yes
@createInstance	No
@messageExchange	No

```

processReceive(<receive>) {
  CALL processTargets
  CREATE a receive statement:
  [@partnerLink]+"PL_IN?"+"[@operation]+"([@variable])"
  ADD to the proctype named [@partnerLink]: a local variable named
  the same with the proctype followed by a cardinar (for example:
  assessor1, assessor2, ...)typed the same with the variable [@variable] and
  a send statement: [@partnerLink]+"PL_IN!"+"[@operation]+"([value of the
  previous declared variable])"
  CALL processSources(the <sources> of this element)
  IF suppressJoinFailure==true and there is no joinCondition then
  create an If construct with condition is the AND-logic of
  all[@linkName]==Lfalse in
}

```

Variable declaration	Mapping rules
Element	No
Complex type	No
Message type (with simple type parts)	Yes
Message type (with complex type parts)	No
Simple type	No

iv. <reply>

Inner element or attribute (marked by	minOccurs	maxOccurs	use (for attributes)	Mapping rules

@)				
correlations	0	1		no
toParts	0	1		no
partnerLink			required	yes
portType			optional	no
operation			required	yes
variable			optional	yes
faultName			optional	no
messageExchange			optional	no

```

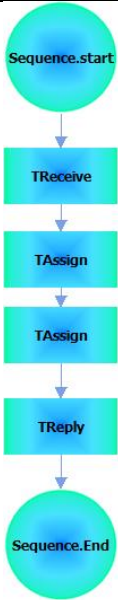
processReply(reply element){
    CREATE a send statement:
    [@partnerLink]PT_OUT![@operation] ([@variable])
    CALL processSources()
    CALL processTargets()
}

```

Mapping rules for LCFG constructs that represent structured activities

The mapping rules from LCFG constructs for structured activities to Promela codes perform on types of construct. For each type of construct, I will traverse all inner vertices and perform algorithms for basic LCFG vertices as described in the above section.

(The input argument is a Sequence *seq* that represents a sequence of Promela statements)

<p>Sequence</p>		<pre> <sequence> processSequence (Sequence) { FOR EACH inner activity PROCESS the activity and ADD it into the seq } e.g. proctype Shipping() { ... [sequential statements] } </pre>
------------------------	---	---

Flow		<pre> <flow> processFlow(Sequence) { FOR EACH activity Create a new proctype Process the activity and add into the newly created proctype ADD a run statement in to the seq } </pre>
If		<pre> <if> processIf(Sequence) { FOR EACH inner condition CREATE a if-statement PROCESS the activity and ADD the newly created if-statement into the seq } </pre>
While Repeat-Until		<pre> <while> and <repeatUntil> processWhileRepeat(Sequence) { CREATE a do-statement CREATE a condition ADD the condition into the do- statement PROCESS inner activity and ADD it following the condition ADD the newly created do- statement into the seq } </pre>

Table 2.5 Mapping rules for LCFG constructs that represent structured activities

Resolving links - synchronization dependencies

In order to model links between activities in `<flow>` activity, I create variables which has the same name with links and can hold one of 3 values: *Ltrue* (for *true*), *Lfalse* (for *false*) and *0* (for *unset*). Because any activity can be the source(s) or target(s) of links, I process activity's `<sources>` and `<targets>` before processing its contents.

v. `<sources>`

Inner element or attribute (marked by @)	minOccurs	maxOccurs	use (for attributes)	Mapping rules
<code><source></code>	1	unbounded	-	yes

`<source>`

Inner element or attribute (marked by @)	minOccurs	maxOccurs	use (for attributes)	Mapping rules
<code><transitionCondition></code>	0	1	-	yes
<code>@linkName</code>	-	-	required	yes

vi. `<targets>`

Inner element or attribute (marked by @)	minOccurs	maxOccurs	use (for attributes)	Mapping rules
<code><joinCondition></code>	0	1	-	partly
<code><target></code>	1	unbounded	-	yes

`<target>`

Inner element or attribute (marked by @)	minOccurs	maxOccurs	use (for attributes)	Mapping rules
<code>@linkName</code>	-	-	required	yes

```

processSourcesTargets(<sources>, <targets>){
  IF the activity contains both <sources> and <targets>
    CREATE an atomic construct
    FOR EACH <source> element
      CREATE expression to access field in variable: VarName.VarPart
      IF the right side expression is numeric THEN
        ADD a part into the variable typedef deliration: the type has
the least bit and the name is the part of the expression
      ELSE IF the right side expression is a string THEN
        IF the string does not exist in mtype yet THEN
          ADD a mtype declaration to spec
        END IF
        ADD a mtype part into the variable typedef deliration
      END IF
    CREATE if-statement ifs
    ADD to ifs an option: "::"[@transitionCondition],[@linkName]=Ltrue
  
```



```

        ADD to ifs an option: "::"![@transitionCondition],[@linkName]=Ltrue
ELSE IF the activity contains only <sources>
CREATE an atomic construct
    FOR EACH <source> element
        CREATE expression to access field in variable: VarName.VarPart
        IF the right side expression is numeric THEN
            ADD a part into the variable typedef deliration: the type has
the least bit and the name is the part of the expression
        ELSE IF the right side expression is a string THEN
            IF the string does not exist in mtype yet THEN
                ADD an mtype declaration to spec
            END IF
            ADD an mtype part into the variable typedef deliration
        END IF
ELSE IF the activity contains only <targets>
CREATE if-statement ifs
ADD to ifs an option: "::"[@transitionCondition],[@linkName]=Ltrue
ADD to ifs an option: "::"![@transitionCondition],[@linkName]=Ltrue
}

```

Randomize values of variables

To simulate the behavior of true application, I add routines that randomize the value of variables and fields in variables into process types corresponding to partner links.

To randomize a numeric field f in a variable v , I use the following codes:

```

v.f=0;
do
:: v.f++
:: v.f-
od;

```

To randomize an mtype field f in a variable v , f can hold the value *first*, *second* and *other*, I use the following codes:

```

v.f=0;
do
:: v.f=first
:: v.f=second
:: v.f=other
od;

```

Because of non-determinism in Spin, $v.f$ will have a random value after the loop.

CHAPTER SUMMARY

In this chapter, I described the problem of verifying BPEL processes and current research trends. After that, I proposed my solution architecture and algorithms in the solution.

Chapter 3. IMPLEMENTATION

In this chapter:

- Realization of the solution described in Chapter 2 in BPEL Verification Tool.
- Detail description of architecture, components and features of the tool.
- Basis of the implementation of the tool.

As I described in the previous chapter, there are 2 main algorithms to transform a BPEL process into a Promela program. However, those algorithms are not enough for a complete solution because there are many other technical problems to be solved to realize it. Some typical problems are how to analyze a BPEL document efficiently, how to represent a graph in a memory and the most significant problem may be how to guarantee generated Promela programs to be correct.

For ease of reading, from this point, I will refer the tool as BVT which stands for BPEL Verification Tool.

3.1. Tool Architecture

BVT tool contains 2 main parts:

- Transformer core: performs the transformation from a BPEL process to a Promela program via a LCFG representation
- BVT's graphical user interface: helps end-users to interact with the program to verify a BPEL process.

The tool architecture is illustrated in Figure 3-1.

The main process of verifying a BPEL specification is as following: A user opens a BPEL document which is a text file. The file will be validated against relevant XML Schema and then transformed to a set of objects in BPEL metamodel. Then, the objects will be transformed into a LCFG object which represents a graph and is displayed on screen. The LCFG object is then transformed into a set of objects in Promela metamodel. In the end of the process, a Promela program will be generated.

The Promela program and desired properties expressed in LTL formulae will be then fed to the Spin model checker. Users need not to know much about Spin to check typical properties of it because the tool includes a GUI for it.

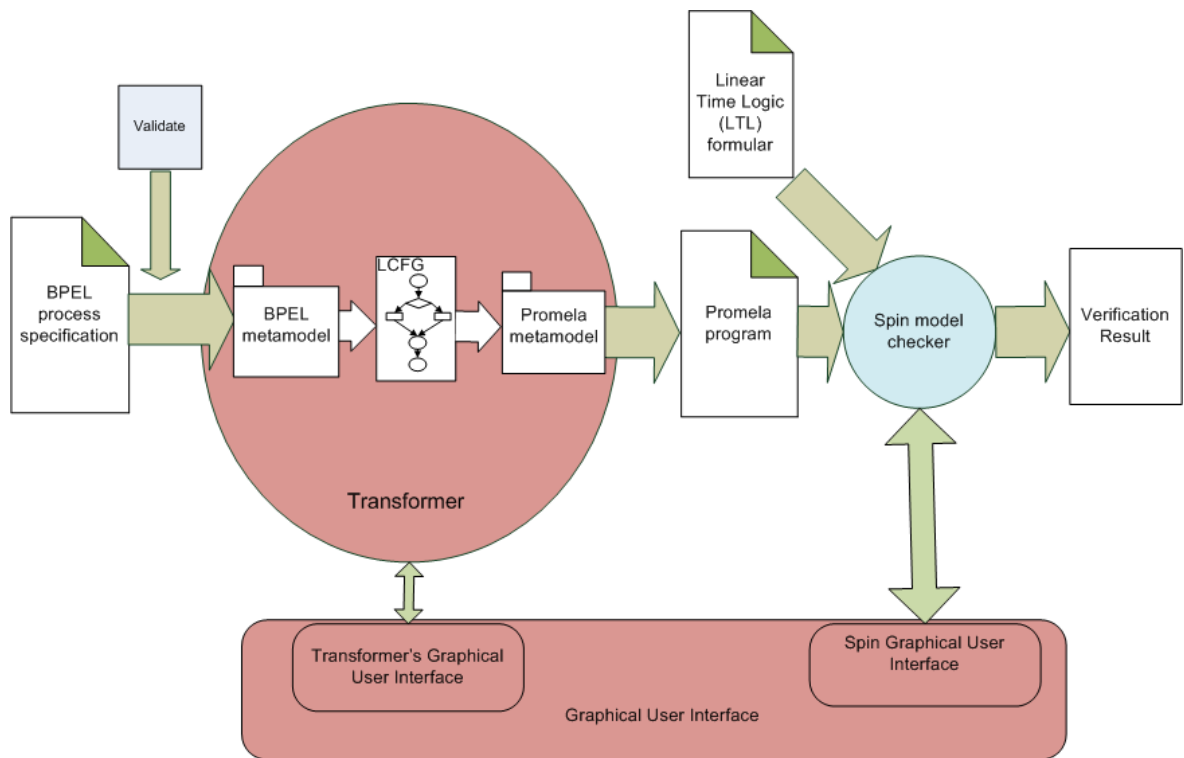


Figure 3-1 BPEL Verification Tool architecture

BVT relies heavily on metamodels, which can be defined as follows:

Definition 3.15 Metamodel

A metamodel of a language is a set of classes, each of which represents some element of the target language

Metamodels help us to manipulate languages in term of objects which fit very well on object-oriented language like Java. From an instance of the target language, I can construct a set of objects based on the metamodel, manipulate them, and then generate another document instance.

The remain of this section, I will provide high level description of metamodels used in BVT. For reference to classes in metamodel, see appendixes.

3.2. Metamodels in BVT

3.2.1. BPEL metamodel

In order to process BPEL documents, which, in principle, are XML documents, I have a wide range of choices such as processing them as text files, using Document Object Model like Java API for XML Processing (JAXP) [27], Xerces2 XML Parser [28] and JDOM [29].

From the XML Schema of abstract process [30] and executable process [31], I used the Java Architecture for XML Binding (JAXB) library (read more on appendix) to

generate the metamodels. The model for abstract and executable processes contains **82 and 79 classes** respectively.

For the ease of serialization of objects, I set the JAXB compiler so that all generated classes implement the Serializable interface.







From a BPEL document, BVT uses unmarshalling functionality of JAXB library to create a set of tightly related objects. The root of them is Tprocess that represents a BPEL process. By using that object, I can access to every element in the BPEL document.

Because there are 2 types of BPEL processes: executable and abstract process, the BPEL metamodel is in 2 packages: `model.bpel.abs` for abstract processes and `model.bpel.exe` for executable processes.

3.2.2. LCFG model

LCFG Model is created to represent a LCFG which is a directed graph. So I create a class LCFG that inherits class `ListenableDirectedGraph` in the JGraphT library (Appendix B.2. JGraphT).

Each node in this class can be bound with an arbitrary object. Besides, I created classes that represent the start and end of graph constructs for structured activities such as `StartSequence` for the start of a *sequence* activity. Types of Start and End nodes are listed in Table 3.1. Besides, LCFG contains many other methods designed specifically for manipulating with BPEL elements: such as APPEND, ADD a new node into graph, and ADD an activity into a graph.

 <p>Process.start</p> <p>Start of a process</p>	 <p>Process.End</p> <p>End of a process</p>
 <p>Sequence.start</p> <p>Start of a sequence</p>	 <p>Sequence.End</p> <p>End of a sequence</p>
 <p>RepeatUntil.start</p> <p>Start of a repeat until</p>	 <p>RepeatUntil.End</p> <p>End of a repeat until</p>


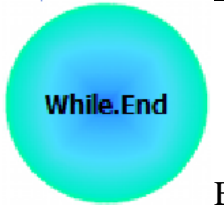

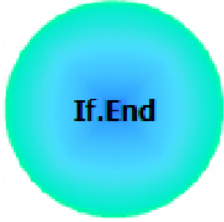
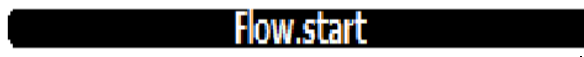



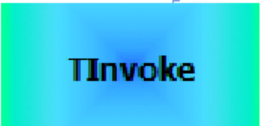



 Start of a <code>while</code>	 End of a <code>while</code>
 Start of an <code>if</code>	 End of an <code>if</code>
 Start of a <code>flow</code>	 End of a <code>flow</code>
 Node for a <code>reply</code> activity	 Node for a <code>receive</code> activity
 Node for an <code>invoke</code> activity	 Node for an <code>assign</code> activity
 Node for a condition	 Transition edge

Table 3.1 Types of nodes in LCFG

Figure 3-2 shows the UML diagram of `model.graph` package which represents LCFG and its elements.

3.2.1. Promela metamodel

In order to generate grammatical correct Promela source codes, I need a systematic method instead of pure text manipulation. Because I use Java, an object-oriented language, a metamodel for Promela language will be most appropriate. After some search on the internet without any result, I decided to create a model by myself. I created an object model based on grammar of Promela language [13]. [32] shows the relationship between a context-free grammar and a metamodel and algorithms to convert between them.

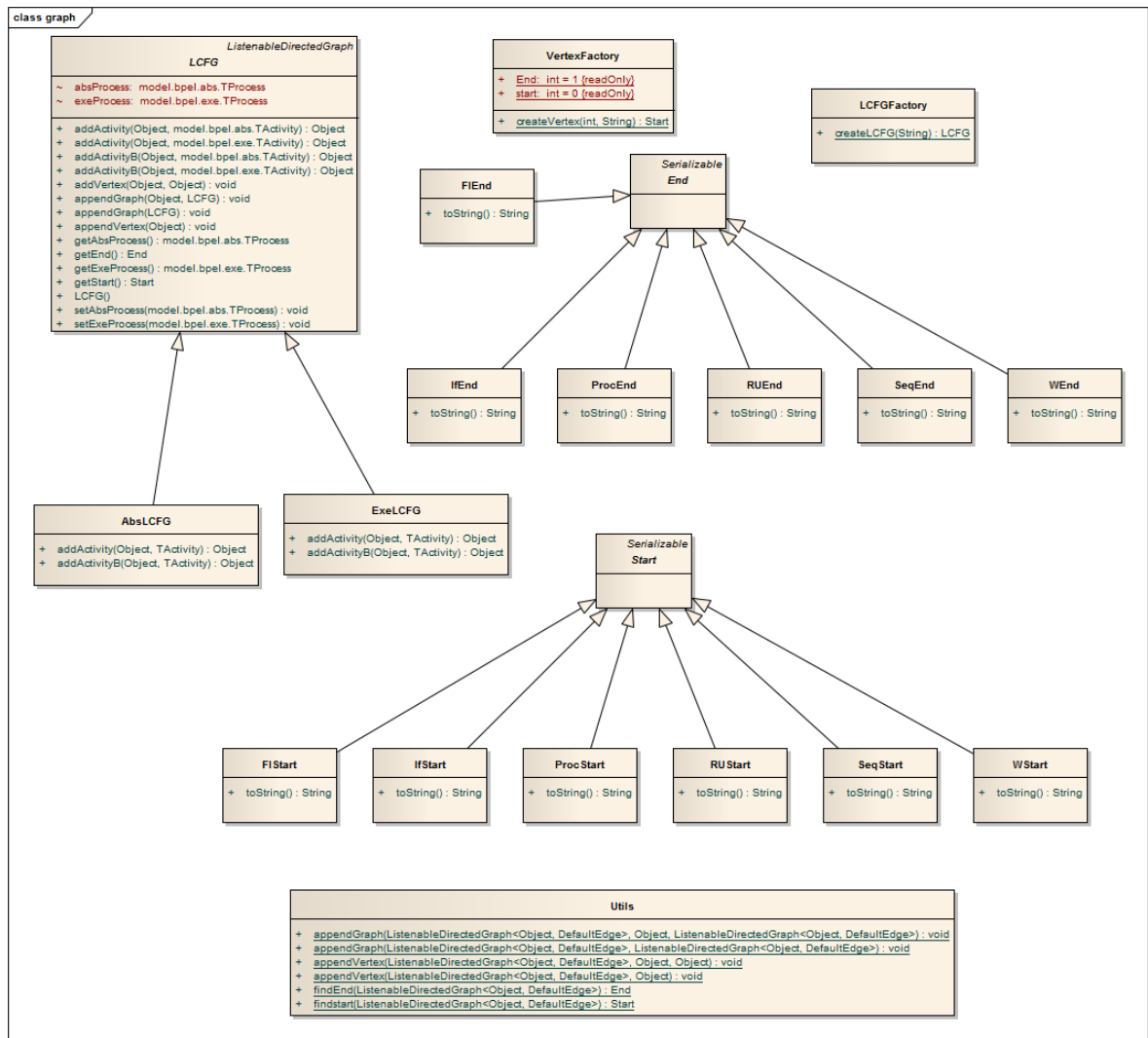


Figure 3-2 Classes in model.graph package

One advantage of generating Promela programs using metamodel is the guarantee that generated program is compatible with grammar rules. On the other hand, metamodel's complexity is a trade-off. The use of metamodel is highly complicated. The reason is that the depth of syntax tree for even simple program is quite high. As an example, let's consider a "hello world" program:

```
init { printf("hello world") }
```

Its syntax tree whose decompositions of nonterminals "name" and "string" are omitted has the depth of 8. That means we have to create and manage a tree of at least 8 self enclosed objects.

```
spec
|-> module -> init
    |-> INIT
    |-> sequence -> step -> stmtnt
        |-> PRINT -> "printf"
        |-> string -> "hello world"
```

My metamodel for Promela consists of **160 classes**. Each class often represents a nonterminal symbol in Promela grammar. The key points in the method of creating the Promela metamodel:

❖ If the right side of the production consists of:

- choices of terminals and nonterminals or nonterminals only

E.g.

```
typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN |  
uname
```

or

```
module: proctype| init| never| trace| utype| mtype| decl_lst
```

→ Create an intermediate class which is an abstract class that implements `ToCode` interface. Then create classes that extend the abstract class for each terminal or nonterminal at the right side. Classes for terminals are quite simple with only method `toCode()`.

- mixture of terminals and nonterminal:

E.g.

```
send      : varref '!' send_args  | varref '!' '!' send_args
```

or

```
ch_init  : '[' const ']' OF '{' typename [ ',' typename ] *  
'}'
```

→ Consider each option as a nonterminal and repeat the step (1).

- only terminals

E.g.

```
chanpoll: FULL | EMPTY | NFULL | NEMPTY
```

→ Create an `Enum` for left side.

❖ If one nonterminal appears alone on the right side more than once

E.g.

```
module   : ...| decl_lst
```

and

```
step     : ...| decl_lst
```

→ Create wrapper class for this nonterminal character. A wrapper class should be name with prefix `W` and contains only one field that is the wrapped class. In the above case, I will create `Decl_lst` and `WDecl_lst`: `Wdecl_lst` contains one field `decl_lst` of the type `Decl_lst`. This rule is used as Java does not allow multiple inheritances.

❖ I used a list to represent the form `[xyz]*`.

E.g.

```
decl_lst: one_decl [ ';' one_decl ] *
```

→ `private List<One_decl> decls = new ArrayList<One_decl>();`

❖ For simple character and string

E.g.

```
number : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

```
string : ''' [ any_ascii_char ] * '''
```

→ Create Alpha, Number and Str classes with methods that check the value constrains.

❖ For the rule of this form, which is very popular in the grammar:

```
arg_lst : any_expr [ ',' any_expr ] *
```

I create a class with a list of inner classes which represent the nonterminal in the right side.

❖ Some other notes about the metamodel:

○ The code also contains helper classes and codes in order to make the use of the code more effective.

○ ToCode interface contains only 1 method toCode() that returns the code corresponding to the object:

```
public interface ToCode {  
    public String toCode() throws Exception;  
}
```

○ Class Spec is the root of Promela grammar.

○ All class must implement interface ToCode. When calling toCode() method on an instance of a Spec, it will be call recursively and return a string of source code.

❖ Naming method: Class names are based on the meaning or function of the code structure it represents.

❖ The method toCode() contains checking functionality that checks the validation of the input data such as null value for field that must not be null and the range of values. The checking routines will throw an exception of the type RuleViolationException. For simplicity, the grammar constraint checking is performed on this point, i.e. code generation point, only.

3.3. Transformer core module – implementation of algorithms

Transformer core module consists of 2 packages:

- transformer.bl for BPEL to LCFG transformation: Method transform() in class BLTransformer transforms a BPEL process (as model.bpel.abs.TProcess or model.bpel.exe.TProcess) into a LCFG interface.

- `transformer.lp` for LCFG to Promela transformation: Method `transform()` in class `LPTransformer` transforms a LCFG into a `Spec` which is the root of Promela grammar.

Because there are two types of processes: executable and abstract, I used Factory Method design pattern in these package to treat them as one kind. UML diagrams of these packages are shown in Figure 3-3 and Figure 3-4.

Algorithms in previous chapter are implemented on metamodels. For algorithm of transforming BPEL processes to LCFGs, I traverse the a BPEL document and create objects from it using BPEL metamodel and then construct a labeled flow control graph and attach them to the graph. For algorithm of transforming LCFGs to Promela programs, I traverse LCFG instance, detach BPEL element objects, construct an instance of `Spec` which is the root of Promela grammar, and then generate Promela code by calling `toCode()` method.

XPath expression is processed by using a lexical analyzer. This analyzer generated from JJTree file for XPath 2.0 [33] using JavaCC parser generator. From tokens resulted from the analyzer, I create a parser for further processing. So far, this processing is limited to comparison expressions and variables with one level of fields.

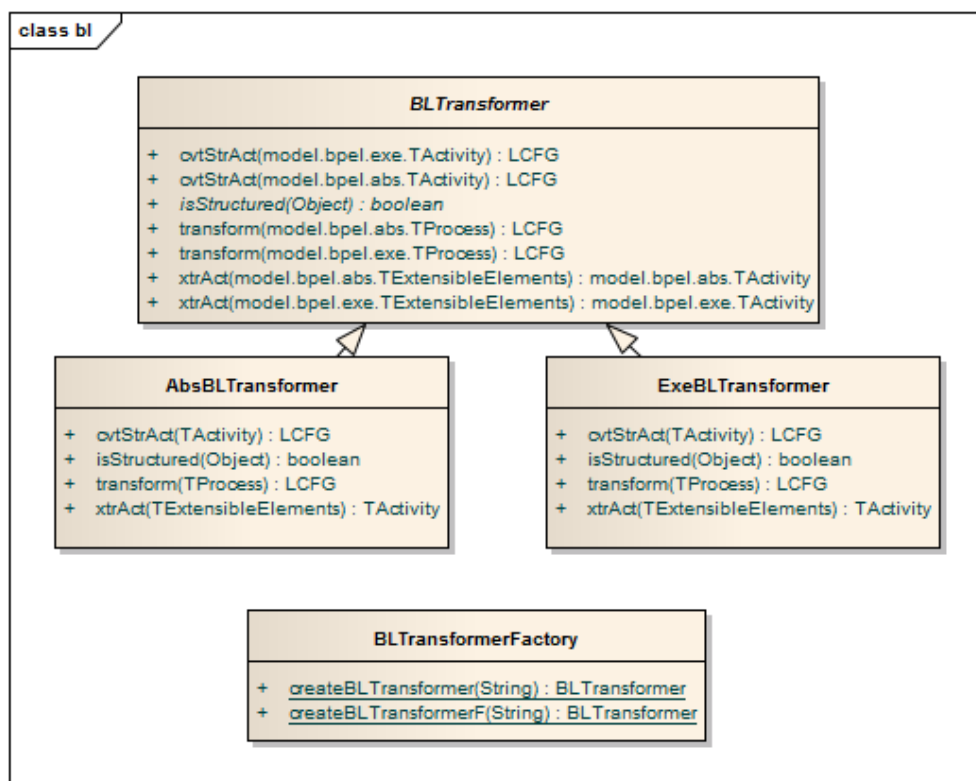


Figure 3-3 Classes in package `transformer.bl`

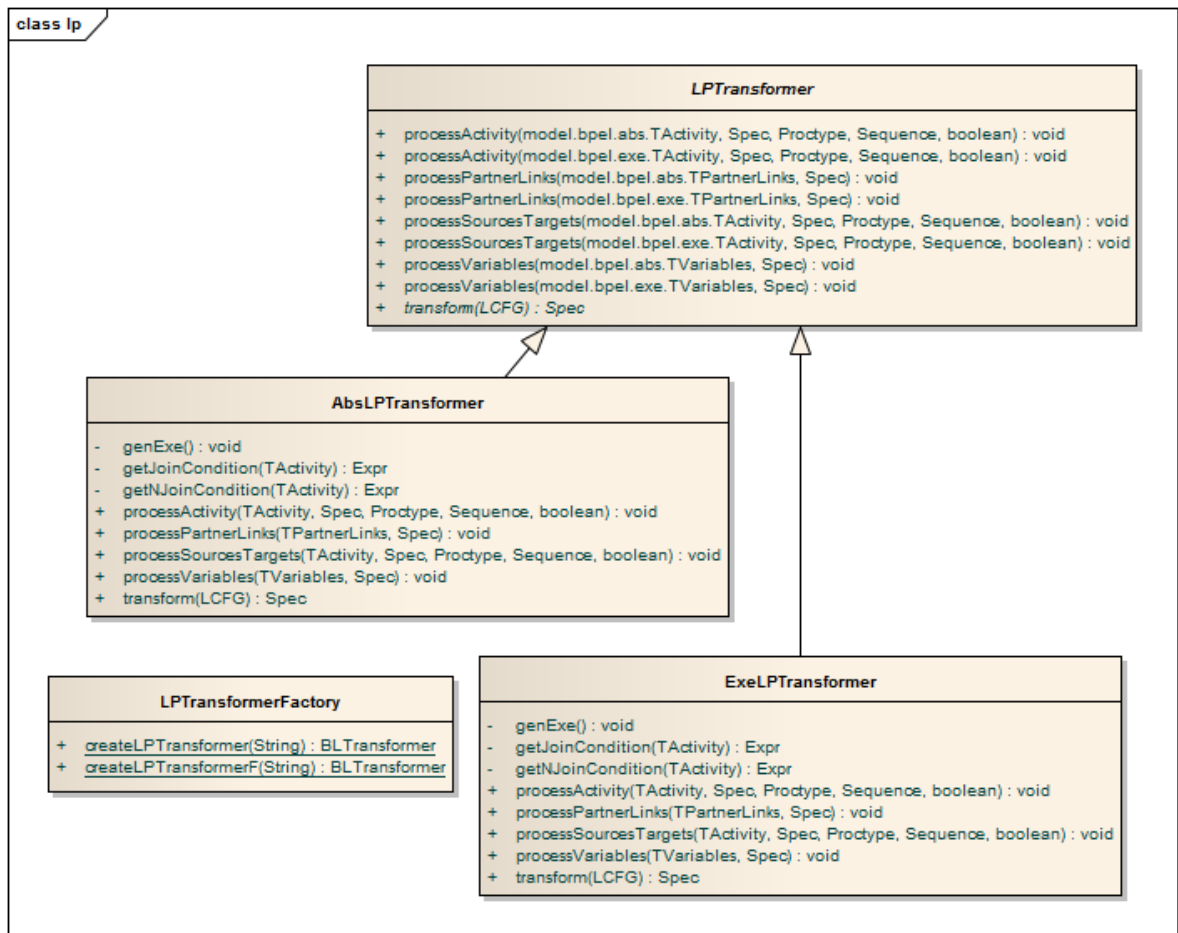


Figure 3-4 Classes in package transformer.lp

3.4. Other features and techniques

- **BPEL document Validation:** Before transforming any BPEL documents, they must be valid. BVT validates them against BPEL 2.0 XML Schema. If there is no error, the process will continue, otherwise, an error will appear. In order to validate these documents, I use `Validator` in JAXP open source library [27].
- **Exporting graph in different file formats:** Tool support exporting LCFGs into DOT[34], GML[35], GraphML[36] that are file formats for graph interchange (using JGraphT), SVG and PNG formats for images processing (using JGraph5).
- **Automatic graph layout and displaying interactive graph on screen:** Vertices and edges in LCFG are laid out using `JGraphTreeLayout` in JGraph5 for pretty display; users can move vertices and edges of graph interactively. This feature is implemented by using JGraph5.
- **Graphical User Interface for Spin:** Users can edit, specify and use all function of Spin via a GUI. This GUI is adopted from jSpin.

- **Syntax highlighting:** Promela codes and LTL formulae are displayed with colors and different fonts, so it's easier for users to edit them. This feature is implemented basing on jsyntaxpane [37] open source library.

For more information about JAXB, JGraphT, JGraph5 and jSpin, see Appendix B.

CHAPTER SUMMARY

In this chapter, I have described the architecture and main parts of BVT, a tool that implements the transformation algorithms. I also provide some details of metamodel for processing BPEL documents, LCFG instances and Promela programs. Finally, I discuss other features related to user interface and exporting and how they are implemented.

Chapter 4. TOOL TESTING AND METHOD EVALUATION

In this chapter:

- Result of a test case for BVT
- Screenshots of BVT in action
- Evaluation of proposed solution
- Evaluation of BVT

4.1. A Transformation Test

To test the transformation algorithms, I use the 4 standards example process in BPEL 2.0 specification [6] as other authors did. Because of the lack of space, I just show the result on the loanApproval process which is specified in [6]. The result LCFG is shown in Figure 4-1 and Promela program is shown in Table 4.1.

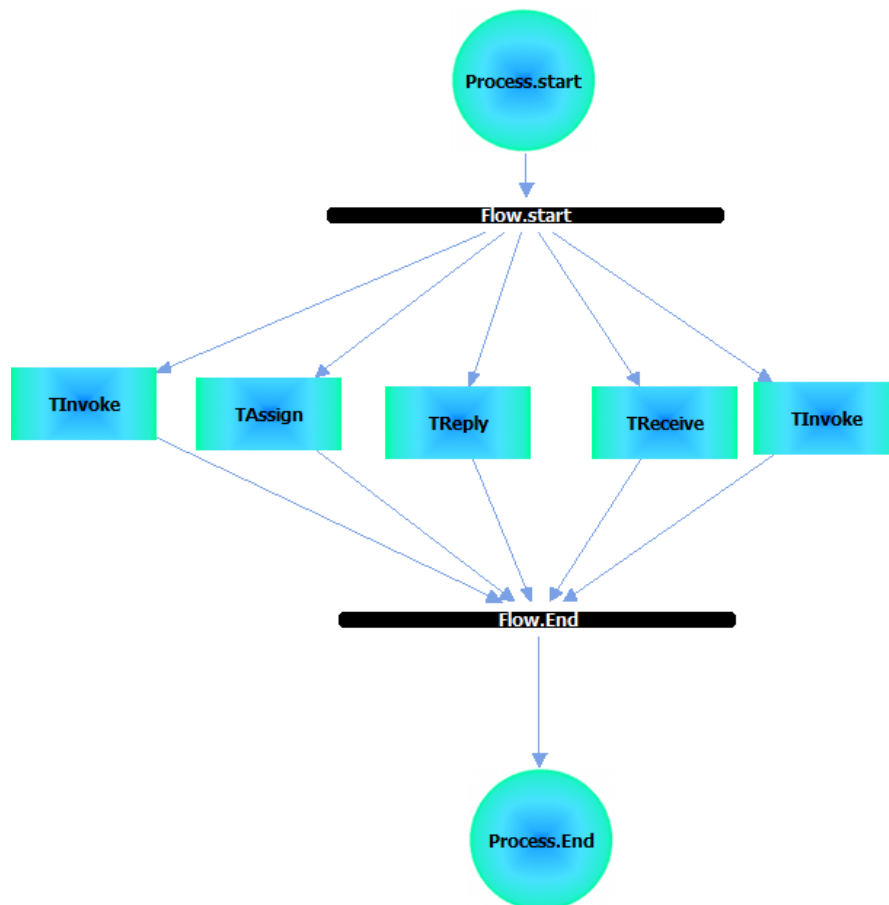


Figure 4-1 LCFG for loanApproval process

```
mtype = { Lfalse }  
mtype = { Ltrue }  
mtype = { request }  
mtype = { check }
```

```

mtype = { low }
mtype = { yes }
mtype = { approve }
mtype = { other }
typedef creditInformationMessage {
byte amount
}
typedef riskAssessmentMessage {
mtype level
}
typedef approvalMessage {
mtype accept
}
chan customerPL_IN = [0] of {mtype, creditInformationMessage}
chan customerPL_OUT = [0] of {mtype, approvalMessage}
chan approverPL_IN = [0] of {approvalMessage, mtype}
chan approverPL_OUT = [0] of {mtype, creditInformationMessage}
chan assessorPL_IN = [0] of {riskAssessmentMessage, mtype}
chan assessorPL_OUT = [0] of {mtype, creditInformationMessage}
creditInformationMessage request_VAR
riskAssessmentMessage risk
approvalMessage approval
mtype receive_to_assess
mtype receive_to_approval
mtype approval_to_reply
mtype assess_to_setMessage
mtype setMessage_to_reply
mtype assess_to_approval
proctype loanApprovalProcess () {
run TReceive1();
run TInvoke2();
run TAssign3();
run TInvoke4();
run TReply5()
}
proctype customer () {
creditInformationMessage customer1;
customer1.amount=0;
do
:: customer1.amount++
:: customer1.amount--
:: break
od;
customerPL_IN!request,customer1;
mtype operation2;
approvalMessage customer3;
end:customerPL_OUT?operation2,customer3
}
proctype approver () {
mtype operation1;
creditInformationMessage approver2;
end:approverPL_OUT?operation1,approver2;
approvalMessage approver3;
do
:: approver3.accept=yes
:: approver3.accept=other
:: break
od;
approverPL_IN!approve,approver3
}
proctype assessor () {
mtype operation1;

```

```

creditInformationMessage assessor2;
end:assessorPL_OUT?operation1,assessor2;
riskAssessmentMessage assessor3;
do
:: assessor3.level=low
:: assessor3.level=other
:: break
od;
assessorPL_IN!check,assessor3
}
proctype TReceive1 () {
mtype operation1;
customerPL_IN?operation1,request_VAR;
atomic{
if
:: request_VAR.amount < 10000;
receive_to_assess=Ltrue
:: !(request_VAR.amount < 10000);
receive_to_assess=Lfalse
fi;
if
:: request_VAR.amount >= 10000;
receive_to_approval=Ltrue
:: !(request_VAR.amount >= 10000);
receive_to_approval=Lfalse
fi
};
}
proctype TInvoke2 () {
mtype operation1;
if
:: receive_to_assess == Ltrue;
assessorPL_OUT!check,request_VAR;
assessorPL_IN?operation1,risk;
atomic{
if
:: risk.level == low;
assess_to_setMessage=Ltrue
:: !(risk.level == low);
assess_to_setMessage=Lfalse
fi;
if
:: risk.level != low;
assess_to_approval=Ltrue
:: !(risk.level != low);
assess_to_approval=Lfalse
fi
};
:: receive_to_assess == Lfalse;
atomic{
assess_to_setMessage=Lfalse;
assess_to_approval=Lfalse
};
fi
}
proctype TAssign3 () {
if
:: assess_to_setMessage == Ltrue;
approval.accept=yes;
atomic{
setMessage_to_reply=Ltrue
};
}

```

```

:: assess_to_setMessage == Lfalse;
atomic{
setMessage_to_reply=Lfalse
};
fi
}
proctype TInvoke4 () {
mtype operation1;
if
:: receive_to_approval == Ltrue || assess_to_approval == Ltrue;
approverPL_OUT!approve,request_VAR;
approverPL_IN?operation1,approval;
atomic{
approval_to_reply=Ltrue
};
:: receive_to_approval == Lfalse && assess_to_approval == Lfalse;
atomic{
approval_to_reply=Lfalse
};
fi
}
proctype TReply5 () {
if
:: setMessage_to_reply == Ltrue || approval_to_reply == Ltrue;
customerPL_OUT!request,approval
fi
}
init {run loanApprovalProcess();
run customer();
run approver();
run assessor()
}

```

Table 4.1 Generated Promela program for loanApproval process

The generated program contains 10 concurrent processes: 3 processes emulate 3 partner links: customer, approver and assessor; 1 process emulates the whole loanApproval process that runs 5 others processes that emulates 5 concurrent activities in <flow> activity; and a process init() that serves as the entry point for all other processes.

I will verify the safety, liveness and another property of the loanApproval model. Due to the explosion of state space, the number of states of this model exceeds my personal computer memory, so I have to change the data type of amount field in creditInformationMessage from short into byte. To make This change not to affect the verification result I also change the right hand side number in comparisons of request_VAR.amount into 100.

The verification result is shown in Table 4.2: besides the statistics on number of states, running time, depth in reachability graph and memory usage, no deadlock and no unreachable state was found in this process.

(Spin Version 5.2.4 -- 2 December 2009)
+ Partial Order Reduction

```

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks         - (disabled by -DSAFETY)
  invalid end states   +

State-vector 100 byte, depth reached 286, errors: 0
  42288 states, stored
  23561 states, matched
  65849 transitions (= stored+matched)
  556 atomic steps
hash conflicts:      587 (resolved)

  6.504   memory usage (Mbyte)

unreached in proctype loanApprovalProcess
  (0 of 6 states)
unreached in proctype customer
  (0 of 10 states)
unreached in proctype approver
  (0 of 9 states)
unreached in proctype assessor
  (0 of 9 states)
unreached in proctype TReceive1
  (0 of 15 states)
unreached in proctype TInvoke2
  (0 of 23 states)
unreached in proctype TAssign3
  (0 of 10 states)
unreached in proctype TInvoke4
  (0 of 11 states)
unreached in proctype TReply5
  (0 of 5 states)
unreached in proctype :init:
  (0 of 5 states)

pan: elapsed time 0.098 seconds
pan: rate 431510.2 states/second

```

Table 4.2 Default verification of loanApproval Promela program with Spin

Now I verify that if the process satisfies the property: **For requests of the same amount, there must be no case in which one of the requests is approved but the other is processed differently.**

To check that property, I add 2 definitions to the above program:

```

#define accepted (approval.accept==yes)
#define rejected (approval.accept!=yes)

```

and using spin against the LTL formula for that property which is “!(⟨>(accepted && rejected))”. (The operator “⟨>” means eventually and “!” means “not”.)

The translated never claim for the *negated* LTL formula is

```

never { /* (⟨>(accepted && rejected)) */
T0_init:
  if
  :: ((accepted) && (rejected)) -> goto accept_all
  :: (1) -> goto T0_init

```



```

    fi;
accept_all:
    skip
}

```

The result is shown in Table 4.3: There is no never claim violation found, i.e. the process satisfies the property.

```

warning: for p.o. reduction to be valid the never claim must be stutter-
invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 5.2.4 -- 2 December 2009)
+ Partial Order Reduction

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 104 byte, depth reached 1590, errors: 0
  42800 states, stored
  39486 states, matched
  82286 transitions (= stored+matched)
  556 atomic steps
hash conflicts:      1064 (resolved)

Stats on memory usage (in Megabytes):
  4.898  equivalent memory usage for states (stored*(State-vector +
overhead))
  3.017  actual memory usage for states (compression: 61.59%)
         state-vector as stored = 58 byte + 16 byte overhead
  2.000  memory used for hash table (-w19)
  0.305  memory used for DFS stack (-m10000)
  5.235  total actual memory usage

unreached in proctype loanApprovalProcess
  (0 of 6 states)
unreached in proctype customer
  (0 of 10 states)
unreached in proctype approver
  (0 of 9 states)
unreached in proctype assessor
  (0 of 9 states)
unreached in proctype TReceive1
  (0 of 15 states)
unreached in proctype TInvoke2
  (0 of 23 states)
unreached in proctype TAssign3
  (0 of 10 states)
unreached in proctype TInvoke4
  (0 of 11 states)
unreached in proctype TReply5
  (0 of 5 states)
unreached in proctype :init:
  (0 of 5 states)

pan: elapsed time 0.219 seconds
pan: rate 195433.79 states/second

```

Table 4.3 Verification result for a property of loanApproval process

4.2. Some Screenshots of BVT

These are some screenshots of the tool when it executes.

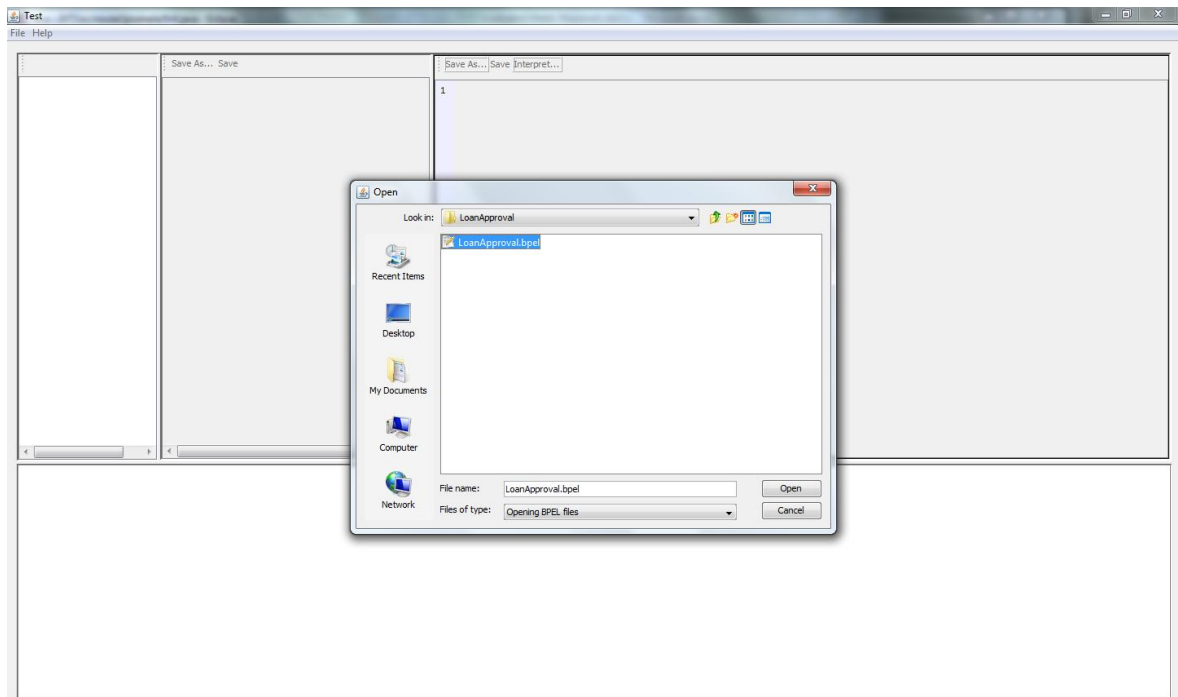


Figure 4-2 BVT screenshot – opening a BPEL document

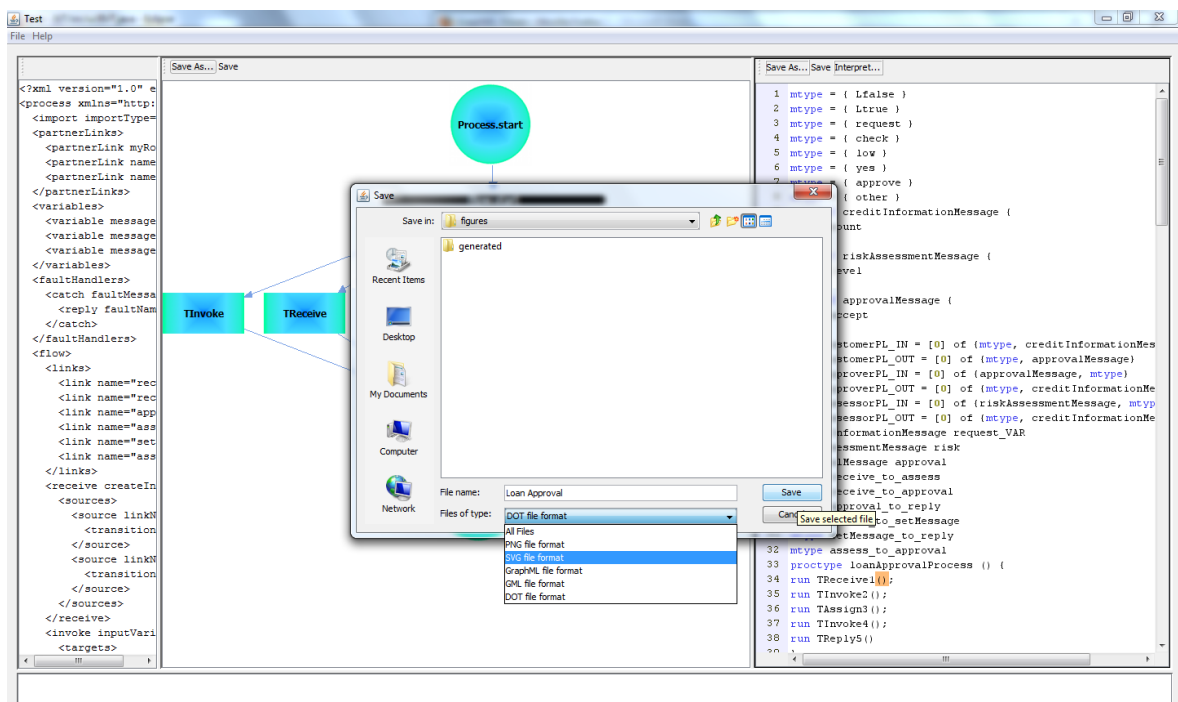


Figure 4-3 BVT screenshot – exporting a graph in many file formats

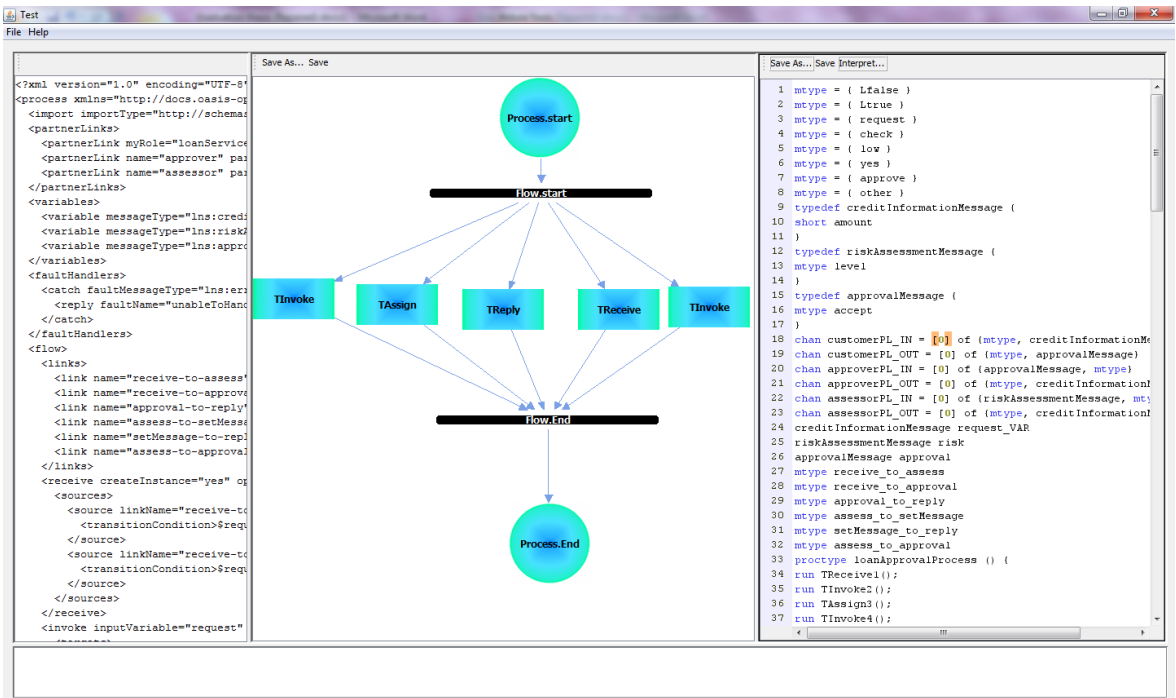


Figure 4-4 BVT screenshot - transformation from a BPEL process into a Promela program

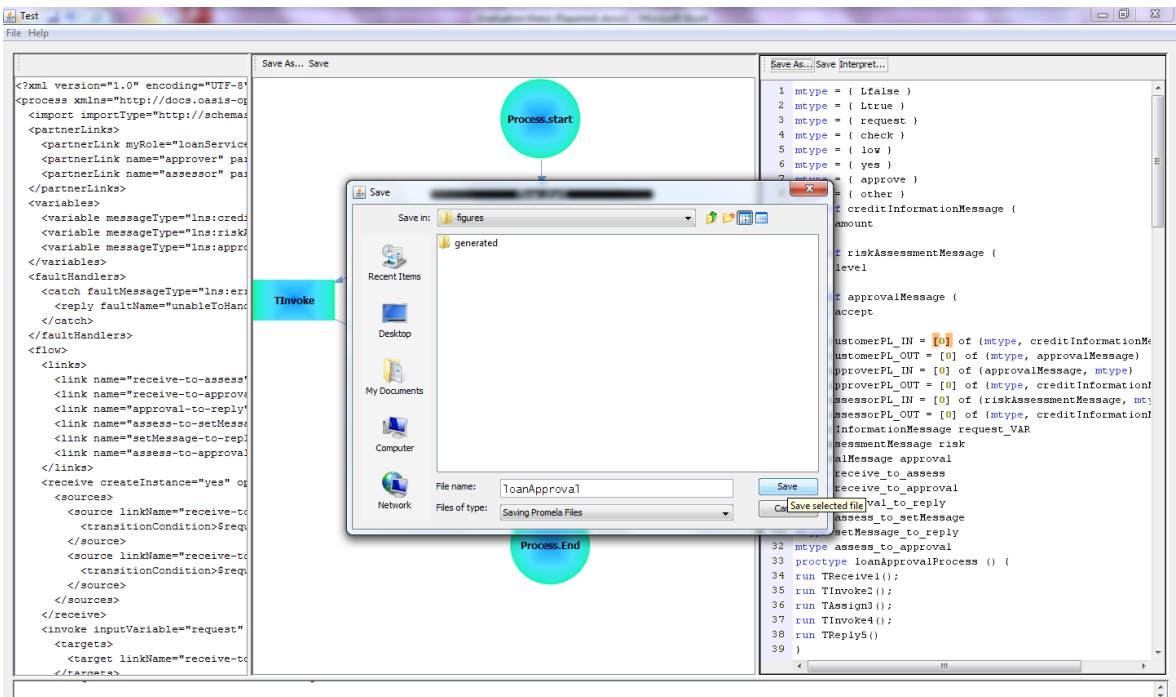


Figure 4-5 BVT screenshot - saving the generated Promela program to a text file

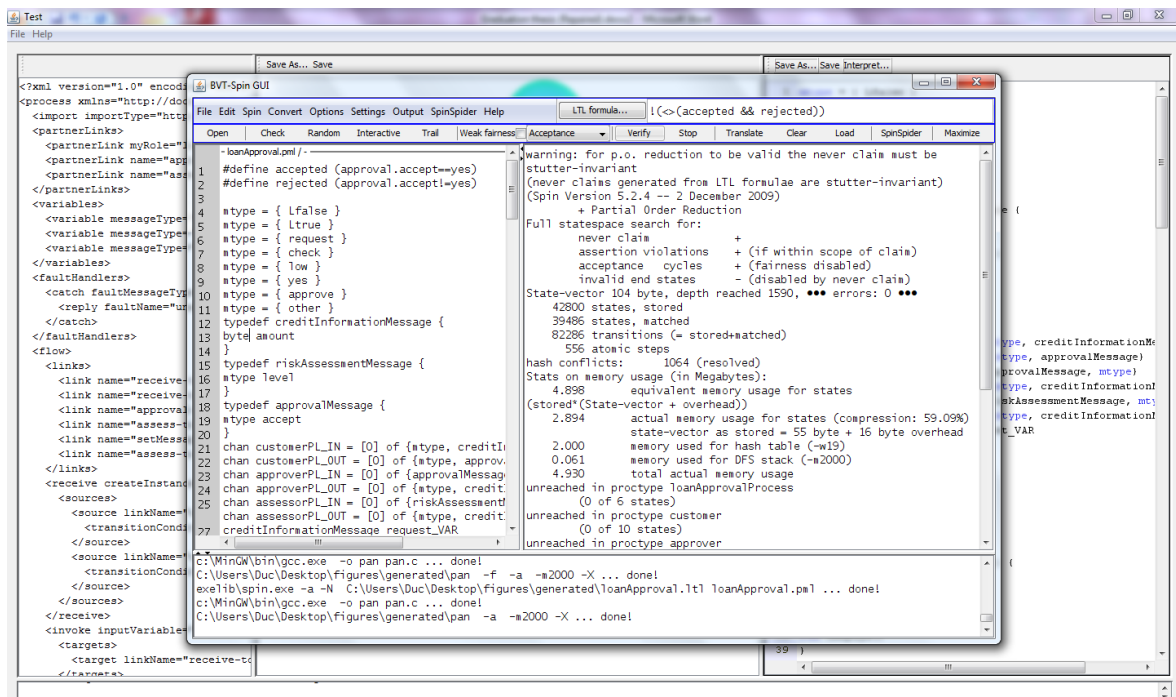


Figure 4-6 BVT screenshot - verifying the generated Promela program with a property

4.3. Evaluations

4.3.1. Evaluation of proposed algorithms

In this section, I compare the features my proposed solution with other approaches done by other researchers could provide: control flow handling, data handling, synchronization dependencies handling and XPath translation. This comparison is shown in Table 4.4.

All of the approaches are able to verify control flow of BPEL processes because it's essential part of any process. Some of approaches use an abstraction of data, i.e. variables, and don't analyze XPath expressions. Although they are only able to verify control flow such as researches [17] and [18] but they are very general and fully developed.

In this table, only 2 approaches done by Xiang Fu et al. (i.e. [19] and [20]) and my solution contain data and XPath expressions handling. To handle XPath expressions, they use CUP parser generator while I used JavaCC parser generator. My solution also takes into account synchronization dependencies handling as they did.

In those papers, [26] includes a generated Promela source code for loanApproval example in [6], but BPEL process was not as fully translated as my solution does because it does not contain randomization and include operations on partner links. Other papers with the same approach do not include generated Promela code and performance so I could not make comparison on these criteria.

No.	Name	Control flow handling	Data handling	Links (Synchronization dependencies) handling	XPath expressions handling
1	Model-based Verification of Web service Compositions [17]	yes	no	no	no
2	Describing and Reasoning on Web Services using Process Algebra [18]	yes	no	no	no
3	Analysis of Interacting BPEL Web Services [19]	yes	yes	yes	yes
4	Model Checking Interactions of Composite Web Services [20]	yes	yes	yes	yes
5	Transforming BPEL to Petri nets[21]	yes	no	yes	no
6	Verifying Web Services Composition Based on Hierarchical Colored Petri Nets[22]	yes	no	yes	no
7	Model-Checking Behavioral Specification of BPEL Applications [23]	yes	no	yes	no
8	LTSA-WS: a tool for model-based verification of web service compositions in Eclipse [24]	yes	no	no	no
9	Towards automatic verification of web-based SOA applications [25]	yes	no	no	no
10	A Methodology and a Tool for Model-based Verification and Simulation of Web Services Compositions [26]	yes	yes	yes	no
11	BVT tool	yes	yes	yes	yes

Table 4.4 Solutions comparison

Author in [38] also divided approaches in 2 main directions. Some of them concentrate fully on the control flow level and use abstraction of data. This first approach is quite successful and some automated tools have been developed such as LTSA-WS [24]. The second approach includes processing XML Schema data types and XPath expressions. While some results have been established, verification is limited to very simple processes.

As the table 4.1 shows, my solution has the following advantages:

- The proposed solution contains the features of other algorithms. Besides being able to verify the control flow, it can handle data in XML Schema data types and XPath expressions and links (synchronization dependencies).

- The use of LCFG as intermediate form and GUI for Spin may help students who study BPEL and model checking method to understand BPEL control flow and the method more easily.
- The Spin model checker is a very popular model checker so there may be more support and concern for this solution from other researchers.
- Generated Promela source code is readable (instead of hardly readable codes from automata) so it can be edited manually to satisfy users' need.

However, the proposed solution has some disadvantages:

- Mapping rules for some activities and BPEL elements are still not fully developed.
- Data handling and XPath processing rules have not covered all cases.

4.3.2. Evaluation of BPEL verification tool

The BPEL Verification Tool has the following advantages and disadvantages as follows:

Advantages:

- It uses all open source, well developed libraries and development tools so it costs no license fee and libraries are updated regularly.
- It has simple graphical user interface with user-friendly features such as syntax highlighting and interactive graph display that helps new users use it more easily.
- It supports various output formats for LCFG so users can use other tools to analyze the control flow of a BPEL process or other image processing.
- The tool is based on Java so it can be adopted on many different platforms.

Disadvantages:

- It requires users to install GNU C Compiler (gcc) if they want to verify the generated Promela program using Spin.

CHAPTER SUMMARY

In this chapter, I have described a test case for the tool, the verification result of generated program and screenshots of BVT. I also compared my solution with 9 other solutions. The comparison shows that my solution has equivalent features to other solutions.

CONCLUSIONS AND FUTURE WORKS

Established Achievements

This thesis has proposed a solution for the BPEL process verification problem and a tool that implements the solution.

Specific results of this thesis (aligned to chapters) are:

- Understanding of related theoretical foundations and tools:
 - o SOA, Web Service, BPEL, model checking and some formal models used in model checking.
 - o BPEL language which is designed for compose web services in orchestration manner and some tools for BPEL process development and execution.
 - o Spin, a specific model checker and Promela, a modeling language for Spin.
- Algorithms for transforming BPEL processes to Promela programs over a graph representation. My translation from BPEL to Promela language via LCFG form still has not been done by other researchers.
- A metamodel for processing BPEL documents based on JAXB open source library for binding XML schemas and Java representations.
- A model for constructing and displaying graphs based on JGraphT and JGraph5 open source graph libraries.
- A metamodel for Promela source code generation based on grammar rules of Promela language.
- Methods to process XML Schema data types and XPath expressions.
- BPEL Verification Tool, a tool that realizes above algorithms and models and helps users to verify BPEL processes via a graphical user interface.

Future Works

Some improvements should be done are as follows:

- Adding more rules into the algorithms so that it can model more activities at higher completion level.
- Adding modeling features of exception and fault handlers into algorithms.
- Adding a feature that supports users who has a little knowledge about Linear Temporal Logic to specify properties in an easier manner, i.e. users can specify queries in a language that can be translated into LTL formulae and embed auxiliary variables as well as assertions by interacting with the LCFG on GUI.

Conclusion

The need of verifying properties of BPEL processes becomes more popular as Web Service architecture and BPEL standards are more widely adopted. This thesis has proposed a method for the problem of BPEL process verification using the Spin model checker and developed a tool that realized the solution. So this thesis has achieved all of its initial objectives.

Due to the limit of time, the proposed solution still has some flaws while it also showed future development directions. For the demand of reliable distributed software is increasing, this approach would have limitless potential application in near future.

Appendix A. Paper (Vietnamese)

This is my paper that won a minor honor and was published in the proceedings of Conference of Scientific Research for students in School of Information and Communication Technology 2010.

Kiểm Định Tiến Trình BPEL Sử Dụng Trình Kiểm Tra Mô Hình SPIN

Bùi Hoàng Đức

Tóm tắt-- BPEL là một chuẩn phổ biến cho việc tích hợp các dịch vụ web. Người dùng có thể sử dụng BPEL để phối hợp các dịch vụ web theo một tiến trình nghiệp vụ và tạo nên một dịch vụ web mới. Vấn đề đặt ra là tính đúng đắn của các tiến trình BPEL cần phải được kiểm định. Trong công trình này, em đề xuất một cách tiếp cận cho việc kiểm định các tiến trình BPEL 2.0 bằng cách sử dụng trình kiểm tra mô hình SPIN. Phương pháp của em bao gồm việc biên dịch chương trình BPEL sang chương trình Promela thông qua đồ thị luồng điều khiển có gắn nhãn. Trong phương pháp này, em xử lý được các phụ thuộc đồng bộ hoá trong ngôn ngữ BPEL và hiển thị trực quan tiến trình BPEL nhằm hỗ trợ người nhà phát triển sử dụng SPIN một cách dễ dàng.

*Từ khóa—*BPEL, KIỂM ĐỊNH, Promela, SPIN.

1. GIỚI THIỆU

Xây dựng hệ thống phần mềm dựa trên web service đã mang lại lợi ích to lớn trên nhiều khía cạnh khác nhau như chi phí, mức độ rủi ro, thời gian, bảo trì ... Trong đó, việc tích hợp các web service làm việc theo đúng kịch bản nghiệp vụ là một yêu cầu quan trọng và BPEL (Business

Process Execution Language) đã được xây dựng để phục vụ cho mục tiêu này [8].

Một yêu cầu được đặt ra khi xây dựng tiến trình BPEL là kiểm tra tính chính xác của chúng. Hiện nay, đã có nhiều nghiên cứu đề cập tới vấn đề này [1]. Những nghiên cứu này thường dịch chuyển tiến trình BPEL sang một định dạng khác như automata [2], EFA [3] hay Petri net [4]; sau đó, sử dụng các model checker để kiểm tra thuộc tính cần thiết của tiến trình BPEL. Đối với đa số người phát triển phần mềm, các nghiên cứu này khá khó hiểu và đòi hỏi nhiều kiến thức về model checking.

SPIN [7] là một model checker được sử dụng khá phổ biến. Các chương trình được viết bằng ngôn ngữ Promela sẽ được SPIN sử dụng để kiểm tra. Mặc dù, người ta có thể dịch chuyển trực tiếp từ tiến trình BPEL sang ngôn ngữ Promela để kiểm tra. Nhưng cách làm này không giúp

^Bùi Hoàng Đức, sinh viên lớp Công Nghệ Phần Mềm, khóa 50, Viện Công nghệ thông tin và Truyền thông, trường Đại học Bách Khoa Hà Nội (điện thoại: (+84) 972347051, e-mail: ducbuihoang@gmail.com).

© Viện Công nghệ thông tin và Truyền thông, trường Đại học Bách Khoa Hà Nội.

người sử dụng hiểu rõ đặc tả tiến trình BPEL và mô hình cần kiểm tra.

Vì vậy, trong bài báo này, tôi đề xuất một phương pháp kiểm tra tiến trình BPEL thông qua SPIN model checker. Tiến trình BPEL được dịch chuyển sang định dạng đồ thị một cách trực quan. Các thông tin chính của tiến trình BPEL vẫn được bảo tồn và sắp xếp một cách hợp lý trên đồ thị. Tiếp theo, định dạng đồ thị sẽ được dịch chuyển sang ngôn ngữ Promela và sẽ được kiểm tra bằng SPIN model-checker. Trong quá trình dịch chuyển này, tôi tập trung giải quyết các vấn đề liên quan đến quan hệ đồng bộ hóa trong một tiến trình BPEL.






Các phần còn lại của bài báo được tổ chức như sau: phần 2 đề xuất phương pháp dịch chuyển từ BPEL sang đồ thị. Phần 3 trình bày phương pháp dịch chuyển từ đồ thị sang ngôn ngữ Promela. Việc cài đặt được trình bày trong phần 4. Tôi đưa ra một case-study để minh họa cho toàn bộ phương pháp này trong phần 5. Cuối cùng, phần 6, là kết luận và hướng phát triển.

2. DỊCH CHUYỂN TỪ BPEL SANG ĐỒ THỊ LUỒNG ĐIỀU KHIỂN CÓ GÁN NHÃN

Tiến trình BPEL đặc tả quy trình nghiệp vụ tương tự như flow-chart. Mỗi thành phần trong quy trình này được gọi là một hoạt động. Mỗi hoạt động có thể là hoạt động cơ bản hoặc một

hoạt động có cấu trúc, chứa các hoạt động khác. Do đó, một cách hình thức chúng ta định nghĩa đồ thị có hướng gán nhãn (Labeled Flow Control Graph, LFCG) như sau: LFCG (V,E) là một đồ thị có hướng, trong đó V là các đỉnh (nút) và E là tập các cung có hướng thể hiện sự dịch chuyển điều khiển giữa các nút.

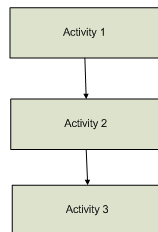
Các nút trong tập nút V có thể được phân loại thành nhiều loại nút, được thể hiện trong Bảng 1.

Loại đỉnh	Thể hiện
Nút bắt đầu (kết thúc)  Nút bắt đầu Nút kết thúc	Bắt đầu (kết thúc) của một quy trình hoặc một hoạt động có cấu trúc
Nút Fork (Join) 	Bắt đầu (kết thúc) của một hoạt động <flow>
Nút điều kiện 	Điều kiện của một hành động rẽ nhánh hoặc lặp
Nút hành động 	Một hành động cơ bản
Cung dịch chuyển 	Thể hiện sự dịch chuyển điều khiển giữa các đỉnh.

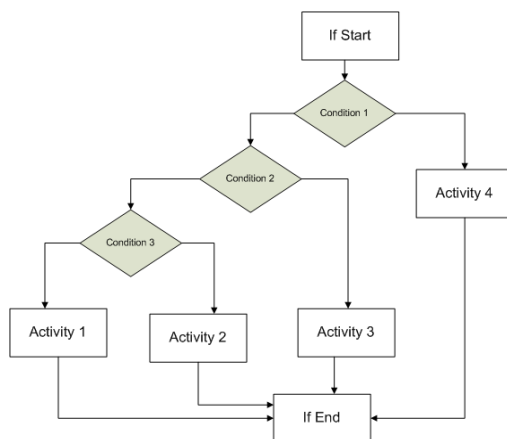
Bảng 1 Các loại nút trong LFCG

BPEL có 7 hoạt động có cấu trúc, trong bài báo này, tôi chỉ xét 5 loại hoạt động có cấu trúc sau:

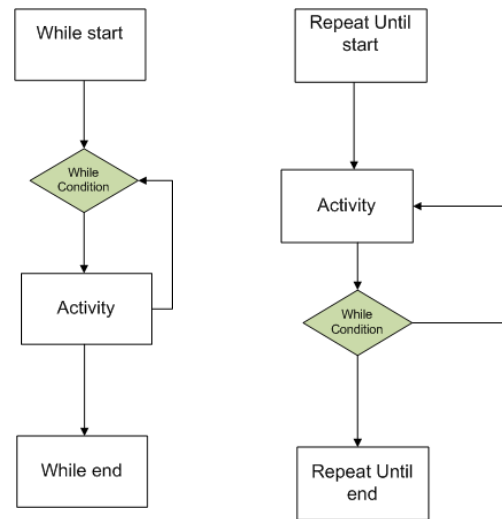
Sequence activity: Thực hiện một chuỗi các hành động tuần tự.



If activity: Hoạt động rẽ nhánh. Nhánh được thực hiện khi thỏa mãn điều kiện.



While activity, Repeat Until activity : Hoạt động lặp khi điều kiện đúng.



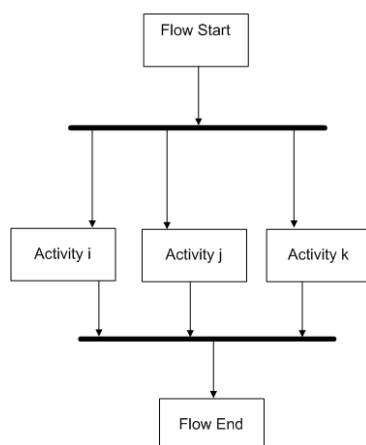
Flow activity: Mô tả các hoạt động song song. Mỗi nhánh của hoạt động này được thực hiện độc lập nhau. Mỗi nhánh có thể là một hoạt động cơ bản hoặc có cấu trúc.

Tuy nhiên, có thể xuất hiện các phụ thuộc đồng bộ hóa được mô tả trong element <link>. Các hoạt động với element <source> hoặc <target> là điểm bắt đầu hoặc kết thúc một link. Hoạt động được gán nhãn <target> chỉ được thực hiện khi mà joinCondition được đánh giá là true. Nếu thuộc tính joinCondition không có thì điều kiện hợp là OR logic của trạng thái của tất cả các link đi vào hoạt động đó. Trạng thái của một link có thể là 'true', 'false' hoặc 'unset'. Do đó, một hoạt động được thực hiện nếu có một link có trạng thái là true. Nếu thuộc tính suppressJoinFailure có giá trị 'true' thì một lỗi bpel:joinFailure sẽ xảy ra và được bắt bởi một bộ xử lý lỗi, ngược lại, nếu suppressJoinFailure là 'false' thì sẽ không fault xảy ra và các trạng thái của các link đi ra sẽ có giá trị là false. Mỗi hoạt động có thể được

gán một hoặc nhiều nhãn <source> hoặc <target>. Ngoài ra, nhãn <source> có thể chứa thành phần **transitionCondition** nhằm thể hiện điều kiện để cho các link đi ra được có giá trị true.

Nếu trong một tiến trình mà joinCondition bị bỏ đi và suppressJoinFailure có giá trị 'false' thì có thể coi các hoạt động trong mỗi nhánh của Flow activity sẽ trở thành tuần tự nếu chúng là source và target của một link. Hoạt động có nhãn source xuất hiện trước và hoạt động có nhãn target xuất hiện sau. Mỗi hoạt động có bao nhiêu nhãn <source> thì sẽ có từng đó cung xuất phát từ nó. Tên của các cung này chính là tên link. Nếu trong nhãn <source> có thành phần **transitionCondition** thì bổ sung thêm nút điều kiện giữa link này.

Một Flow activity được biểu diễn bởi nút fork và join. Trong đó, nhãn của nút fork lưu thông tin về các link xuất hiện trong Flow activity đó.



3. DỊCH CHUYỂN TỪ ĐỒ THỊ SANG NGÔN NGỮ PROMELA

Để phục vụ cho quá trình kiểm tra, cấu trúc đồ thị LCFG cần được dịch chuyển sang chương trình Promela. Mỗi chương trình Promela chứa 3 thành phần chính: tiến trình, kênh thông điệp và biến. Cấu trúc chính của đồ thị sẽ được dịch chuyển sang tiến trình chính, còn nhãn của các nút là những thông tin cần thiết hỗ trợ cho việc định nghĩa kênh thông điệp và các biến.

Các biến trong chương trình Promela được từ các biến trong tiến trình BPEL. Ở đây, để sinh ra các biến trong Promela, tôi chỉ xét đến văn bản BPEL thay vì xét đến các văn bản WSDL và XSD tham chiếu đến từ văn bản BPEL trên vì như vậy sẽ khiến cho việc biên dịch trở nên hiệu quả hơn. Đầu tiên, rất nhiều thành phần trong biến có thể không được dùng đến trong tiến trình BPEL, nên vẫn đảm bảo ngữ nghĩa của tiến trình BPEL khi chuyển sang Promela. Ngoài ra, các văn bản WSDL có chứa nhiều định nghĩa kiểu khác cũng như tham chiếu đến các văn bản WSDL và XSD khác, khiến cho việc xử lý khá phức tạp. Nếu chỉ xét trong 1 file BPEL thì ta cũng có đủ thông tin để mô hình hoá thành chương trình Promela. Cách tiếp cận này đã được sử dụng trong [5].

Kiểu dữ liệu của biến được xác định dựa vào ý nghĩa của biến trong tiến trình. Ví dụ như trong tiến trình loanApproval [8], có biến request, kiểu creditInformationMessage. Trước

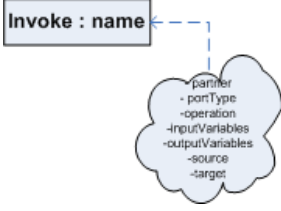
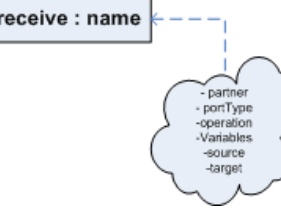
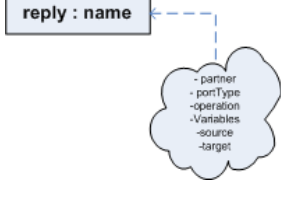

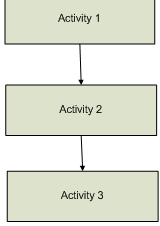
hết, ta định nghĩa kiểu typedef creditInformationMessage. Khi duyệt, gặp biểu thức truy cập vào trong thành phần của biến '\$request.amount < 10000', với vế phải là số, ta sẽ bổ sung thêm vào kiểu dữ liệu với thành phần amount, kiểu short (vì 10000 có thể biểu diễn bởi kiểu short). Ta có đoạn code Promela:

```
typedef creditInformationMessage{
    short amount;
};
```

Mỗi portType ứng với 2 kênh thông điệp là in channel và out channel. In channel phục vụ cho việc nhận thông điệp và Out channel phục vụ cho việc gửi thông điệp của tiến trình chính. Kiểu của channel là kiểu dữ liệu được sử dụng trong portType.

Mỗi đồ thị tương ứng với một proc chính của chương trình. Nguyên tắc dịch chuyển từ đồ thị LCG sang câu lệnh Promela được xây dựng dựa trên sự tương đồng về ngữ nghĩa. Bên cạnh tiến trình chính, ta còn phải tạo ra các proctype khác đại diện cho các web service bao hàm trong tiến trình và web service gọi tiến trình.

Bảng 2 cho ta các ánh xạ chính từ các cấu trúc đồ thị sang ngôn ngữ Promela.

Nút	Câu lệnh
	<pre>portType_OUT ! output_var portType_IN ? input_var</pre> <p>Nhận và gửi thông tin giữa các biến và kênh truyền.</p>
	<pre>portType_IN ? variable</pre> <p>Nhận dữ liệu từ kênh vào biến</p>
	<pre>portType_OUT ! variable</pre> <p>Gửi dữ liệu từ biến ra kênh</p>
	<pre>to=from</pre> <p>Khi gán, có chú ý tới các thành phần dữ liệu của biến hoặc phân tích biểu thức</p>
	<p>Câu lệnh tuần tự</p>

	<pre>If :: Cond1 -> :: ... :: else -> fi</pre>
	<pre>Do :: Cond -> ... Od</pre>
	<p>(1) Mỗi nhánh tương ứng với một proc.</p> <p>(2) Định nghĩa từng proc theo các nguyên tắc trong bảng này.</p> <p>(3) Trong proc chính, gọi đến các proc đã định nghĩa ở phía trên. Sử dụng câu lệnh run.</p>

Bảng 2 Biểu diễn trên ngôn ngữ Promela

Giải thuật sau đây mô tả quá trình duyệt đồ thị và dịch chuyển sang ngôn ngữ Promela.

Bước 1: Khởi tạo phần khai báo và tiến trình chính.

Bước 2: Duyệt đồ thị LCG

Bước 2.1 : Nhận 1 nút trong đồ thị

Bước 2.2 : Xác định loại nút và sử dụng nguyên tắc dịch chuyển trong bảng 1

Bước 2.3 : Nếu là nút đại diện cho structured activity, quay lại bước 2.1. Nếu không, nhảy sang bước 2.4.

Bước 2.4: Xét nhãn của nút, bổ sung biến, channel và kiểu dữ liệu vào phần khai báo.

Bước 3: Tổng hợp phần khai báo

Bước 3.1 : Xác định các kiểu dữ liệu typedef và mtype

Bước 3.2 : Khai báo biến

Bước 3.3 : Khai báo channel

// Quá trình duyệt là tuần tự

Với các link trong hoạt động flow, nếu không có joinCondition và suppressJoinFailure là 'true' thì có 2 lựa chọn cho người dùng:

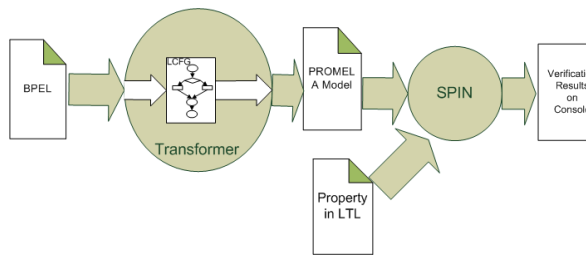
(3.1) Giữ nguyên cấu trúc song song bằng cách chuyển các activity trong flow thành các proctype.

(3.2) Chuyển sang cấu trúc tuần tự bằng cách thêm các nút điều kiện.

4. CÀI ĐẶT

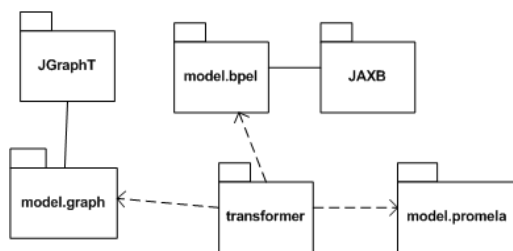
Trong phần này, chúng tôi trình bày về kiến trúc tổng quát của công cụ được xây dựng nhằm phục vụ quá trình dịch chuyển và kiểm tra tiến

trình BPEL. Kiến trúc chính của công cụ được mô tả trong Hình 1.



Hình 1 Các thành phần của công cụ

Thành phần chính của công cụ này là phần Transformer. Đây là thành phần chịu trách nhiệm dịch chuyển giữa các định dạng. Đầu vào của thành phần này là một tiến trình BPEL và kết quả mà phần Transformer trả ra là một chương trình Promela tương ứng với tiến trình đó. Với vai trò đó, thành phần này chứa các gói tương ứng với nhiệm vụ dịch chuyển giữa các mô hình.



Trong cài đặt của chương trình, tôi sử dụng các mô hình lớp (object model) để thao tác với các văn bản BPEL, biểu diễn LCFG và sinh ra chương trình Promela.

Mô hình BPEL gồm các gói model.bpel.abs và model.bpel.exe

chứa hàng trăm lớp được sinh ra từ XML Schema của BPEL 2.0 cho các tiến trình trừu tượng và khả chạy bằng cách sử dụng thư viện JAXB [9].

Mô hình LCFG được cài đặt bởi gói model.graph. Trong đó, lớp quan trọng nhất là lớp LCFG, kế thừa lớp đồ thị có hướng trong thư viện JGraphT [11], thể hiện một đồ thị luồng điều khiển có gắn nhãn.

Mô hình Promela được xây dựng từ văn phạm của ngôn ngữ Promela [10]. Trong đó, có 3 gói class chính: model.promela chứa các lớp thể hiện các ký tự chưa kết thúc của ngôn ngữ; model.promela.literal chứa các lớp thể hiện các ký tự kết thúc; model.promela.op chứa các lớp thể hiện các toán tử.

Ngoài ra, các thuật toán dịch được cài đặt và thao tác trên các mô hình trên.

5. VÍ DỤ MINH HOẠ

Để minh họa cho quá trình chuyển dịch qua các định dạng và kiểm tra các thuộc tính của tiến trình BPEL, tôi sử dụng tiến trình “Loan Approval” [8]. Quan sát ban đầu có thể khiến người dùng nghĩ tiến trình này chỉ bao gồm các hoạt động được thực hiện song song. Tuy nhiên, khi phân tích kỹ hơn, chúng ta thấy rằng các hoạt động này có ràng buộc với nhau về quan hệ đồng bộ hóa. Phần sau đây, tôi sẽ mô tả các bước của quá trình chuyển dịch sang các dạng biểu

diễn khác nhau và kiểm tra một thuộc tính cần thiết của tiến trình này.

Bước thứ nhất cần thực hiện là biểu diễn tiến trình BPEL trên dưới dạng đồ thị LCFG. Các quy tắc dịch chuyển được mô tả trong phần hai. Toàn bộ tiến trình này sẽ được biểu diễn bởi một đồ thị LCFG như Hình 2.

Sau khi xây dựng đồ thị LCFG của tiến trình BPEL, bước tiếp theo là dịch chuyển sang ngôn ngữ Promela. Ở đây, các hoạt động flow được ánh xạ theo cách (3.1). Một kết quả ánh xạ tương tự có thể tham khảo trong [5]. Chương trình có thể tạo ra chương trình Promela với cách ánh xạ (3.2) nhưng tôi không trình bày ở đây.

Kết quả là tạo ra được một chương trình Promela với các biến và kiểu dữ liệu

```
mtype = {low};
mtype = {yes};
mtype = {other};
mtype = {Ltrue, Lfalse, unset};
typedef creditInformationMessage{
    short amount; };
typedef approvalMessage{
    short accept;};
typedef riskAssessmentMessage{
    short level;};
chan customerPL = [0] of
{creditInformationMessage};
chan approverPL = [0] of
{approvalMessage};
chan assessorPL = [0] of
{riskAssessmentMessage};
creditInformationMessage request;
approvalMessage approval;
riskAssessmentMessage risk;
short rec_to_assess=unset;
short rec_to_app=unset;
short app_to_rep=unset;
```

```
short assess_to_setmsg=unset;
short setmsg_to_rep=unset;
short assess_to_app=unset;
```

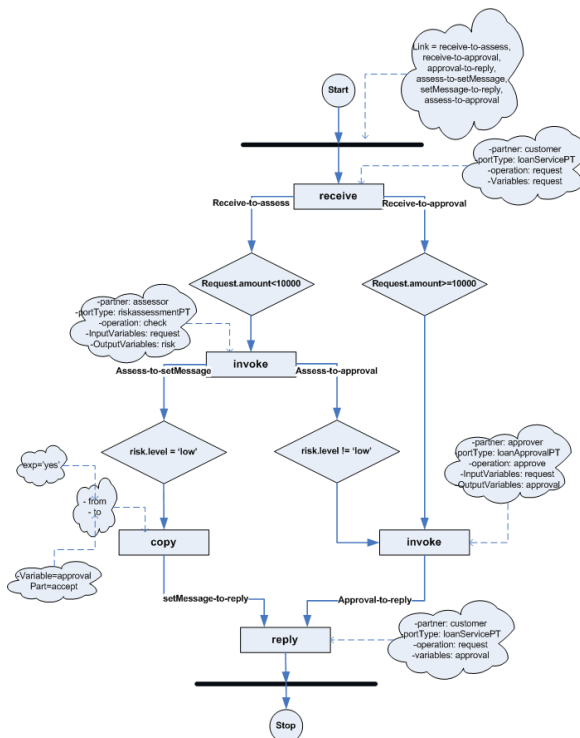
Chương trình sinh ra gồm có 8 proctype. Trong đó, 5 proctype là Receive1(), Invoke1(), Assign1(), Invoke2() và Reply1() xuất phát từ 5 activity trong flow activity. Proctype loanApproval() sẽ chạy 5 tiến trình này

```
proctype loanApproval() {
    run Receive1();
    run Invoke1();
    run Assign1();
    run Invoke2();
    run Reply1();
}
```

Bên cạnh đó có 3 proctype customer(), approver() và assessor() xuất phát từ các portType – các web service tương tác với tiến trình.

Cuối cùng, tiến trình init() sẽ khởi động chương trình Promela.

```
init{
    run loanApproval();
    run customer();
    run approver();
    run assessor();
}
```

Hình 2 Minh hoạ đồ thị LCFG tương ứng với tiến trình loanApproval

Người sử dụng có thể đặt các câu hỏi dưới dạng biểu thức LTL để kiểm tra tiến trình có thỏa mãn các câu hỏi đó hay không. Ví dụ, trong tiến trình “Loan approval”, một câu hỏi được đặt ra là “Với hai yêu cầu như nhau, có thể xảy ra trường hợp lúc thì được chấp nhận, lúc thì bị kiểm tra thêm hay không?”.

Để trả lời câu hỏi này, chúng ta khai báo một biến result kiểu int. Khi tiến trình bắt đầu được thực hiện gán result bằng 0. Nếu được yêu cầu được chấp nhận (tức là sau hoạt động gán) thì giá trị của result bằng 1. Nếu yêu cầu bị kiểm tra thêm (tức là sau hoạt động invoke của approver) thì giá trị của result bằng 2. Ngoài các câu lệnh khai báo và thay đổi giá trị của biến result, toàn bộ câu hỏi này sẽ được mô tả như sau:

```
#define accepted (result==1)
#define rejected (result==2)
!(<>(accepted && rejected))
```

Kết quả kiểm tra được thể hiện trong Bảng 3

```
(Spin Version 5.2.4 -- 2 December 2009)

+ Partial Order Reduction

Full statespace search for:

    never claim - (none specified)
    assertion violations +
    cycle checks - (disabled by -DSAFETY)
    invalid end states +

State-vector 96 byte, depth reached 65561, ... errors: 0 ...

    1667332 states, stored
    65537 states, matched
    1732869 transitions (= stored+matched)
    0 atomic steps

hash conflicts: 959894 (resolved)

    218.113 memory usage (Mbyte)

unreached in proctype loanApproval
    (0 of 18 states)

unreached in proctype customer
    (0 of 10 states)

unreached in proctype assessor
    (0 of 10 states)

unreached in proctype approver
    (0 of 10 states)

unreached in proctype :init:
    (0 of 5 states)

pan: elapsed time 2.69 seconds
pan: rate 620518.05 states/second
```

Bảng 3 Kết quả kiểm định thuộc tính

6. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Trong bài báo này, tôi đã đề xuất một phương pháp kiểm tra tiến trình BPEL một cách trực quan. Việc dịch chuyển tiến trình BPEL sang dạng đồ thị LCG giúp người sử dụng dễ dàng nắm bắt được tiến trình và loại bỏ đi những thông tin không cần thiết cho quá trình kiểm tra. Ngoài ra, chúng tôi đã đề xuất giải pháp đối với các hoạt động đồng bộ hóa.

Trong tương lai, tôi sẽ tiếp tục hoàn thiện việc chuyển đổi các activity liên quan đến việc bắt lỗi trong tiến trình BPEL và hỗ trợ người sử dụng trong việc xây dựng các biểu thức LTL một cách dễ dàng hơn.

7. LỜI TRI ÂN

Em xin gửi lời cảm ơn sâu sắc đến phó giáo sư, tiến sĩ Huỳnh Quyết Thắng, giảng viên Viện Công Nghệ Thông Tin và Truyền Thông, Đại học Bách Khoa Hà Nội và thạc sĩ Phạm Thị Quỳnh, giảng viên đại học Sư Phạm Hà Nội đã tận tình giúp đỡ em thực hiện công trình này.

TÀI LIỆU THAM KHẢO

- [1] Franck van Breugel, Maria Koshkina. “Models and Verification of BPEL”. Unpublished Draft, September, 2006.
- [2] Xiang Fu, Bultan, Jianwen Su. “Analysis of Interacting BPEL Web Services”. Proceedings of the 13th international conference on World Wide Web. Pages: 621 – 630. 2004
- [3] Shin Nakajima. “Lightweight formal analysis of web service work flows”. Progress in Informatics, 2005.
- [4] ChristianStahl. “A Petri net semantics for BPEL”.2004

- [5] José García-Fanjul, Javier Tuya, Claudio de la Riva. “Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN”. International Workshop on Web Services. 2006
- [6] Al-Gahtani Ali, Al-Muhaisen Badr, Dekdouk Abdelkader. “A Methodology and a Tool for Model-based Verification and Simulation of Web Services Compositions”. International Conference on Information and Communications Security.2004.
- [7] Spin - Formal Verification. <http://spinroot.com>
- [8] OASIS Web Services Business Process Execution Language (WSBPEL) TC. <http://www.oasis-open.org/committees/wsbpel/>
- [9] JAXB Reference Implementation. <https://jaxb.dev.java.net/>
- [10] Promela GRAMMAR. <http://spinroot.com/spin/Man/grammar.html>
- [11] JGraphT, a graph library. <http://www.jgrapht.org>

Appendix B. Some Tools And Libraries Used In The Thesis

B.1. JAXB

JAXB stands for Java Architecture for XML Binding which is a library in Java that help application to access XML documents. JAXB has two main features: marshalling Java objects into XML and marshalling XML back into Java objects. In other words, JAXB allows storing and retrieving data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure.

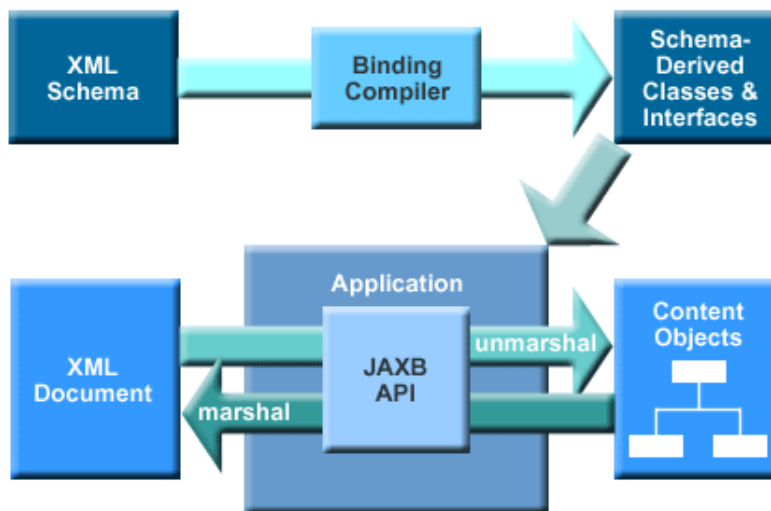


Figure B-1 JAXB Architecture

B.2. JGraphT

JGraphT [39] is “a free Java graph library that provides mathematical graph-theory objects and algorithms. JGraphT supports various types of graphs including:

- directed and undirected graphs.
- graphs with weighted / unweighted / labeled or any user-defined edges.
- various edge multiplicity options, including: simple-graphs, multigraphs, pseudographs.
- unmodifiable graphs - allow modules to provide "read-only" access to internal graphs.
- listenable graphs - allow external listeners to track modification events.
- subgraphs graphs that are auto-updating subgraph views on other graphs.
- all compositions of above graphs.”

JGraphT library focuses on data structures to represent graphs in memory and algorithms of graph traversal and manipulation.

B.3. JGraph

JGraph 5 [40] is a Swing-compliant, open source (BSD) graph component for Java. This library focuses on rendering; automated lay outing and editing graphs on Swing GUI.

B.4. Jspin

jSpin [41] is a graphical user interface for the Spin model checker. The program is an open source (GNU) Java library. The user interface of jSpin is quite simple with menus, a toolbar and three adjustable text areas but provide controls for most of functionalities of the Spin. New Spin users can use it to interact with Spin easily instead of using the command line interface of the Spin.

Appendix C. Javadoc of Main Packages in BVT

Transformer	
transformer.bl	Contains classes that transform BPEL documents into LCFGs.
transformer.lp	Contains classes that transform LCFGs into PROMELA programs.

BPEL Model	
model.bpel.abs	Contains classes that represent elements in abstract BPEL processes.
model.bpel.exe	Contains classes that represent elements in executable BPEL processes.

Graph Model	
model.graph	Contains model classes of LCFG

PROMELA Model	
model.promela	Contains model classes of PROMELA language
model.promela.literal	Contains model classes of literal characters in PROMELA language
model.promela.literal.op	Contains model classes of operations in PROMELA language

Test cases	
testcase	Contains test classes.
testcase.transformation	Contains test classes for transformations.

GUI	
jspin.filterSpin	Classes adapted from jSpin filterSpin.
jspin.jspin	Classes adapted from jSpin.
jspin.spinSpider	Classes adapted from jspin spinSpider.
ui	Contains GUI classes.

Other Packages	
exceptions	Classes representing exceptions in BVT.
jsyntaxpane	Classes adapted from jsyntaxpane library.
jsyntaxpane.actions	
jsyntaxpane.components	
jsyntaxpane.lexers	
jsyntaxpane.syntaxkits	
jsyntaxpane.util	
parser.xpath2	Lexical analyzer and parser for XPath 2.0 expressions.
util	Contains utility classes that are used by JVT.

Bibliography

- [1] Vu Thi Huong Giang, “Service-Oriented Software Engineering, Lecture Notes,” 2009.
- [2] Hugo Haas, “Designing the architecture for Web services,” May. 2003.
- [3] Philippe Le Hégarret, “Introduction to Web Services,” Mar. 2003.
- [4] W3C Recommendation, “SOAP Version 1.2” Available: <http://www.w3.org/TR/soap12-part1/>.
- [5] Wikipedia, the free encyclopedia, “Comparison of BPEL engines” Available: http://en.wikipedia.org/wiki/Comparison_of_BPEL_engines.
- [6] OASIS Standard, “Web Services Business Process Execution Language Version 2.0,” *Web Services Business Process Execution Language Version 2.0* Available: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
- [7] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Higher Education, 2001.
- [8] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*, 1999.
- [9] Mordechai (Moti) Ben-Ari, *Principles of the Spin Model Checker*, Springer, 2008.
- [10] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.
- [11] Abdulla Eid, *Finite omega-Automata and Buchi Automata*, University of Illinois, Department of Computer Science, 2009.
- [12] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004.
- [13] Spinroot, “Promela grammar” Available: <http://spinroot.com/spin/Man/grammar.html>.
- [14] Franck van Breugel and Maria Koshkina, “Models and Verification of BPEL,” 2006.
- [15] Carnegie Mellon University, “The SMV System” Available: <http://www-2.cs.cmu.edu/~modelcheck/smv.html>.
- [16] “NuSMV: a new symbolic model checker” Available: <http://nusmv.irst.itc.it/>.
- [17] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer, “Model-based Verification of Web service Compositions,” *Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference(ASE)*, 2003.
- [18] GwenSalaun, LucasBordeaux, and MarcoSchaerf, “Describing and Reasoning on Web Services using Process Algebra,” *Proceeding to IEEE International Conference on Web Services 2004*, 2004.
- [19] Xiang FU, Tevfik Bultan, and Janwen Su, “Analysis of Interacting BPEL Web Services,” *UCSB Computer Science Department Technical Report 2004-2005*, 2005.
- [20] Xiang FU, Tevfik Bultan, and Janwen Su, “Model Checking Interactions of Composite Web Services,” *Proceedings of the 13th International World Wide Web Conference*, 2004.
- [21] Sebastian Hinz, Karsten Schmidt, and Christian Stahl, “Transforming BPEL to

- Petri nets,” *Proceedings of the 3rd International Conference on Business Process Management*, 2005.
- [22] YanPing Yang, QingPing Tan, and Yong Xiao, “Verifying Web Services Composition Based on Hierarchical Colored Petri Nets,” *ACM workshop on Interoperability of Heterogeneous Information Systems (IHIS'05)*, Bremen, Germany: 2005.
- [23] Shin NAKAJIMA, “Model-Checking Behavioral Specification of BPEL Applications,” *Electronic Notes in Theoretical Computer Science 151 (2006)* 89–105, 2006.
- [24] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer, “LTSA-WS: a tool for model-based verification of web service compositions in Eclipse,” *Proceeding of the 28th International Conference on Software Engineering*, Shanghai: 2006, p. 4.
- [25] Xiangping Chen, Gang Huang, and Hong Mei, “Towards automatic verification of web-based SOA applications,” *Asia-Pacific Web Conference 2008*, Shenyang, China: 2008.
- [26] AL-GAHTANI Ali, AL-MUHAISEN Badr, and DEKDOUK Abdelkader, “A Methodology and a Tool for Model-based Verification and Simulation of Web Services Compositions.”
- [27] Oracle Corporation, “Java API for XML Processing,” *jaxp: JAXP Reference Implementation* Available: <https://jaxp.dev.java.net/>.
- [28] Apache Software Foundation, “Xerces2 XML Parser for Java,” *Xerces2 Java Parser Readme* Available: <http://xerces.apache.org/xerces2-j/>.
- [29] JDOM Project, “Java Document Object Model,” *JDOM* Available: <http://www.jdom.org/>.
- [30] OASIS, “XML Schema for Abstract Process Common Base for WS-BPEL 2.0,” *Schema for Abstract Process Common Base for WS-BPEL 2.0* Available: http://docs.oasis-open.org/wsbpel/2.0/OS/process/abstract/ws-bpel_abstract_common_base.xsd.
- [31] OASIS, “XML Schema for Executable Process for WS-BPEL 2.0,” *Schema for Executable Process for WS-BPEL 2.0* Available: http://docs.oasis-open.org/wsbpel/2.0/OS/process/executable/ws-bpel_executable.xsd.
- [32] Marcus Alanen and Ivan Porres, *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*, Finlan: Abo Akademi University.Turku Centre for Computer Science, 2003.
- [33] W3C, “JITree file for XPath 2.0,” *XPath 2.0 Grammar Test Page* Available: <http://www.w3.org/2007/01/applets/xpath-grammar.jjt>.
- [34] AT&T Research, “Graphviz - Graph Visualization Software” Available: <http://www.graphviz.org/>.
- [35] Michael Himsolt, “GML: A portable Graph File Format” Available: <http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html>.
- [36] GraphML Team, “The GraphML File Format” Available: <http://graphml.graphdrawing.org/>.
- [37] ayman.alsairafi, “jsyntaxpane - Project Hosting on Google Code” Available: <http://code.google.com/p/jsyntaxpane/>.
- [38] Frédéric Servais, “Verifying and Testing BPEL Processes,” *Université Libre de Bruxelles*, 2006.

- [39] JGraphT Project, “JGraphT - a free Java Graph Library,” *Welcome to JGraphT - a free Java Graph Library* Available: <http://jgrapht.sourceforge.net/>.
- [40] JGraph Ltd, “JGraph - The Java Open Source Graph Drawing Component - version 5,” *JGraph - The Java Open Source Graph Drawing Component* Available: <http://www.jgraph.com/jgraph5.html>.
- [41] Mordechai (Moti) Ben-Ari, “jspin - A graphical user interface for the Spin model Checker,” *The jSpin development environment for Spin* Available: <http://code.google.com/p/jspin/>.