

Swinburne University of Technology

Faculty of Information and Communication Technologies

HIT4000 Honours Project

A Thesis on

Using temporal business rules to verify and
guide service composition

Phan, Manh Tan (2293803)

BIS (Honours) Supervisors

Dr Yan Jin

Professor Jun Han

Abstract

Service Oriented Architecture (SOA) is the thriving paradigm for developing intra and inter-organizational enterprise applications. SOA fosters interoperability, reusability, flexibility, scalability and productivity in software development. In SOA, computerized business functionalities are exposed as web services to be used by clients from within or outside the organization. To develop complex functionalities, very often we need to compose single web services into composite web services. Service composition is thus becoming a dominant method for developing Web service applications.

When designing a service composition, it is important to ensure that the composition conforms to the requirements. Most of the existing conformance checking methods require the users to be able to write the requirement specifications in formalisms like temporal logics, etc. These methods are thus inaccessible to practitioners who often do not have a strong mathematical background. In addition, whenever the designated service composition does not conform to the set requirements, post development error corrections can be time consuming and troublesome. Therefore, it would be desirable to assist practitioners in developing a web service composition that conforms to the requirements in the first place.

Our research is aimed at providing an intuitive, user-friendly method for static verification and design time guidance for service composition. Some required properties for service composition are stated using *temporal business rules*, rules that govern the occurrences or sequencing of business activities and reflect the business logics employed by collaborating parties during the composition process. We then model-check the composite service against the rules and, based on the model-checking result, provide composition guidance to the service composer at each step of service composition.

The contributions of the thesis are as follows. It builds on the pattern-based property specification language PROPOLS for specifying temporal business rules. It presents a

framework for statically verifying service compositions against a set of temporal business rules. It introduces a method to provide automated guidance in forms of suggestions for inserting, removing, and reordering business activities to the service designer during the composition process. Finally, the thesis presents a prototype tool, called BPELCheck4Guide, which implements the proposed verification and guidance approach.

Acknowledgement

I would like to sincerely thank my supervisors Professor Jun Han and Dr Yan Jin for their vision, guidance, discussions and support during the course of this work and for their careful reading and appraisals of drafts of this thesis. I thank Dr Jian Yu, Institute of Computing Technology, Chinese Academy of Sciences, Beijing for his continuous help with the research work. I also want to thank Dr Howard Foster, Imperial College, London for his timely response with regards to the BPEL2LTS tool. Thanks also go to Zheng Li for his discussions and comments on the research work.

Declaration

This thesis contains no material which has been submitted for the award of any other degree or diploma. To the best of my knowledge, this thesis contains no material previously published or written by other people except where due reference is made.

Phan, Manh Tan

July 17, 2006

Table of Content

1	INTRODUCTION	11
1.1	THE RESEARCH QUESTIONS	11
1.2	THE APPROACH TAKEN	12
1.3	RESEARCH ACCOMPLISHMENTS	13
1.4	LIMITATIONS AND DELIMITATIONS	14
1.5	THESIS STRUCTURE	14
1.6	SUMMARY	17
2	BACKGROUND.....	18
2.1	SERVICE-ORIENTED ARCHITECTURE	18
2.1.1	<i>Web services technology.....</i>	<i>19</i>
2.1.2	<i>Semantic web services</i>	<i>20</i>
2.1.3	<i>Services composition</i>	<i>20</i>
2.1.4	<i>Service orchestration and service choreography</i>	<i>21</i>
2.1.5	<i>The BPEL language.....</i>	<i>21</i>
2.2	MODEL-CHECKING SERVICE COMPOSITION.....	23
2.2.1	<i>Model-checking.....</i>	<i>24</i>
2.2.2	<i>Application in service composition.....</i>	<i>24</i>
2.3	PROPERTY PATTERNS	24
2.3.1	<i>FSA semantics of property patterns.....</i>	<i>25</i>
2.3.2	<i>Applications</i>	<i>27</i>
2.4	OTHER RELATED WORK.....	27
2.5	SUMMARY	28
3	A MOTIVATING EXAMPLE	30
3.1	THE BUSINESS SCENARIO	30
3.2	BUSINESS ACTIVITIES EXPOSED AS WEB SERVICES OPERATIONS	31
3.3	BUILDING THE BUSINESS PROCESS	32
4	SPECIFICATION OF TEMPORAL BUSINESS RULES	33
4.1	ABOUT PROPOLS	33

4.2	THE LANGUAGE DESIGN	34
4.3	LOGICAL COMPOSITIONS OF PATTERNS.....	37
4.4	THE EXAMPLE RULES	38
4.5	FSA SEMANTICS OF THE RULES.....	42
4.6	SUMMARY	44
5	VERIFICATION OF BPEL SERVICE COMPOSITION	45
5.1	THE VERIFICATION FRAMEWORK.....	45
5.2	CONVERT TEMPORAL BUSINESS RULES TO COMPLEMENTARY TDFSA	47
5.3	REPRESENTING BPEL IN LTS	50
5.4	SEMANTIC MAPPING	52
5.5	CONVERTING LTS TO TDFSA	54
5.6	INTERSECTING COMPLEMENTARY OF RULE TDFSA AND BPEL TDFSA	56
5.7	VERIFICATION RESULTS OF THE EXAMPLE BPEL SCHEMA.....	60
5.8	SUMMARY	64
6	COMPOSITION GUIDANCE	65
6.1	THE GUIDANCE FRAMEWORK.....	65
6.2	AN EXAMPLE VIOLATION SCENARIO	66
6.3	VIOLATION ANALYSIS	67
6.3.1	<i>Identify error states</i>	67
6.3.2	<i>Identify error paths</i>	67
6.3.3	<i>Identify error-entry states and error-entry events</i>	69
6.4	AUTOMATIC REMEDY PLANS GENERATION.....	70
6.4.1	<i>Forward extension</i>	70
6.4.2	<i>Backward insertion</i>	71
	<i>Backward correction</i>	73
6.4.3	73
6.4.4	<i>Deletion and reordering</i>	76
6.5	PATTERN SPECIFIC GUIDANCE.....	76
6.6	EXAMPLE GUIDANCE RESULTS.....	78
6.7	SUMMARY	81
7	TOOL IMPLEMENTATION.....	82
7.1	DESIGN	82

7.2	PACKAGE “FSA”	83
7.3	PACKAGE “CHECKER”	84
7.4	PACKAGE “GUIDER”	85
7.5	PACKAGE “GUI”	86
7.6	SUMMARY	87
8	CONCLUSION	88
8.1	GOALS AND ACCOMPLISHMENTS	88
8.2	LIMITATIONS AND FUTURE WORK	89
APPENDIX 1: FORMAL MATHEMATICAL BACKGROUND FOR VERIFICATION AND GUIDANCE.....		91
APPENDIX 2: PROTOTYPE TOOL TECHNICAL DETAILS		96
APPENDIX 3: SAMPLE XML DEFINITIONS OF RULES AND BPEL SCHEMAS		100
APPENDIX 4: THE EXAMPLE WSDL FILE.....		106
APPENDIX 5: THE EXAMPLE BPEL SCHEMA.....		110
REFERENCES		113

List of figures

FIGURE 1.1	THE GENERAL APPROACH TO VERIFYING AND GUIDING SERVICE COMPOSITION [22]	12
FIGURE 2.1	SERVICE ORIENTED ARCHITECTURE: THE BIG PICTURE [32]	19
FIGURE 2.2	PPS PATTERNS [22]	25
FIGURE 2.3	SEMANTICS OF PATTERN “ P EXISTS AFTER Q ”	26
FIGURE 2.4	SEMANTICS OF PATTERN “ S PRECEDES P BETWEEN Q AND R ”	27
FIGURE 4.1	PROPOLS ONTOLOGY [36]	36
FIGURE 4.2	COMPOSITE PATTERN [36]	37
FIGURE 4.3	PROPOLS COMPOSITE PATTERN STRUCTURE [36]	38
FIGURE 4.4	THE EXAMPLE BUSINESS RULES REPRESENTED AS PROPERTIES	39
FIGURE 4.5	EXAMPLE BUSINESS RULES REPRESENTED IN PROPOLS	42
FIGURE 4.6	DFSA SEMANTICS OF RULE 1	43
FIGURE 4.7	DFSA SEMANTICS OF RULE 2	43
FIGURE 4.8	DFSA SEMANTICS OF RULE 3	44
FIGURE 5.1	THE VERIFICATION FRAMEWORK [35]	46
FIGURE 5.2	THE TDFSA (LEFT) AND ITS COMPLEMENT TDFSA (RIGHT) OF RULE 1	49
FIGURE 5.3	THE TDFSA (LEFT) AND ITS COMPLEMENT TDFSA (RIGHT) OF RULE 2	49
FIGURE 5.4	THE TDFSA (LEFT) AND ITS COMPLEMENT TDFSA (RIGHT) OF RULE 3	50
FIGURE 5.5	THE STRUCTURE OF THE EXAMPLE BPEL SCHEMA	51
FIGURE 5.6	THE LTS REPRESENTATION OF THE EXAMPLE BPEL SCHEMA	52
FIGURE 5.7	EXAMPLE SEMANTIC ANNOTATIONS TO WEB SERVICE OPERATIONS IN WSDL[36]	53
FIGURE 5.8	THE DFSA OF THE EXAMPLE BPEL SCHEMA	55
FIGURE 5.9	THE TDFSA OF THE EXAMPLE BPEL SCHEMA	55
FIGURE 5.10	THE INTERSECTION OF THE COMPLEMENT OF RULE 3 TDFSA AND BPEL TDFSA	59
FIGURE 5.11	VERIFICATION RESULT OF THE BPEL SCHEMA AGAINST RULE 1 – CONFORMING..	61
FIGURE 5.12	VERIFICATION RESULT OF THE BPEL SCHEMA AGAINST RULE 2 - CONFORMING ..	62
FIGURE 5.13	VERIFICATION RESULT OF THE BPEL SCHEMA AGAINST RULE 3 – NOT CONFORMING	63
FIGURE 6.1	THE GUIDANCE FRAMEWORK [22]	66
FIGURE 6.2	THE BPEL DFSA WITH ERROR PATH HIGHLIGHTED IN DOTTED	69
FIGURE 6.3	COMPOSITION PROGRESS WITH GUIDANCE AID [22]	78
FIGURE 6.4	FORWARD EXTENSION GUIDANCE	79

FIGURE 6.5 BACKWARD INSERTION GUIDANCE..... 80
FIGURE 7.1 DESIGN OF BPELCHECK4GUIDE 83
FIGURE 7.2 STRUCTURE OF THE“FSA” PACKAGE..... 84
FIGURE 7.3 STRUCTURE OF THE “CHECKER” PACKAGE 85
FIGURE 7.4 STRUCTURE OF THE “GUIDER” PACKAGE 86
FIGURE 7.5 STRUCTURE OF THE “GUI” PACKAGE 87

1 Introduction

Service Oriented Architecture (SOA) is becoming the prominent paradigm for building cross-organizational distributed software applications. SOA helps reduce integration expenses, increase asset reuse, and increase business agility thus making business response quicker and more cost-effective to the changing market conditions. Central to SOA is the Web services technology. Each web service is a software system that is discoverable and assessable remotely via standard network protocols like HTTP. Within the context of SOA, service composition refers to the process of forming applications, processes, or more complex services from other less complex services. Typically, the service composition is specified using a composition language such as BPEL [3] or BPML [6]. Among them, BPEL is more popular and widely-used in the industry.

1.1 The research questions

In the process of developing service compositions, it is essential to ensure that the composite services be developed bearing the desired behavioural properties. Model-checking is a classical approach for software verification in which the properties of the software are modeled using formalisms like temporal logics, finite state automata, etc. A body of work, such as [16] and [31], has been proposed to apply model-checking techniques in specifying and verifying BPEL service compositions. Those methods, even though rigorous, are too formal and require the user to have an expertise in formal languages. The need for a more user-friendly method for service composition properties specification and verification that is suitable for practitioners like business domain experts, business analysts and software testers becomes a major research problem [35][22]. In addition, as post development verification and correction can be expensive, a systematic approach is needed that can proactively guide the service composer toward developing a service composition that conforms to the set properties on the fly.

The research questions

From these research problems, the following research questions are to be addressed in this thesis:

- (1) *How to provide a user-friendly approach that enables the static verification of service composition?*
- (2) *How to assist/guide the service composer, during composition, to develop a composite service that conforms to predefined properties?*

1.2 The approach taken

Our approach to solving the above mentioned problems in the context of BPEL service composition is illustrated in Figure 1.1 below. Generally, we employ model-checking with pattern based properties specification to solve the first problem. We then utilize the verification result to provide guidance in the form of providing *remedy plans*, which list the steps to be taken to remedy the errors uncovered in the verification.

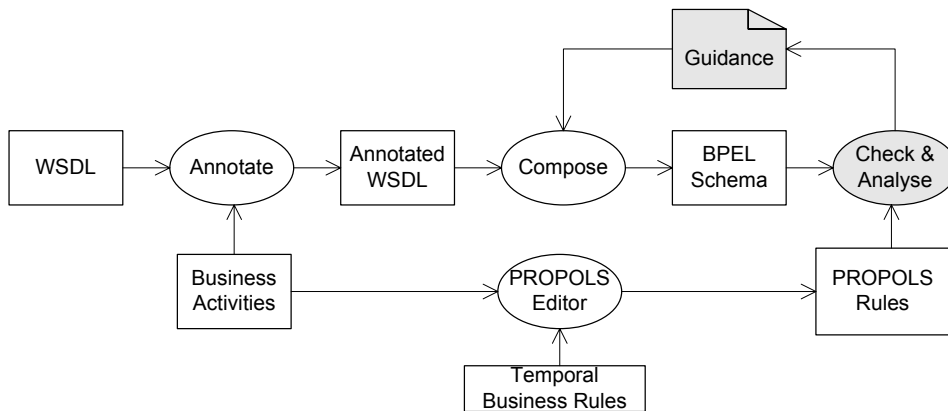


Figure 1.1 The general approach to verifying and guiding service composition [22]

From a business point of view, a service composition is a business process which orchestrates some simple business activities to achieve a complex goal; we thus firstly start with a set of business activities. The occurrences and sequencing of these business activities are constrained by a set of temporal business rules, which we will take into consideration. Generally, the rule writers can be system analysts, business analysts, or

domain experts. Since they usually do not have expertise in formal temporal logics, we adopt the pattern-based language Property Specification Pattern Ontology Language for Service Composition (PROPOLS) [35] for specifying the rules. PROPOLS is based on and extends the Property Patterns System (PPS) proposed by Dywer et al [10], which defines a set of commonly occurring high-level properties specification abstractions for some common formalisms to enable people who are not experts in those formalisms to read and write formal specifications. PROPOLS will be discussed in details in chapter 4 and PPS will be discussed in section 2.3.

As our analysis is done at the business activity level we annotate the Web service operations to be composed with business activities. The annotation utilizes WSDL-S [34] as detailed in section 5.4. The BPEL designer can then make use of the annotated WSDL-S files as the basis for composition.

In the next steps, the business rules defined in PROPOLS and the BPEL schema are both converted into FSA representation. The verification is conducted by model-checking the BPEL FSA against the rule FSA. If there is a violation then the violation is analysed and the remedy plans are generated. After that, violation information and remedy plans are communicated back to the service designer to allow real-time error correction and thus to assist the service composition designer in designing a composite service that conforms to the set temporal business rules.

1.3 Research accomplishments

The work described in this thesis is part of a research project in services composition. The project involves a professor, two postdoctoral researchers and the author of the thesis as an undergraduate Honours student. The research project has achieved four main accomplishments:

- 1) The development of a verification framework to verify a composite service (in the form of a BPEL schema) against a set of temporal business rules,
- 2) The proposal of PROPOLS, an ontology-based property specification language based on PPS to specify service composition properties,

- 3) The development of a method to assist/guide service designers to build correct composite services during the service composition process, and
- 4) The development of BPELCheck4Guide, a prototype to demonstrate the feasibility of the approach

As a member of the group, the author of the thesis has contributed actively to these four accomplishments. Specifically, the author has contributed to the shaping, refining and testing (via implementation) of the research ideas and frameworks for BPEL service composition verification and guidance, and to the semantics for logical composition support of PROPOLS. The author has also designed the verification and guidance generation algorithms. In addition, the author has co-authored two research papers [35][22], submitted to international conferences, which report the results of this research project. Finally, the author has architected and implemented the BPELCheck4Guide prototype tool.

1.4 Limitations and delimitations

The research work in this thesis is equivalent to one semester, twelve weeks, full load in the Faculty of ICT, Swinburne University of Technology. Due to this time constraint, the work of the thesis has the following limitations:

- It only considers composite services as represented by BPEL schemas. However, it is believed that similar methods can be applied to any service composition languages e.g. BPML, WSCDL, etc.
- In our work, web service parameters and the correlation between web service operations in a BPEL schema are not considered.
- For the moment, guidance is generated only when a violation occurs. It would be desirable to have forward guidance - guidance when no error occurs as well.
- The prototype implementation is subject to improvement and optimization.

1.5 Thesis structure

Figure 1.2 depicts the thesis structure. Readers should read the thesis following the arrows.

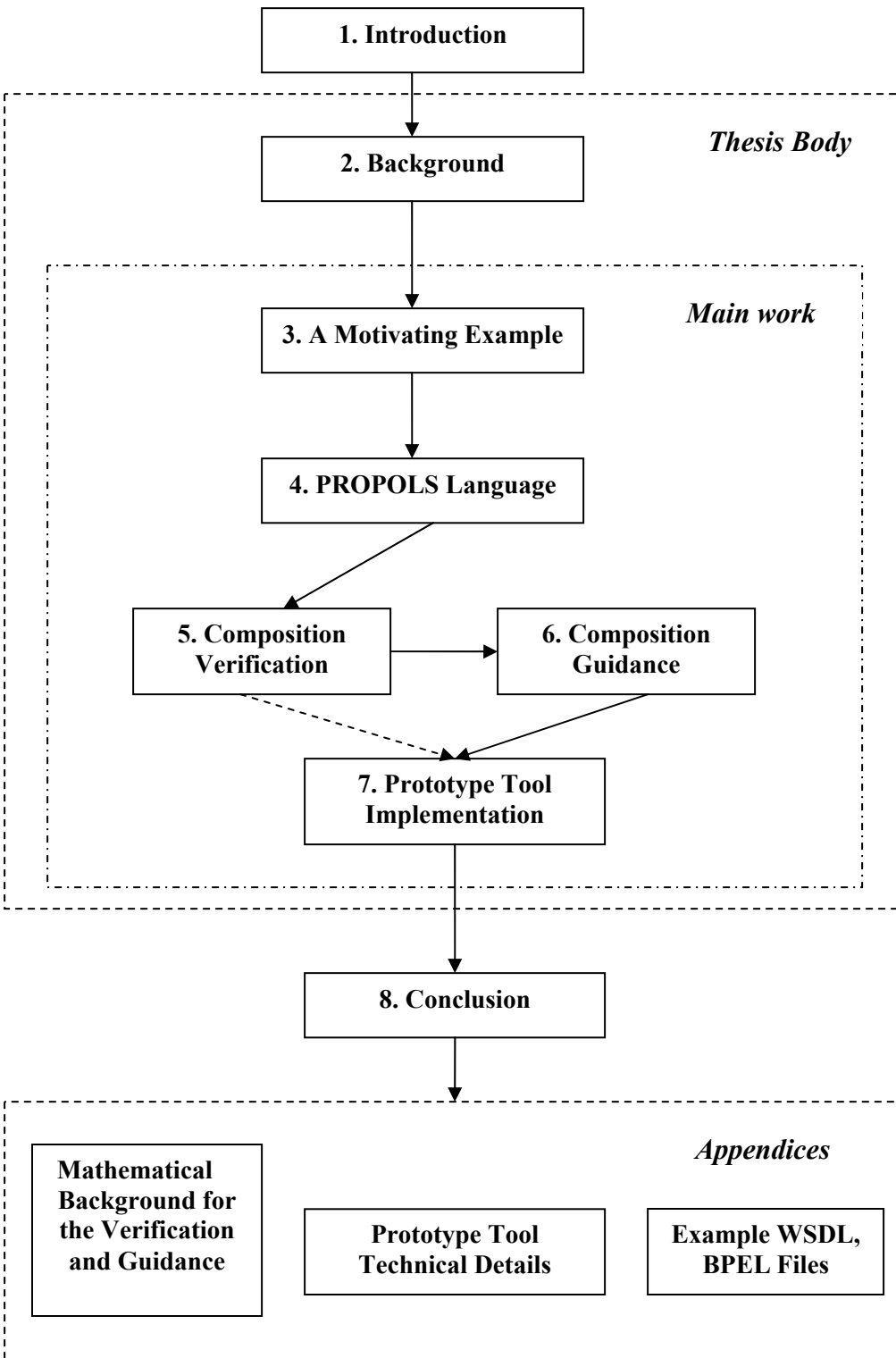


Figure 1.2 Thesis structure

Chapter 2 provides a background to the work of this thesis. It first introduces the Service Oriented Architecture and related technologies. It will then give a general introduction to model checking and its application in service composition, followed by a discussion of the Property Patterns System. The chapter ends with an overview of other work related to the research problems addressed in this thesis.

Chapter 3 presents a motivating example of the online order process of the SwinPC computer supplier company. It discusses the business scenarios and some of the issues that will be addressed by the proposed approach in this thesis.

Chapter 4 discusses PROPOLS, the language to specify temporal business rules for service compositions.

Chapter 5 describes the framework for verifying a composite service (in the form of a BPEL schema) against a set of PROPOLS properties. It presents the steps of the verification process.

Chapter 6 introduces the initial approach toward a guidance generation method to assist service composer during the composition time, utilizing the result of the checking step.

Chapter 7 describes the design and implementation of the BPELCheck4Guide tool, which is a prototype of the checking and guiding framework in chapter 5 and 6.

Chapter 8 concludes the thesis and discusses limitations of the current work and future work.

The Appendices present some mathematical background on the semantics of the service composition verification and guidance and a brief introduction to the formalisms

employed in this work. They also provide the user manual for the BPELCheck4Guide prototype and related files of the example.

1.6 Summary

In this chapter, we have described the general context of our work, the research problems, and goals. We have also outlined the approach taken in tackling the problem and the main achievements of the work. In addition, we have presented a description of the structure and organization of the thesis.

2 Background

This chapter provides the general background to our work and contains four sections. The first section will provide a brief introduction to Service Oriented Architecture, the broad context in which our work fits. We then discuss model-checking and its application in verifying services composition in section 2.2. Section 2.3 will present the PPS for specifying system properties. Finally, we look at some other related research work in section 2.4.

2.1 Service-Oriented Architecture

Service-Oriented Architecture (SOA) represents a software architectural viewpoint that defines the use of services to meet the requirements of software users [30]. In a SOA environment, resources are accessible via standardized service accessing protocols. SOA implementations are often based on, but not limited to, Web Services technologies.

One of the most important features of SOA is the loosely coupled nature of application services. Services can be implemented using different technologies, operate on different platforms but they must all use standardized interfaces (if being implemented as Web Services, they all need an XML-based WSDL description file) so that others can have access to them.

The key aspect of SOA is it promotes the reuse and interconnection of existing IT assets across different departments or systems within an enterprise. Rather than having duplication of functionalities in different systems like traditional architectures, in SOA each atomic function is encoded as a service and made available enterprise wide.

Figure 2.1 provides the big picture of Service Oriented Architecture, in which services are developed by service providers, described in XML, published in service registries, administered by service administrators, and finally discovered and consumed by service consumers.

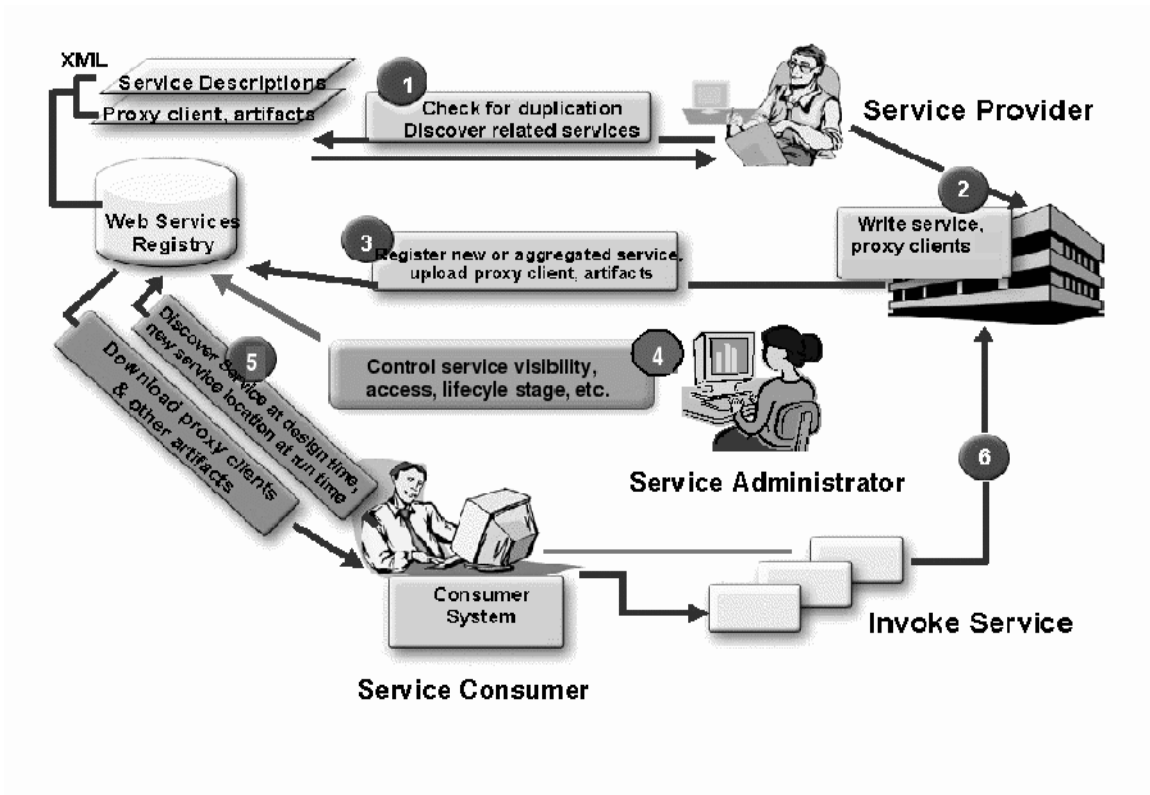


Figure 2.1 Service Oriented Architecture: the big picture [32]

2.1.1 Web services technology

A Web Service is a software system that is accessible via standard network protocols and has interfaces described in a machine understandable and processable format like WSDL [5]. To interact with a Web service, other software agents send and/or receive messages that are prescribed in its interface. The most popular network protocol used for interacting with web services is HTTP; other alternatives can be SMTP, FTP, and XMPP. Web services technology allows for cross-languages, cross-platforms interoperability thanks to its open standards. Some of the standards employed in Web Services technologies are: SOAP, Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). SOAP defines an XML messaging protocol for basic service interoperability. WSDL introduces a common grammar for describing services. UDDI provides the infrastructure required to publish and discover

services in a systematic way. Together, these specifications allow applications to find each other and interact following a loosely coupled, platform independent model [5].

2.1.2 Semantic web services

Due to the heterogeneous nature of services in a SOA environment, interoperability between services become one of the most significant concerns. Much research effort has been made on forming a conceptual framework for the “semantic web”, which is aimed at defining machine-readable descriptions of resources to facilitate automated information gathering by computers. The semantic web community has defined and standardized some of the semantic descriptive languages like RDF, OWL, and DAML for network resources description. The OWL-S [9], which is a refined version of DAML-S in OWL format for use in SOA, provides a core set of markup language constructs for describing the properties and capabilities of Web services in unambiguous, computer-interpretable form.

2.1.3 Services composition

Very often, a single service offered does not meet the requirements of service consumers, which leads to the need for service composition. Thanks to service composition, applications, processes, and more complex services can be formed from other less complex services. There exist two approaches to service composition, one favored by the business world and one by the research community. The industry has developed a number of XML-based standards to formalize the specification of Web services composition and execution. This approach is primarily syntactical and is based on the concept of workflows. With this approach, services with matching input, output and function names can be put together. This approach follows a simpler goal in terms of automating the composition process and has achieved results [7]. There have been a number of service composition languages developed and applied widely like BPEL (4WS), BPML and WSCDL. Developers are also provided with powerful platforms like Microsoft BizTalk Server, BEA Web Logic, IBM Web Sphere and Active BPEL to compose and execute the services. However, the composition process is manual and requires the developers to have an understanding about the technical details of the services being offered. The

semantic Web community, on the other hand, focuses on reasoning about web resources by explicitly declaring their preconditions and effects with terms precisely defined in ontology. For the composition of Web services, they draw on the goal-oriented inference from planning. The results, however, are not as significant [7].

2.1.4 Service orchestration and service choreography

In the context of Service Oriented Architecture, the terms service orchestration and choreography are both used to describe the composition of web services in a process flow. There are, however, fundamental differences between the two concepts.

Service Orchestration describes how web services can interact with each other at the message level, including the business logic and execution order of the interactions. These interactions may span applications and/or organizations, and result in a long-lived, transactional, multi-step process model [27].

Service Choreography tracks the sequence of messages that may involve multiple parties and multiple sources, including customers, suppliers, and partners. Choreography is typically associated with the public message exchanges that occur between multiple web services, rather than focusing on a specific business process that is executed by a single party [27].

Another important distinction between web services orchestration and choreography is orchestration refers to an executable business process that may interact with both internal and external web services. For orchestration, the process is always controlled from the perspective of one of the business parties. Choreography is more collaborative in nature; it describes how each party involved in the process interacts with others [27].

2.1.5 The BPEL language

The BPEL language (sometimes referred to as BPEL4WS or WS-BPEL) [3] is an XML-based web services orchestration language with a focus on business process description. BPEL superseded IBM's Web Service Flow Language - WSFL and Microsoft's Web

Services for Business Process Design language XLANG. It defines the model and grammar for coordinating the interaction between different participant's web services in a business process. BPEL employs web services as its communication mechanism. Therefore, BPEL's messaging facilities rely on the use of low-level web services interfaces, such as those defined in WSDL, for incoming and outgoing messages. BPEL can serve either as an implementation language for executable processes or as a description language for non executable business processes. A BPEL process is a composite web service by itself thus BPEL is considered a service composition language.

The BPEL language is central to the work in this thesis. Therefore, the following paragraphs will provide more details about the language.

A BPEL process consists of a number of activities, partners with corresponding correlation sets. The process can also include a fault handler and a compensation handler in case of faults.

BPEL defines the following atomic activities

- Invoke: invoking a specific Web service
- Receive: the server waits to receive a Web service invocation from a client.
Normally a BPEL process begins with a receive activity, i.e. a request from client
- Reply: the server responds to an invocation request. Typically a Reply activity is paired with a previous Receive activity
- Wait: Wait for an amount of time or until some deadline
- Assign: Copy a value to a variable, typically from a received message
- Throw: Throw an exception
- Terminate: Terminate the process instance
- Empty: Do nothing. Typically the empty activity is used to synchronize concurrent activities

The first three activities (Invoke, Receive, and Reply) are considered web-service-interaction activities because they directly indicate communication with participant web services.

Additional constructs are defined in BPEL to model services partners and parties involved in the business process.

- Partners: Partners are web services to be invoked in the process.
- Variables: The data containers used by the process, defined in reference to WSDL messages types. Variables are used to store state data and process history.
- FaultHandlers: The routines to handle exceptions
- CompensationHandlers: The routines to perform the compensation when a transaction rolls back
- EventHandlers: The routines to handle asynchronous external events
- CorrelationsSet: The precedence and correlations among Web services invocation

To specify the flow of a process, BPEL defines control structures for combining primitive activities into more complex algorithms.

- Sequence: For sequential execution of activities
- While: To implement a loop based on a condition
- Switch: For multi-ways branching based on the examination of a variable value
- Pick: For selecting among alternative paths based on an external event
- Flow: For parallel execution. The order of execution of activities in a flow is mandated by the services links between the activities

With the above constructs, BPEL can model complex business processes and is a powerful service composition language [30].

2.2 Model-checking service composition

This section will provide a brief introduction into the area of model-checking and some of its applications in verifying services composition.

2.2.1 Model-checking

Model-checking [8] is a formal approach to algorithmically verify the dynamic behaviours of a formal software system (the model) against a set of desired properties (the specification). The model is usually expressed as a transition system like LTS. The properties are often stated in temporal logics.

2.2.2 Application in service composition

There has been a number of works applying a model-checking methodology in verifying service composition with the main differences between them being the specification language in use. Foster [15] models BPEL process in the process algebra language Finite State Process (FSP) with the underlying Label Transitioning System semantics and utilizes the model checker LTSA for verifying common properties like live lock, dead lock and other properties stated in FSP. Fu [16] attempted to model check BPEL process utilizing the well-known SPIN model checker. In his work, a BPEL process is modeled in SPIN's input language Promela and properties are specified using Buchi Automata. In [31], Stahl maps BPEL into Petri net and, with the help of verification tool LORA (Low Level Petri net Analyzer), verifies CTL properties.

2.3 Property patterns

Due to its formal nature, conventional model-checking approaches require the properties writers to be experts in formalisms like temporal logics. To assist practitioners in reading and writing formal property specifications, Dywer et al proposed the Property Pattern System (PPS) [10] as “high-level specification abstractions” to those formalisms so that practitioners can specify the properties without the need for a background in formal methods. According to the data from [11], 92% of 555 specifications from different sources matched one of the patterns. We will discuss the PPS in details in this section.

Figure 2.2 lists all PPS constructs, those on the left hand side are properties patterns while those on the right hand side are scopes. The scopes define the time intervals during which the patterns take effect. The combination of a patterns and scopes forms properties that may be used to govern the behaviours of software systems.

Scope “globally” refers to the whole period of an application run. Scope “before S ” refers to the portion before the first occurrence of S . “after R ” refers to the portion after the first occurrence of R . “between R and S ” refers to those portions whose beginning is marked by R and whose end is marked by S . “after R until S ” is similar except that the last portion can be open-ended.

For a given scope, pattern “ P precedes Q ” requires that Q always be preceded by P within the scope. Pattern “ P leads to Q ” requires that P always be followed by Q within the given scope. “ P is absent” states that P never occurs within the scope. “ P is universal” means that only P can occur within the scope. Finally, “ P exists” requires that P must occur within the scope. It can be quantified with a minimum, maximum, or exact number of times on P ’s occurrences.

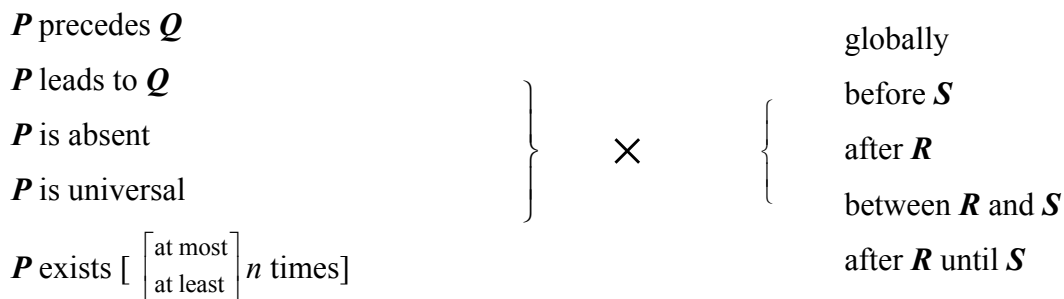


Figure 2.2 PPS patterns [22]

2.3.1 FSA semantics of property patterns

The semantics of pattern properties can be given in Linear Temporal Logic, Computation Tree Logic, Quantified Regular Expression, or Finite State Automata (FSA). Figures 2.3 and 2.4 show the FSA semantics of two example pattern properties as in [21]. In these

figures, a double circle denotes an accepting state of the FSA and a single circle denotes a non-accepting state.

Figure 2.3 is the FSA semantics of the property “ P exists after Q ”. In this figure, the symbol O denotes any other events except for P and Q . This property mandates the occurrence of event P after the occurrence of event Q . Specifically, the occurrence of Q will move the FSA out of the accepting initial state S_0 to the non-accepting state S_1 , and only an occurrence of event P at state S_1 can bring the FSA back to the accepting state S_2 .

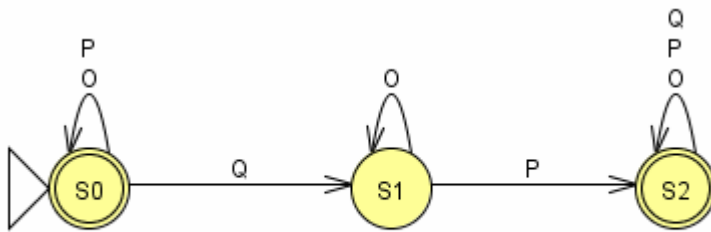


Figure 2.3 Semantics of pattern “ P exists after Q ”

Figure 2.4 is the FSA semantics of the property pattern “ S precedes P between Q and R ”. In this figure, the symbol O denotes any other events except for P , Q , R , and S . This property regulates that between the occurrence of a pair of events Q and R , if event P is to occur then event S must occur before it. Specifically, event Q will bring the FSA out of the initial state S_0 to state S_1 . At state S_1 , if S occurs then the FSA transits to state S_3 , at which P can occur and then R can occur. However, if at S_1 P occurs without a previous S , the FSA will transit to state S_2 , at which R is blocked.

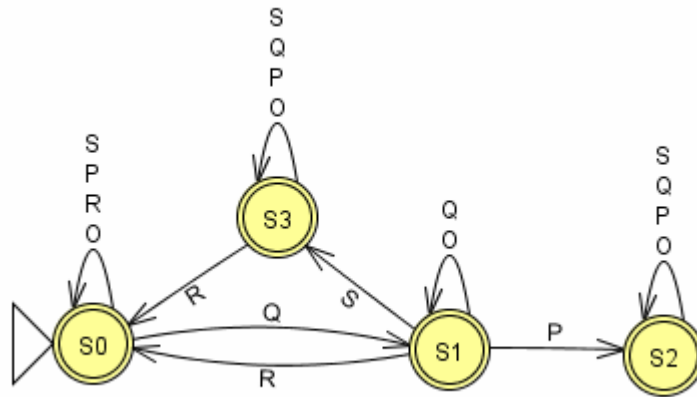


Figure 2.4 Semantics of pattern “*S* precedes *P* between *Q* and *R*”

2.3.2 Applications

PPS has been used in describing services behavioral properties like the Interaction Property Patterns discussed below. However, to our best knowledge, there is no work that applies PPS to state service composition properties.

Interaction Property Patterns

[23] utilizes PPS to specify web services interaction protocols called Interaction Property Patterns (IPP). PPS patterns and scopes are used to define basic and higher-level operators to specify the dependency rules between the invocations of business services. In [23], an OWL based Interaction Property Patterns ontology system has been designed and added to the OWL-S Service Model of web services as a complement.

2.4 Other related work

The work in [7] gives an overview of the service composition area and presents a precise comparison between the two basic approaches toward service composition namely the manual approach and the automatic approach.

There has been a body of work in the area of automatic service composition. One of the most formal studies of service composition is the work in [5], which proposed an information model based on BPEL constructs and utilizing OCL to represent composition rules, for dynamic (and automatic) phased web services composition. The work in [24] presents an OWL-S based Goal Description Language for web services to catch user requirements and constraints which provide a mechanism to detect inconsistency in goals but it does not deal with the constraints between the services themselves. For instance, [4] and [25] adopt the model-driven architecture, automatically generating code skeletons in BPEL from manually built abstract process models expressed in UML activity diagrams. [28] and [33] relies on rich semantic descriptions of component Web services given in OWL-S or abstract BPEL processes, and uses model checking based planning to automatically produce executable BPEL processes. However, these works are aimed at a fully automatic service composition process, which is not mature or widely adopted at present [7].

Work also has been done in the area of manual or semi-automated end-user service composition. In [19] the authors proposed the framework and a language, the VINCA language, to allow end-users to compose business services. Within the VINCA framework, [13] provides a mechanism to support dynamic resource matching and binding utilizing QOS and [18] provides a solution to model user context to cater for personalized service space. [18] proposes a method for automated service composition assistance utilizing properties patterns. The end user composition method is promising but is also not at a mature state as generally, end users are not expected to be able to state complex control structures and logical flows so the composed web services are often simple and linear.

2.5 Summary

In this chapter, we have presented the general background to this thesis. Firstly, we presented an overview of the Service Oriented Architecture, web services and service composition. Secondly, we looked at the area of model-checking and some of its application in service composition. Thirdly, we examined the PPS, which serves as the

Chapter 2 Background

base for our work. Finally, we have reviewed some related research work in the area of service composition. In the next chapter, we will introduce a motivating example for our work.

3 A Motivating Example

This chapter presents a motivating example for our work as a basis for illustration. The example is adapted from [36]. The computer supplier company SwinPC attempts to automate its order processing operations and allows business clients to place orders online. The IT department of the company develops a service composition for automating that process. In this chapter, we firstly discuss the business scenario. We then list the business operations that the company and its business partners have exposed as Web services. The chapter ends with the discussion of some issues in the service composition context that this research project will address.

3.1 The business scenario

The typical day-to-day operations of SwinPC's Order Processing Unit (SOPU) in the marketing department are receiving and verifying orders from clients, contacting the manufacturing department for assembling the ordered package, thus fulfilling the order, and finally receiving payment from the client. SwinPC specifies the following business rules that SOPU must adhere to.

1. Any order from the customer must be checked before processing. For an invalid order, the customer will get a rejection while for a valid one, the customer will get a confirmation.
2. The Bank will transfer payment to SwinPC only after the order is fulfilled and an invoice is issued to the customer.
3. SwinPC fulfills the order only when it is confirmed that the customer has deposited money paid for the order into a third party Bank. However, if the payment has been made to SwinPC, then this rule is no longer applicable.

These business rules, hereby referred to as Rule 1, Rule 2, Rule 3, specify the requirements for the software developed to automate the order processing at SOPU.

3.2 Business activities exposed as web services operations

SOPU's operations are currently semi-automated in the sense that operations are computer assisted but the entire process still needs to be coordinated by a human staff member. SOPU wishes to fully automate the process utilizing the Service Oriented Architecture paradigm. In the first step, SOPU asked the IT department to wrap all existing business operations related to the order processing as web services. Specifically, the following business activities from SOPU need to be exposed as web services operations.

- 1 PlaceOrder business service: client will be able to invoke this business service to place orders. SwinPC has developed the web services operation *placeOrder()* for this purpose.
- 2 ConfirmPayment business service: the Bank business partner will be able to use this business service to confirm with SOPU that a deposit for the order has been made. SwinPC has developed the web services operation *confirmPayment()* for this purpose.

SwinPC's manufacturing department (hereby referred to as the Manufacturer) also need to expose the following business activities as web services operations.

- 1 CheckOrder business service: SOPU can request the that Manufacturer verifies the order and returns a positive message if it is valid and a negative message otherwise. The web services operation *checkOrder()* has been developed for this purpose.
- 2 FulfilOrder business service: SOPU can request that the Manufacturer fulfills the order and makes it ready for delivery. The web services operation *fufilOrder()* has been developed for this purpose.

Finally, SwinPC requires its business clients (hereby referred to as the Customers) to expose the following operations using web services so that the client can be informed of the progress of order processing.

- 1 ConfirmOrder business service: The Customers must provide this service to be able to get order confirmation from SOPU. The web services operation *confirmOrder()* has been developed by the Customers and the WSDL files made available to SwinPC for this purpose.

- 2 RejectOrder business service: The Customers must provide this service to be able to get order rejection from SOPS. The web services operation *rejectOrder()* has been developed by the Customers and the WSDL files made available to SwinPC for this purpose.

3.3 Building the business process

A service designer decides to use BPEL to integrate the above available web services into an order processing composite service. This service will be made available from SwinPC to be invoked by its business clients whenever they want to order products from SwinPC. The service designer uses the BPEL GUI Editor “Active BPEL” [1], which allows him to drag and drop business activities and graphically define the structure of the business process thus saving him a lot of time. However, there are two things that are still of concern to the service designer.

1. How can he make sure the designated BPEL schema satisfies the business requirements stated in section 3.1 without having to deploy and validate the BPEL schema at run time?
2. As error correction is troublesome, is there any method to assist him to develop a correct service composition on the fly?

Our work addresses these two concerns. It involves service composition property specification, conformance checking, and guidance based on property specifications. These issues are discussed in the next three chapters.

3.4 Summary

In this chapter, we have presented a motivating example, the SwinPC Company wishing to automate its online ordering process. We have introduced the business scenarios, the set of business activities exposed as web services operations and the issues faced by a service designer when composing a BPEL schema for the order processing business process. The example will be used in the rest of the thesis to demonstrate the developed methods.

4 Specification of Temporal Business Rules

As current methods for verifying service composition properties require the specification to be written in a formalism like temporal logics, which requires the writer to have a strong mathematical background, there is the need for a user-friendly and intuitive property specification language for service composition for use by practitioners. We designed the Property Specification Pattern Ontology Language for Service Composition (PROPOLS) [36] to address this need. This chapter starts by giving an overview of the language design and constructs, followed by a discussion about the support of logical composition of property patterns in the language. It then shows how the business requirements in section 3.1 are refined and restated as a set of properties using the Property Patterns System. Finally, it presents the PROPOLS representation of those properties.

4.1 About PROPOLS

PROPOLS is a language for specifying temporal business rules in our work. PROPOLS is based on and extends PPS so that practitioners just need to write high level, abstract specifications and do not need to deal with low-level formalisms such as LTL, QRE, FSA, etc. In PROPOLS, the temporal rules are stated at the business activity level (in contrast to the Web service level) which encourages direct involvement of business experts and helps reduce chances of misunderstanding of business requirements by the BPEL designer.

PROPOLS's elements and semantics is based largely on PPS with the extension of logical composition support, which will be discussed in the next section. A PROPOLS file defines a collection of properties for a service composition with each property being a rule or a logical composition of rules governing the existence or ordering of the elementary services within a service composition. Each rule consists of a pattern element and a scope element. The pattern element specifies the existence behaviours (absent, exist, universal) of a single business activity or the temporal relationship (leads to,

precedes) between two activities. The scope defines an interval of time during which the pattern takes effect. Our PROPOLS language follows a similar approach as the IPP in [23] but differs from IPP in two main features 1) PROPOLS extends IPP with the support for logical composition between patterns 2) PROPOLS is used for specifying the internal properties of a service composition while IPP specifies interaction properties of a single web service.

PROPOLS is designed as an ontology language which makes it easier for machine-processing. Making PROPOLS an ontology language also has the following advantages. First, ontology can be used to define standard terminology for the pattern system. Second, practitioners like business experts can use concepts from already-existing domain ontology to define pattern properties at a high level of abstraction. After being defined, the pattern properties themselves also become part of the shared formal domain knowledge, so they can be reused in a wide scope. At present, OWL is the most widely used Web ontology language so we choose OWL as the base language of PROPOLS.

4.2 The language design

Each element of PROPOLS is defined as a class in OWL with the relationships between these classes forming the language structure. Some essential classes are discussed below.

OrderPattern class: This class defines the set of order patterns as discussed in section 2.3. We use the name *LeadsTo* to replace the original *Response*. Here, *P LeadsTo Q* has the same semantics as *Q RespondsTo P*.

OccurrencePattern class: This class defines the set of occurrence patterns as discussed in section 2.3. Note that *BoundedExists* is a subclass of *Exists*, it has additional *equalBound*, *maxBound* and *minBound* properties compared with *Exists*.

Scope class: Every elemental pattern has a scope coming from *globally*, *before*, *after*, *between* or *until*.

Operation class: This is the class for service operations. An operation is considered an event in the pattern properties. It has a *provider* property linking to its provider and a

conceptReference property linking to the corresponding concept in some ontology. As we will see in section 5.4, the *conceptReference* can be used to map an event in pattern properties to a Web service operation semantically.

Expression, Unary and Binary classes: *Expression* is a general class for patterns and operations. For the succinctness of the PROPOLS ontology, all the patterns and scopes are subclasses of *Unary* or *Binary*. Therefore, they share the properties defined in *Unary* or *Binary*, either having an *operand* property pointing to an *Expression* or having a *firstOperand* property and a *secondOperand* property. For example, *Exists* is a unary pattern and *Between* is a binary scope. Further restrictions are set for different kinds of expressions. For example, any of the *Scopes* only accept operations as its operands.

ConstraintList class: As shown in Figure 4.1, this class is a container for pattern properties/constraints.

Figure 4.1 shows a graphical overview of the PROPOLS ontology generated using Ontoviz [26], an ontology visualization plug-in for an OWL editor tool Protégé.

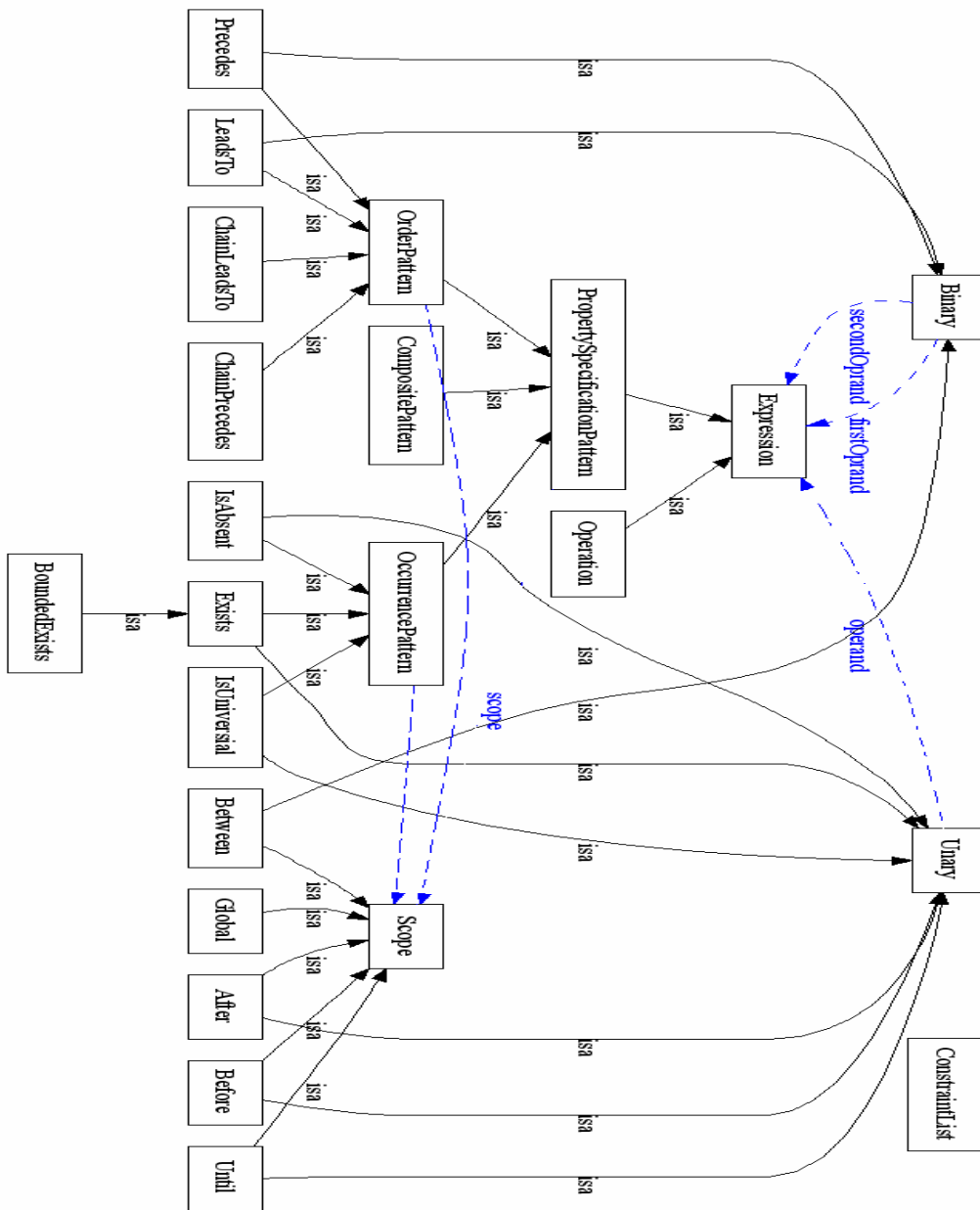


Figure 4.1 PROPOLS Ontology [36]

4.3 Logical compositions of patterns

Adding the support for logical composition of rules helps increase the expressiveness of PPS. With composite patterns, complex properties can be stated in a straightforward manner. Some requirements like the first rule of the motivating example in section 3.1 can not be expressed without logical composition of patterns. That requirement claims an ‘exclusive or’ relation between *order confirmation* and *rejection*. It also demands the precedent occurrence of the *check order* activity. We may specify this requirement as a composite pattern property as in Figure 4.2

```
(Customer.ConfirmOrder exists globally
  Xor Customer.RejectOrder exists globally)
  And Manufacturer.checkOrder precedes
    Customer.ConfirmOrder globally
  And Manufacturer.checkOrder precedes
    Customer.RejectOrder globally
```

Figure 4.2 Composite Pattern [36]

Figure 4.3 shows the sub-structure of the *CompositePattern* class. The composition relationship can be either *And*, *Not*, *Imply*, *Or*, or *Xor*. In addition, the logical composition can be nested to give more complex and expressive rules. The semantics for these logical compositions is based on FSA composition semantics given in Appendix 1.

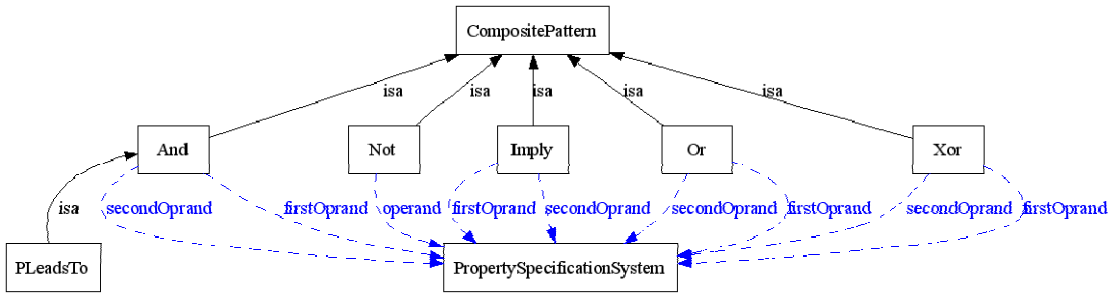


Figure 4.3 PROPOLS Composite Pattern Structure [36]

4.4 The example rules

In this section, we will show how business rules can be encoded in PROPOLS and how the underlying semantics for the rules is represented.

The three business rules in section 3.1 are refined into the following pattern properties.

Rule 1

```
(Customer.ConfirmOrder exists globally
Xor Customer.RejectOrder exists globally)
And Manufacturer.checkOrder precedes
    Customer.ConfirmOrder globally
And Manufacturer.checkOrder precedes
    Customer.RejectOrder globally
```

Rule 2

```
Customer.IssueInvoice leads to Bank.ConfirmPayment globally
```

Rule 3

```
Bank.ConfirmDeposit precedes Manufacturer.FulFilOrder before
    Bank.ConfirmPayment
```

Figure 4.4 The example business rules represented as properties

These properties above are then encoded into PROPOLS. Figure 4.5 shows the PROPOLS code segments of the three rules.

Rule 1

```
<and rdf:ID="And_Property">
  <secondOprand rdf:resource="#Pre_Rejt"/>
  <firstOprand rdf:resource="#And_inst_1"/>
</and>
<precedes rdf:ID="Pre_Rejt">
  <scope><global rdf:ID="Global_Inst"/></scope>
  <firstOprand>
    <operation rdf:ID="CheckOrder">
      <callee rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
      >#Manufacturer</callee>
    <conceptReference
```

```

rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >#CheckOrder</conceptReference>
</operation>
</firstOprand>
<secondOprand rdf:resource="#getRejectNotifcation"/>
</precedes>

<and rdf:ID="And_inst_1">
  <firstOprand>
    <xor rdf:ID="Xor_Inst">
      <secondOprand>
        <exists rdf:ID="Reject_Exists">
          <scope rdf:resource="#Global_Inst"/>
          <operand rdf:resource="#getRejectNotifcation"/>
        </exists>
      </secondOprand>
      <firstOprand>
        <exists rdf:ID="Confirm_Exists">
          <scope rdf:resource="#Global_Inst"/>
          <operand rdf:resource="#getConfirmNotification"/>
        </exists>
      </firstOprand>
    </xor>
  </firstOprand>
  <secondOprand>
    <precedes rdf:ID="Pre_Conf">
      <secondOprand rdf:resource="#getConfirmNotification"/>
      <scope rdf:resource="#Global_Inst"/>
      <firstOprand rdf:resource="#checkOrder"/>
    </precedes>
  </secondOprand>
</and>

<operation rdf:ID="getConfirmNotification">
  <conceptReference
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >#ConfirmNotification</conceptReference>
  <callee rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

```

```

    >#Customer</callee>
</operation>
<operation rdf:ID="getRejectNotifcation">
    <conceptReference
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >#RejectNotification</conceptReference>
    <callee rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >#Customer</callee>
</operation>

```

Rule 2

```

<leads to rdf:ID="Rule_2">
    <scope><globally rdf:ID="Globally_Inst"></globally></scope>
    <firstOperand"><operation rdf:resource="IssueInvoice">
        <callee rdf:datatype="&XMLSchema;anyURI">#Customer</callee>
    </operation></firstOperand>
    <secondOperand/><operation rdf:resource="ConfirmPayment">
        <caller rdf:datatype="&XMLSchema;anyURI">#Bank</caller>
    </operation></secondOperand>
</precedes>

```

Rule 3

```

<precedes rdf:ID="Rule_3">
    <scope><before rdf:ID="Before_Inst"><operand>
    <operation rdf:resource="ConfirmPayment">
        <caller rdf:datatype="&XMLSchema;anyURI">#Bank</caller>
    </operation></operand></before></scope>
    <firstOperand"><operation rdf:resource="ConfirmDeposit">
        <caller rdf:datatype="&XMLSchema;anyURI">#Bank</caller>
    </operation></firstOperand>
    <secondOperand/><operation rdf:resource="FulfilOrder">

```

```

    <callee rdf:datatype="&XMLSchema:anyURI">#Manufacturer</callee>
  </operation></secondOperand>
</precedes>

```

Figure 4.5 Example business rules represented in PROPOLS

4.5 FSA semantics of the rules

This section shows the FSA semantics of the three properties in the previous section. Because of space limitations, for all reflexive transitions, we only show their event ids. The mapping between event ids and the events is given in Table 4.1 below.

Event id	Event	Event id	Event
0	RejectOrder	6	ConfirmPayment
1	PlaceOrder	7	IssueInvoice
2	CheckOrder	8	CheckOrderReply
3	ConfirmOrder	9	orderResultFail
4	ConfirmDeposit	10	orderResultOk
5	FulfilOrder		

Table 4.1 Mapping of events and event ids

The FSA semantics of Rule 1 above is presented in Figure 4.6. The accepting states of the FSA are state 6 and state 8 as highlighted in double circles. This means that this property will only accept sequences of events that bring the FSA to either state 6 or state 8. As a result, event sequences *<CheckOrder, RejectOrder>* or *<CheckOrder, ConfirmOrder>* are both accepted by the rule while that of *<CheckOrder, RejectOrder, ConfirmOrder>* or *<CheckOrder, ConfirmOrder, RejectOrder>* both lead the FSA to the non-accepting state 7 and are, thus, invalid.

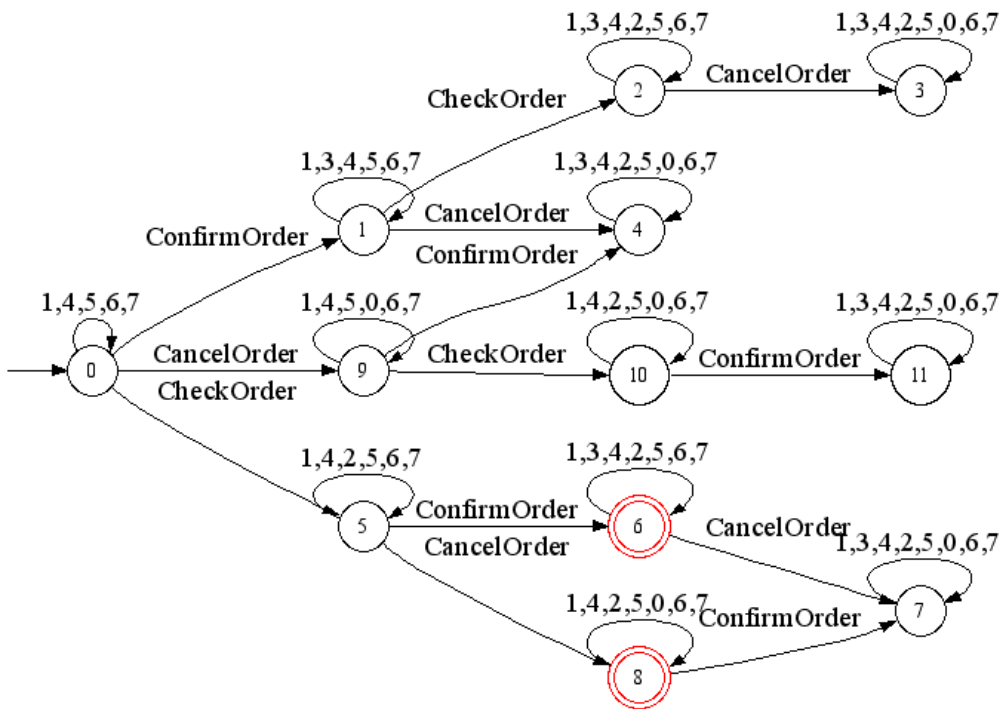


Figure 4.6 DFSA semantics of Rule 1

The FSA semantics of rule 2 above is given in Figure 4.7 below. It can be seen that a sequence of events containing *IssueInvoice* will lead to the non-accepting state 1, and only an occurrence of the event *ConfirmPayment* after that can bring the FSA back to the accepting state 0.

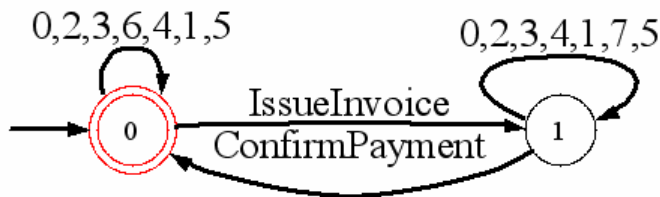


Figure 4.7 DFSA semantics of Rule 2

Figure 4.8 presents the semantics of Rule 3. It can be seen that at the accepting state 0, any event can occur. However, if *FulfilOrder* occurs first without a prior occurrence of *ConfirmDeposit* or *ConfirmPayment* then after that *ConfirmPayment* is blocked.

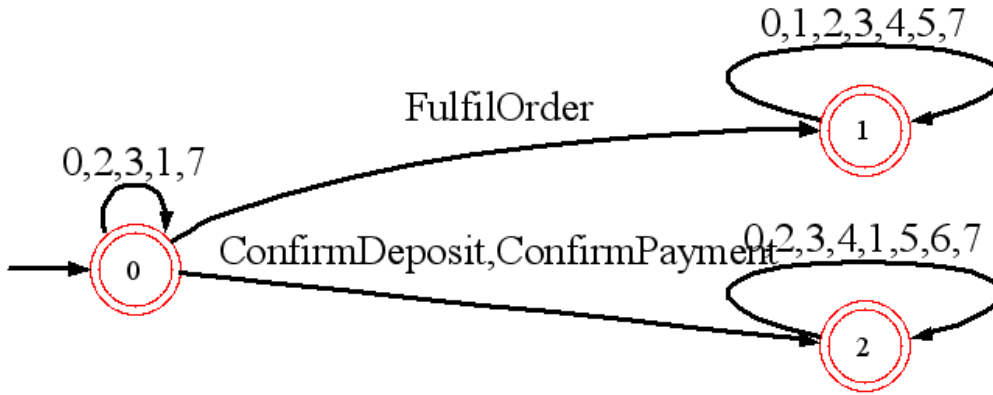


Figure 4.8 DFSA semantics of Rule 3

4.6 Summary

In this chapter, we have introduced the PROPOLS language for specifying service composition properties. We have discussed the language’s main design and its main elements. We have also illustrated the use of the language with examples from Chapter 3. In the next chapter, we will show how a service composition can be verified against a set of rules defined in PROPOLS.

5 Verification of BPEL Service Composition

This chapter describes the approach to verifying a service composition in the form of a BPEL schema, against a set of temporal business rules defined in PROPOLS. We will refer to the SwinPC motivating example in chapter 3 for illustration purposes. The chapter starts by introducing the general framework for verification, followed by the detailed steps in verifying the BPEL schema.

5.1 The verification framework

We employ an approach similar to that of [16] for Buchi automata based model-checking for the BPEL verification. The most significant difference between our work and the work in [16], [31], and [14] is that we focus on a practitioner-oriented approach to property specification. The used pattern-based specification language is easier to understand and use, compared to the formal languages they use, which require deep knowledge in formal methods.

We verify the conformance of the BPEL schema to the PROPOLS rules by checking whether the accepting event sequence set of the BPEL Total Deterministic FSA (TDFSA) is included in the accepting event sequence set of the rule TDFSA. This is done by testing the emptiness of the intersection of the BPEL TDFSA and the complement of the rule TDFSA. For more details about the mathematical foundation of this reasoning, please refer to Appendix 1

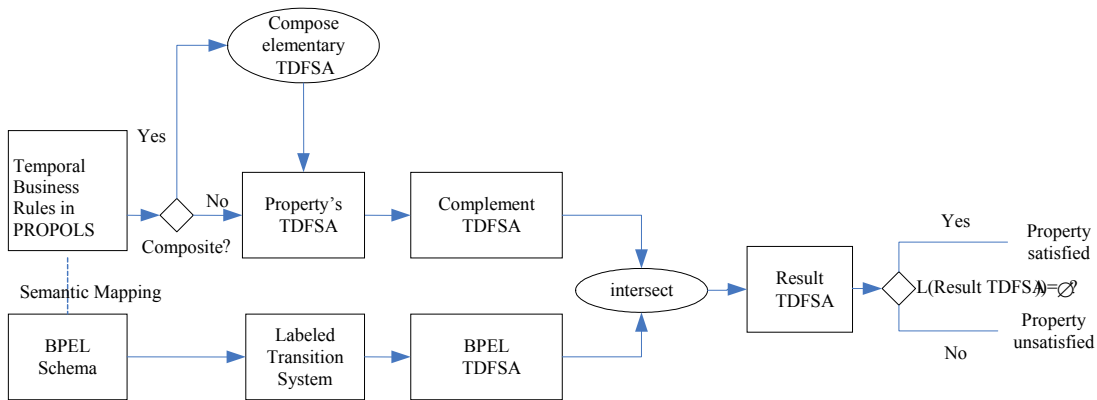


Figure 5.1 The verification framework [35]

As shown in Figure 5.1, the verification is conducted in 3 steps:

1. For every rule, a semantically equivalent total and deterministic FSA (TDFSA) is built. If the rule is composite, the corresponding TDFSA is constructed by composing the TDFSA of its elementary rules according to the composition semantics defined in Appendix 1.
2. For the BPEL schema, a finite and deterministic LTS model is generated, from which a TDFSA is built by introducing the set of accepting states and a trap state that collects all unacceptable events at each state.
3. The conformance can then be verified by intersecting the complement of rule TDFSA and the BPEL TDFSA and checking for the existence of any accepting sequence of events in the resultant FSA. If such a sequence exists, the BPEL schema does not conform to the rules. Otherwise it conforms to the rules.

The following sections explain in details the steps involved in verifying a BPEL Schema.

5.2 Convert temporal business rules to complementary TDFSA

For every temporal business rule defined in PROPOLS, a deterministic FSA instance is initialised by actualising the labels with business activities. Formally, that FSA represents the language accepted by the business rule. As explained above, the principle of the verification is to check for the non-emptiness of the intersection between the complementary of the rule TDFSA and the BPEL TDFSA; therefore, the rule FSA will first need to be converted into the complementary of its corresponding total FSA. This section will explain the procedure of converting the rule FSA into the complementary TDFSA.

To convert from a non-Total DFSA to a Total DFSA, we introduce an extra trap state and create a set of transitions targeting that trap state for every rejected event at every state. The complementary TDFSA is then computed by toggling the acceptance status of every state of the TDFSA. For the mathematical foundation of this process, please refer to Appendix 1.

As we will see in section 5.6, when the two TDFSA are intersected, the pre-condition is that they must have the same alphabet. Therefore, when converting the rule and the BPEL DFSA to their Total DFSA, we make them share the union of their alphabet. We introduce the concept of “*total with respect to*” a specific alphabet. A DFSA is considered total with respect to an alphabet if every event in that alphabet is acceptable at every state of the DFSA. In our context, when no set of alphabet is mentioned, the considered alphabet is the union of the BPEL DFSA and the rule DFSA alphabets.

Algorithm 5.1 is the pseudo code for function *transformToTotalFSA()* which takes as input a FSA and a set of events as the alphabet, and transforms that FSA into a total FSA relative to the specified alphabet. The procedure starts in line 2 by adding a non-accepting trap state into the FSA. Then for every state within the states set of the FSA (including the trap state) it calculates the set of all non-accepting events at that state. If that set is not empty then it creates a transition from that state to the trap state, labeled with the calculated set of events.

```

1  FUNCTION transformToTotalFSA (fsa, alphabet)
2    fsa.addState(trapState)
3  FOR EACH state IN fsa.getStates()
4    toTrapEvtSet = alphabet - fsa.getEventsForState(state)
5    IF toTrapEvtSet IS NOT EMPTY THEN
6      fsa.addFSATransition (state,
7                            trapState,
8                            toTrapEvtSet)
9    END IF
10 END FOR
11 END

```

Algorithm 5.1 Transform DFSA to TDFSFA with regards to an alphabet

Figure 5.2 is the TDFSFA and the complement TDFSFA of the first example business rule calculated using algorithm 5.1. It can be seen from this figure that 1) the TDFSFA is accepting every event at every state; 2) accepting states in the TDFSFA becomes non-accepting states in its complement. In this figure, state 7, 9, and 11 are the added non-accepting trap states.

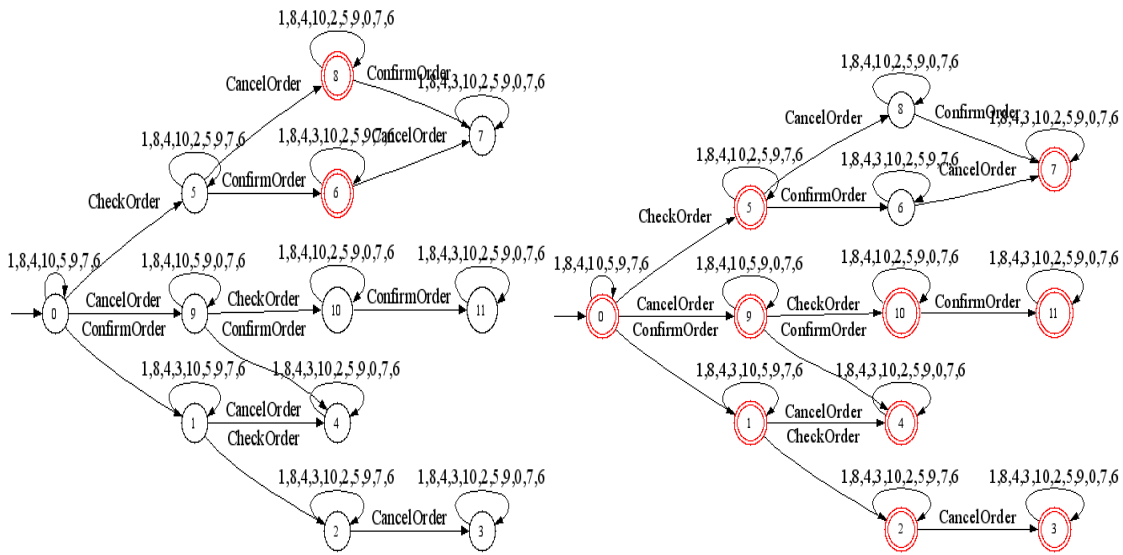


Figure 5.2 The TDFSA (left) and its complement TDFSA (right) of Rule 1

Figure 5.3 shows the TDFSA and its complement of the second example business rule in section 3.1. As the Rule 2 FSA is total itself, no trap state has been added.

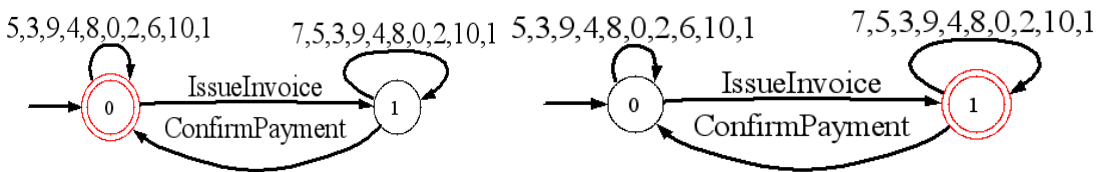


Figure 5.3 The TDFSA (left) and its complement TDFSA (right) of Rule 2

Figure 5.4 shows the TDFSA and its complement of the third example business rule in section 3.1. It is seen that the non-accepting state 3 has been introduced to collect the rejected event *ConfirmPayment* at state 1.

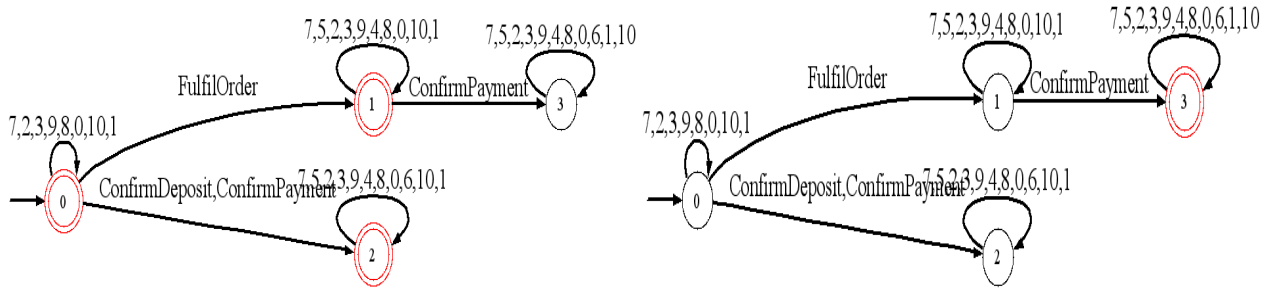


Figure 5.4 The TDFSA (left) and its complement TDFSA (right) of Rule 3

5.3 Representing BPEL in LTS

The intermediary step taken when converting a BPEL schema into its FSA representation is to represent the schema in LTS. Foster [15] created a framework for mapping a BPEL schema into Finite State Process, a process algebra language. The semantics is represented in LTS. We adopt Foster’s BPEL2LTS APIs to get the LTS representation of the BPEL schema. According to Foster, the generated LTS is deterministic. Note that Foster’s method of translation from BPEL to LTS does not consider operation parameters. It assumes the uniqueness of the tuple (*activity_type*, *provider*, *operation_name*), in which *activity_type* is either “invoke”, “receive” or “reply”; *provider* is the service provider, to differentiate between Web services activities.

The BPEL Schema of SOPU’s order processing

Figure 5.5 shows the main structure of the BPEL schema of one of the implementations of the online purchase process for SOPU. This diagram keeps all the message exchange primitives in the actual BPEL schema. It uses black boxes to represent the other participants of the process, including the Customer, Manufacturer and the Bank.

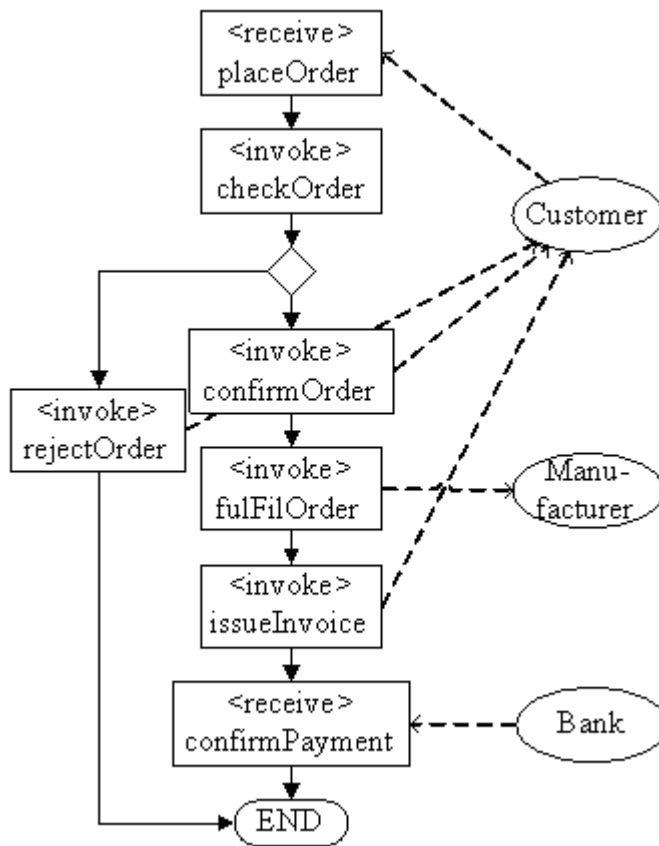


Figure 5.5 The structure of the example BPEL schema

Figure 5.6 is the corresponding LTS of this BPEL schema generated by using Foster’s tool [14]. In this figure, state 0 is the initial state of the process and state E is the end state.

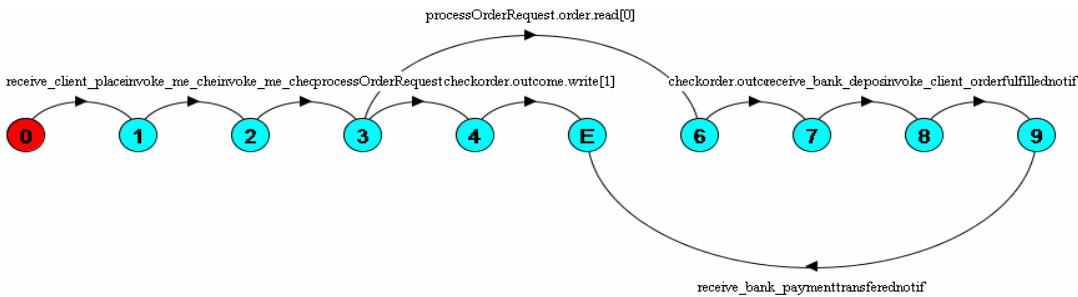


Figure 5.6 The LTS representation of the example BPEL schema

5.4 Semantic mapping

As BPEL schemas contain specific web services operations, in the generated LTS, the labels also represent web services operations. On the other hand, PROPOLS operations represent business activities and each business activity may correspond to multiple web services operations. To bridge this discrepancy, we introduce a semantic-mapping step to semantically map web services operations to a PROPOLS business activity. The mapping is based on the condition that both the PROPOLS operations (the business activities) and the web services operations refer to the same concept in the domain ontology. The PROPOLS operations can tell their semantics via the *conceptReference* property. For Web service operations, there are two typical ways in which one can add semantic annotations: by using an external annotation file or directly appending semantic annotations to the WSDL file using an extension of WSDL called WSDL-S [34]. In this context, we use the latter approach. As exemplified in Figure 5.6, we extend the Web service's operation definition for customers with the WSDL-S semantic extension element *wssem:modelReference*.

```
<portType name="Customer">
  <operation name="confirmOrder"
    wssem:modelReference="
      http://www.purl.org/onto/operationOnto#ConfirmOrder">
    ...
  <operation name="rejectOrder"
    wssem:modelReference="
      http://www.purl.org/onto/operationOnto#RejectOrder">
    ...
```

Figure 5.7 Example Semantic Annotations to Web Service Operations in WSDL[36]

In the above example, the Web services operation *confirmOrder()* and PROPOLS operation *ConfirmOrder* refer to the same ontology concept, so we use *ConfirmOrder* in place of *confirmOrder()* in the conformance checking process. Similarly, all web services operations in the BPEL LTS are replaced by the corresponding business activities as shown in table 5.1. In this table, a partner is the collaborating service from the point of view of the party for whom the service composition is built. In this case, the partners refer to the business partners involved in SOPU's online order processing business process. A provider is the party who provides the web service that contains the operation.

Business activities in PROPOLS	Web Service Operations	Partner	Provider
<i>CheckOrder</i>	<i>checkOrder()</i>	<i>Manufacturer</i>	<i>Manufacturer</i>
<i>RejectOrder</i>	<i>rejectOrder()</i>	<i>Customer</i>	<i>Customer</i>
<i>ConfirmOrder</i>	<i>confirmOrder()</i>	<i>Customer</i>	<i>Customer</i>
<i>ConfirmDeposit</i>	<i>confirmDeposit()</i>	<i>Bank</i>	<i>SOPU</i>
<i>FulfilOrder</i>	<i>fulfilOrder()</i>	<i>Manufacturer</i>	<i>Manufacturer</i>
<i>ConfirmPayment</i>	<i>confirmPayment()</i>	<i>Bank</i>	<i>SOPU</i>
<i>IssueInvoice</i>	<i>issueInvoice()</i>	<i>Customer</i>	<i>Customer</i>

Table 5.1 Matching Between PROPOLS Operations and Web Service Operations

5.5 Converting LTS to TDFSA

We then convert the BPEL LTS into a TDFSA in a straightforward process. Firstly, we mark the end states in the LTS as accepting states. After that, we introduce a trap state that collects all unacceptable events at each state.

Usually, the alphabet of the BPEL FSA contains a set of *non-web-services-interaction* events such as the assignment of values between messages. Those events are not constrained by the rules and therefore, not considered in the checking process. We treat all those *non-web-services-interaction* events as internal events of the BPEL FSA. Based on Foster's label abstraction method, we identify all Web services events, thus external events, in the BPEL FSA by considering all the LTS labels prefixed with BPEL's web services activities "*invoke*", "*reply*" and "*receive*". All other labels are then considered to be representing internal events. As we will see in section 5.6, when doing the conformance checking of the BPEL FSA against the rule FSA, all transitions caused by internal events in the BPEL FSA will be matched to a reflexive transition at every state of the rule FSA.

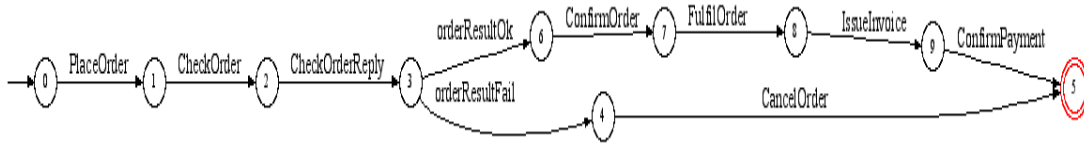


Figure 5.8 The DFSA of the example BPEL schema

Figure 5.8 is the Deterministic FSA (DFSA) of the BPEL schema and Figure 5.9 is its corresponding Total Deterministic FSA (TDFSA). The TDFSA is computed from the DFSA using the algorithm 5.1 described in section 5.2 above. Note that because of space limitations, in Figure 5.9 and some other figures in the following chapters, event ids are shown instead of the events themselves. Please refer to Table 4.1 for the mapping of those event ids and the actual events. Also note that in Figure 5.8 the two events *orderResultOk* and *orderResultFail* did not exist in the original structure of the BPEL schema. These events are automatically generated by Foster [15]’s algorithms when translating BPEL to LTS. These two are internal events of the BPEL DFSA.

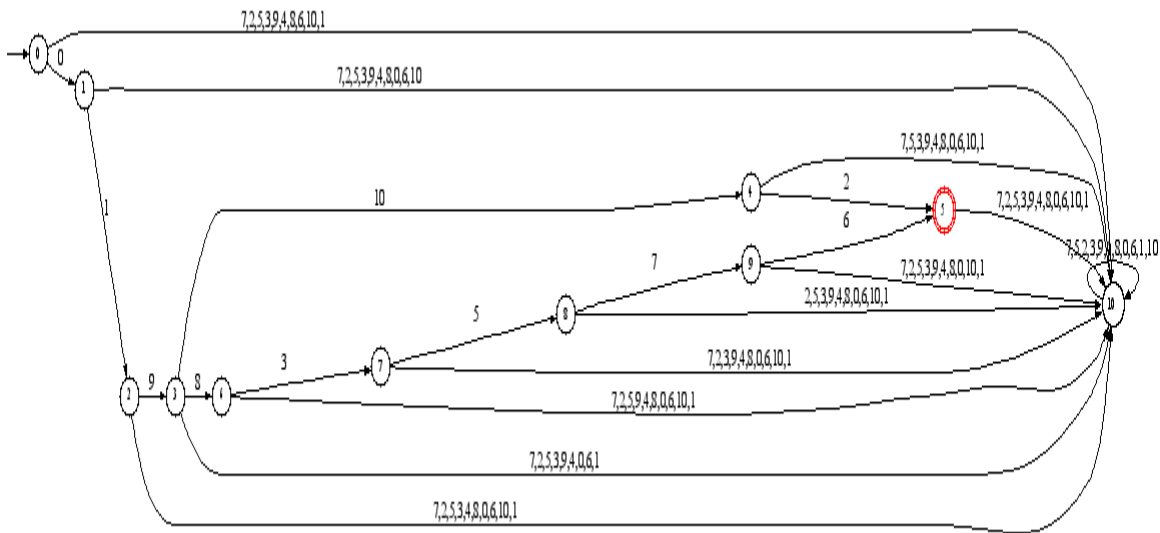


Figure 5.9 The TDFSA of the Example BPEL Schema

5.6 Intersecting complementary of rule TDFSA and BPEL TDFSA

The mathematical background of the intersection between DFSAs is given in Appendix 1, in this section we explain the algorithm to compose via intersection the BPEL TDFSA and the complement of the rule TDFSA. This algorithm can also be used for other composition methods like *union*, *exclusive*, or *imply*. It accepts as parameters the composition method (intersection, union, exclusive, imply), and the shared alphabet of the two FSA. It recursively traverses the two FSAs, combines the states according to the passed composition method, and creates the transition diagram for the resultant FSA.

```

1  FUNCTION composeTotalFSA(    method,
2                                alphabet,
3                                retFSA,
4                                fsa1,
5                                fsa2,
6                                combinedState,
7                                statel1,
8                                state2,
9                                processedSet)
10 FOR EACH e IN alphabet
11     IF processedSet.containsTripple(statel1, state2, e) THEN
12         CONTINUE
13     ELSE
14         processedSet.addTriple(statel1, state2, e)
15     END IF
16         NState1 = fsa1.getNextState(statel1, e)
17         NState2 = fsa2.getNextState(state2, e)
18         combinedNState = retFSA.findStateByName(
19             combineName(NState1.getName(), NState2.getName())
20         IF combinedNState == NULL THEN
21             combinedNState = combineState(NState1, NState2,
22                 method)
23         retFSA.addState(combinedNState)
24     END IF
25     retFSA.addTransition(combinedState, combinedNState, e)
26 composeTotalFSA(    method,
27                     alphabet,
28                     retFSA,
29                     fsa1,
30                     fsa2,
31                     combinedNState,
32                     NState1,
33                     NState2,
34                     processedSet)
35 END FOR
36 END

```

Algorithm 5.2 Composing TDFSA's

Applying algorithm 5.2 to intersect the complement of the Rule 3 TDFSA and the BPEL TDFSA results in a DFSA as shown in Figure 5.10.

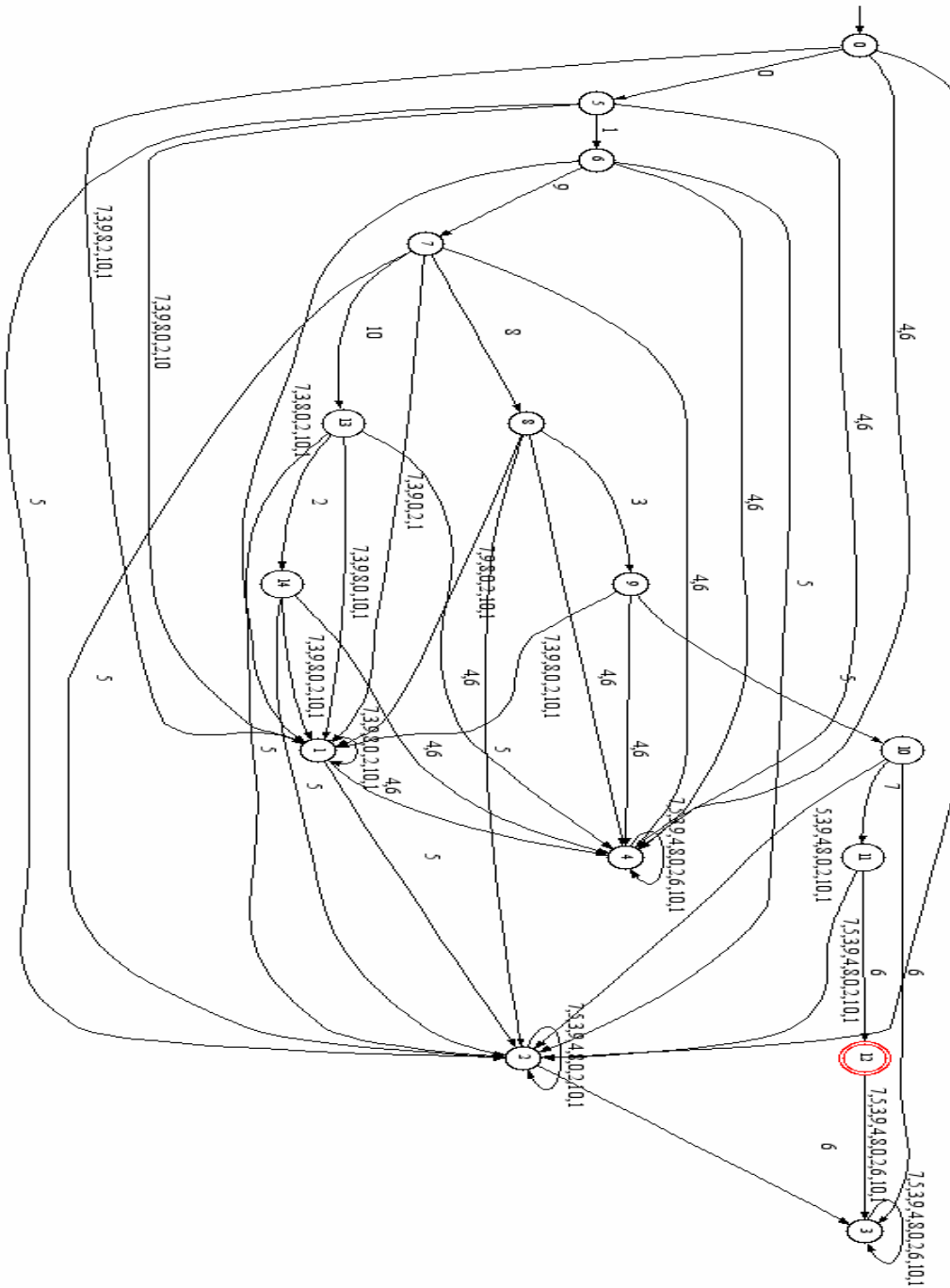


Figure 5.10 The intersection of the complement of Rule 3 TDFSA and BPEL TDFSA

5.7 Verification results of the example BPEL schema

Figures 5.11, 5.12, and 5.13 show the verification results of the designated example BPEL schema in Figure 5.5 against Rule 1, Rule 2, and Rule 3 (see section 3.1). It can be seen from these figures that the schema conforms to Rule 1 and Rule 2 however, it fails to conform to Rule 3. Specifically, Rule 3 requires the prior occurrence of *Bank.ConfirmDeposit* for *Manufacturer.FulfilOrder* to occur before the occurrence of *Bank.ConfirmPayment*. However, *Bank.ConfirmDeposit* has not been inserted into the BPEL schema which results in a violation. The BPEL schema thus has not met all the set requirements.

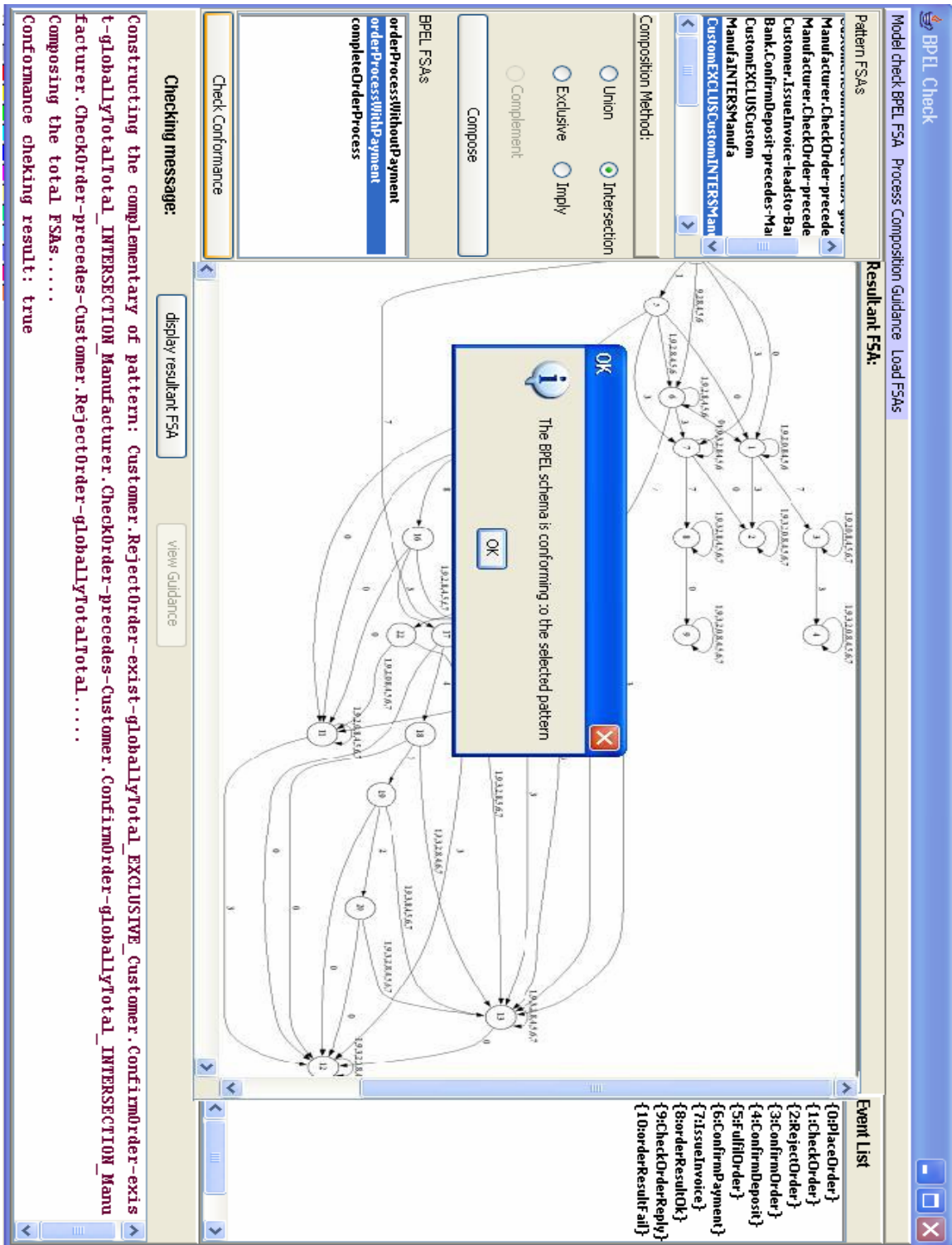


Figure 5.11 Verification result of the BPEL schema against Rule 1 – Conforming

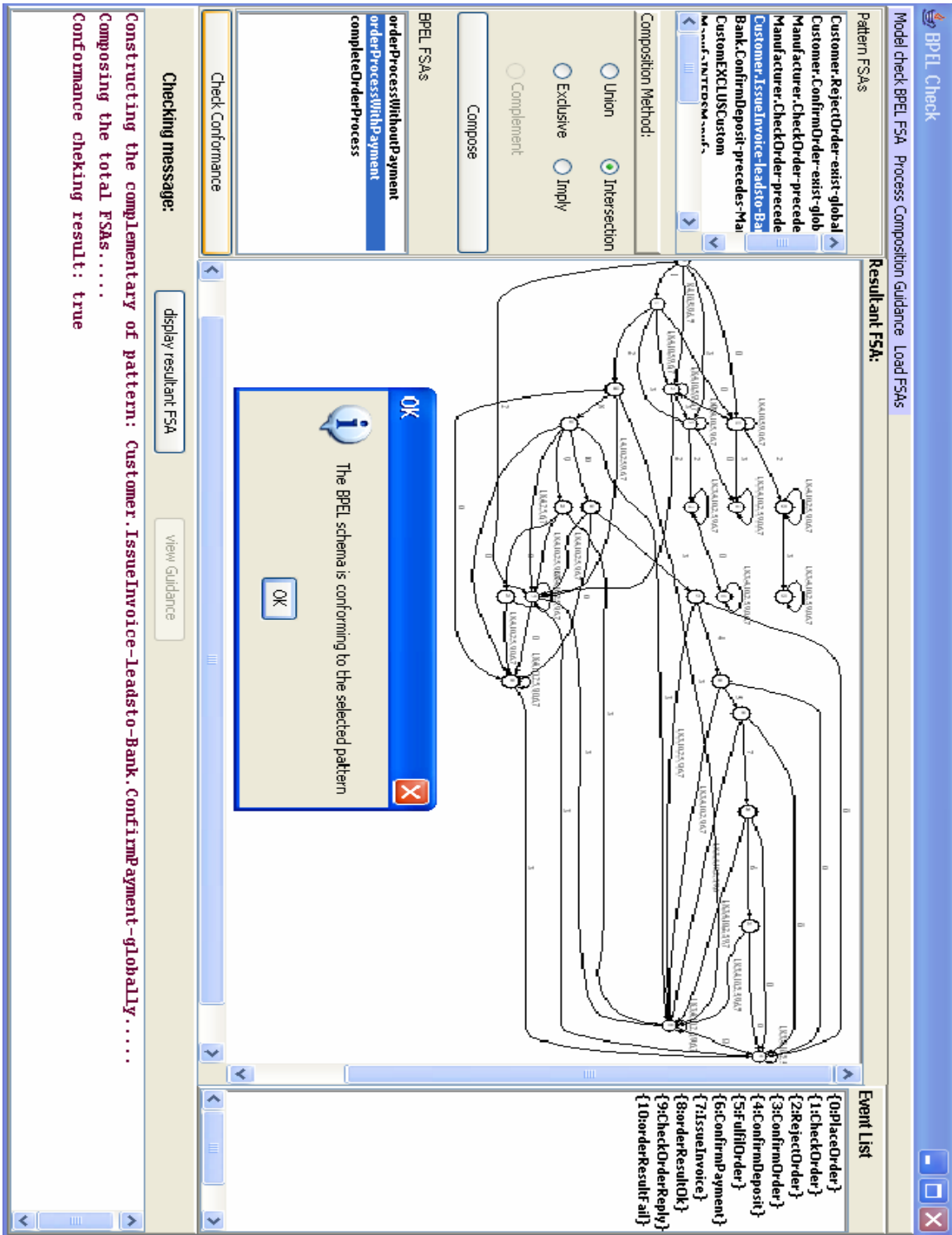


Figure 5.12 Verification result of the BPEL schema against Rule 2 - Conforming

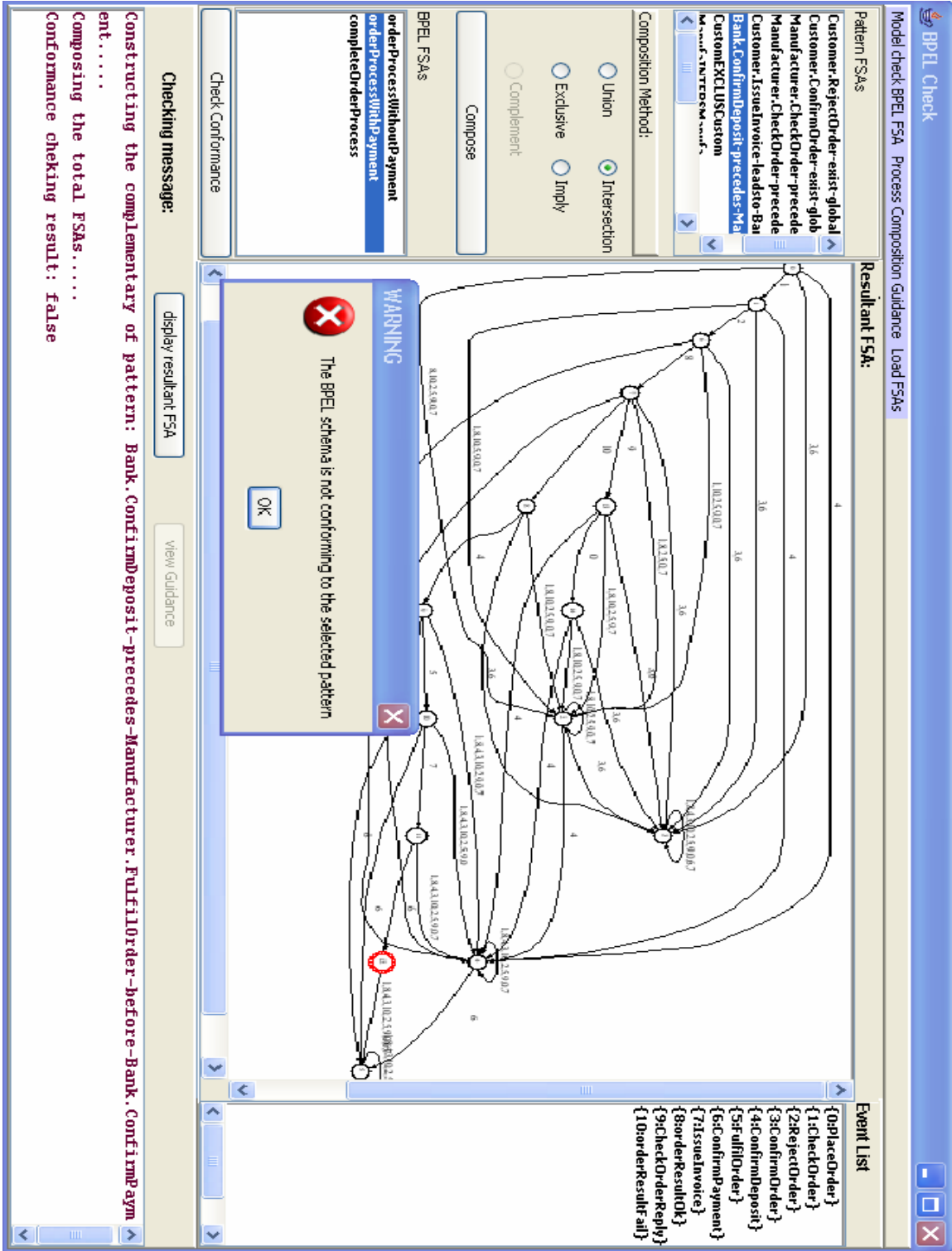


Figure 5.13 Verification result of the BPEL schema against Rule 3 – Not conforming

5.8 Summary

In this chapter, we have described the steps of the verification process, which represents both the property patterns and the BPEL schema in FSA, and then checks the language inclusion of the two FSAs. The result of the conformance checking can first be used to determine whether the designated BPEL schema satisfies all the specified requirements. Secondly, the result can be used to assist the service designers in designing a correct service composition. This is discussed in the next chapter.

6 Composition Guidance

In this chapter, we discuss the application of the conformance checking result for guiding the service designer in designing a correct service composition. The chapter starts by introducing a general approach to providing guidance. It then presents the procedure for analyzing the violations and generating remedy plans. It ends by showing how guidance can be used in action to aid the service designer with his composition process.

6.1 The guidance framework

The composition guidance framework is aimed at supporting the manual service composition process with some automated features. Our composition guidance follows a similar approach to [18]. However, we target service developers who are composing complex BPEL schemas while [18] targets end-user service composers who compose simpler and more linear service compositions.

Figure 6.1 depicts the general framework for providing guidance. It can be seen from this figure that we utilize the PROPOLS specification and the result of verification discussed in chapters 4 and 5 for generating guidance messages. During the composition process, whenever a syntactically correct BPEL schema is made available, it is model checked against the set of temporal business rules (the requirements). If the result of the model-checking shows no violation then there is no guidance. Otherwise, the violation is analysed and guidance is generated. The guidance message, including the error information and remedy plans, is then communicated back to the service composer for him to update the BPEL schema accordingly. The guidance message can be suggestions on next steps in the business process, identifications of missing steps, and/or propositions for activity re-ordering or substitution. This process is repeated until the service composer has a complete and correct service composition.

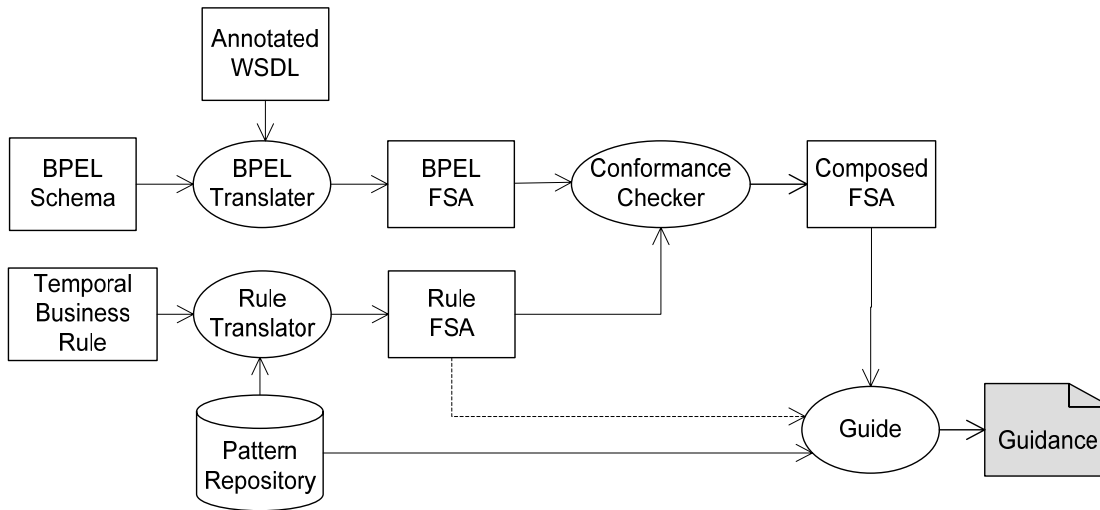


Figure 6.1 The guidance Framework [22]

6.2 An example violation scenario

This section presents a scenario in which, during the composition process, the service designer has a temporary BPEL schema that does not conform to the set requirements. In this scenario a service composer for the SOPU order process discussed in previous chapters has just added the latest activity `<receive>confirmPayment` from the Bank partner-link into the BPEL schema. The diagram for the updated BPEL schema is given in Figure 5.5 in the previous chapter.

Model-checking the schema against the rule “*Bank.ConfirmDeposit* precedes *Manufacturer.FulfilOrder* before *Bank.ConfirmPayment*” as in Figure 5.4 reveals a violation. That is because *Bank.ConfirmDeposit* has not been inserted into the BPEL schema at a position before *Manufacturer.FulfilOrder* as required by the rule. The resultant DFSA is given in Figure 5.10.

In the following sections, we will present the methods of analyzing the error situation and generating guidance and will refer to this violation scenario for illustration purposes.

6.3 Violation analysis

This section discusses the violation analysis phase of guidance generation. When a violation occurs, the resultant FSA, the rule TDFSA and the TDFSA of the BPEL process are all taken into analysis to reveal the root causes of the violation. This violation analysis process starts with identifying the error states and the corresponding error paths. It then identifies, along each path, the *error-entry* state and the corresponding *error-entry* event.

6.3.1 Identify error states

The error states are the set of accepting states in the resultant FSA. (For the mathematical foundation of this, please refer to Appendix 1). In the resultant FSA in Figure 5.10, the error state is state 12 and is highlighted in a double circle. After that, based on the naming structure of the resultant FSA, we identify the corresponding error states of the rule TDFSA and the process DFSA. For example, in our case, we query the name structure of the error state 12 and has the result $\{1: 3, 2: 5\}$, which means the corresponding state of the rule FSA is state 3 while that of the BPEL FSA is state 5. If the error state of the rule FSA exists in the rule FSA before the totalisation process, it is said to be *waiting*, otherwise it is said to be *trapped*. In this case, the error state is *trapped*.

6.3.2 Identify error paths

For each error state in the composed FSA, the corresponding error paths are identified. An error path is a sequence of states leading from the initial state to the error state. As the FSAs are deterministic, knowing an error path means we know the sequence of events corresponding to the transition along this path. Therefore, the term error path may refer to a sequence of states or a sequence of events interchangeably.

Algorithm 6.1 illustrates the procedure to identify all the error paths. This is a customized graph paths searching algorithm taking a “*from*” state, a “*to*” state, and the FSA to be searched as parameters and returning all the available paths from the “*from*” state to the “*to*” state. A resultant path is a sequence of states with no loop, which means any state on the path appears only once. Based on the sequence of states, it is straightforward to identify the sequence of events for the path.

The algorithm starts with a setup function to initialize all the necessary parameters and call the main recursive function. Initially the set of all paths contains only a single path and that single path contains only the “*to*” state. This path, together with the information about the “*from*” state and the FSA are passed to the recursive function.

The recursive function starts in line 8. It searches backward from the “*to*” state to the “*from*” state. The “*to*” state is initially the real “*to*” state and will be updated as the search goes backward. For any current “*to*” state, the algorithm identifies all the “*previous*” states, those having transitions leading to the current “*to*” state, ignoring states that already exist in the current path, and for each identified “*previous*” state, it clones the current path, prepends that state into the cloned path and adds the new path to the “*all paths*” set. The algorithm then recursively continues with the “*to*” state now becoming the “*previous*” state, until it reaches the “*from*” state. The “*all paths*” set, by then, is filled with all the possible paths between this “*from*” and the “*to*” state.

```

1 FUNCTION findPath(fsa, from, to) AS Set<Paths>
2   allPaths = empty Set of type Path
3   initialPath = to
4   allPaths.add(initialPath)
5   recursiveFindPath(fsa, initialPath, from, allPaths)
6   RETURN allpaths
7 END
8 FUNCTION recursiveFindPath(fsa, curPath, FSAtate from, allPaths)
9   lastState = first state in curPath
10  FOR EACH prevState IN fsa.getPredecessor(states)
11    IF path.contains(prevState) THEN
12      CONTINUE
13    END IF
14    newPath = clonePath(path)
15    newPath = prevState + newPat
16    IF prevState.equals(from) THEN
17      allPaths.add(newPath)
18    END IF

```

```

19 recursiveFindPath(fsa, newPath, from, allPaths)
20 END FOR
21 END
    
```

Algorithm 6.1 Identify error paths

In our case, the error state of the resultant FSA is state 12. Applying the find path algorithm above into the resultant FSA in Figure 5.10 (error state 12) returns only one error path. Mapping this error path to the BPEL schema reveals the BPEL error path highlighted in dotted line as in Figure 6.2 below.

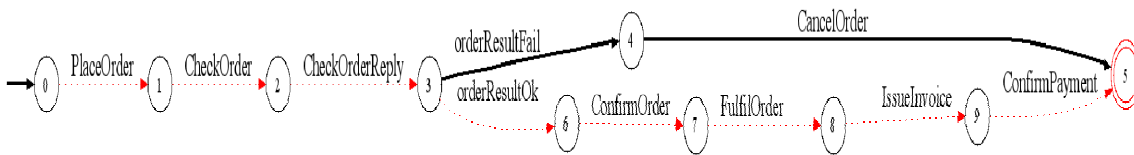


Figure 6.2 The BPEL DFSA with error path highlighted in dotted

6.3.3 Identify error-entry states and error-entry events

For each error state in the resultant FSA, the corresponding error state of the rule FSA is identified. Along each error path of the resultant FSA, the *error-entry* state is the first state that corresponds to the rule FSA’s error state. The *error-entry* state is the same as the error state or positions earlier in the error path. The event that brings the resultant FSA from the last accepting state to the *error-entry* state is called the *error-entry* event. The identification of the *error-entry* event is important, as there always exists one remedy for any violation - the removal of the *error-entry* event

In our example, the *error-entry* state is the same as the error state, which is corresponding to the state 5 of the BPEL FSA. The *error-entry* event is thus *Bank.ConfirmPayment*.

6.4 Automatic remedy plans generation

Knowing the causes of the error, the guidance can be automatically generated using a number of algorithms that we have developed. Each algorithm targets a specific remedy strategy such as forward extension, backward insertion, backward correction, deletion and reordering.

It is worth noting that, since the above algorithm works on the rule FSA, the returned sequence consists of business activities. As there may generally be one-to-many mappings between the activities and WS operations, we do not automatically convert them to WS operations but rather let the BPEL designer make the choice. The details of each strategy are discussed below.

6.4.1 Forward extension

The forward extension strategy attempts to find the shortest sequence of business activities that, if added to the end of the schema, can make the BPEL FSA conform to the violated rule. When generating remedy plans, priority is given to the plan that does not affect the current composed schema. Therefore, forward extension is the preferred strategy. Forward extension will succeed only if there are transitions out of the error state thus it is only applied when the error state is *waiting*.

Algorithm 6.2 presents the implementation of the forward extension strategy. It is applied to the rule FSA to find a sequence of events which can bring the rule FSA out of the current *waiting* state. The algorithm is a general graph depth-first search algorithm, which searches for a path from a given state to any accepting state. It recursively traverses all the paths leading out of the “*from*” state, examining the acceptability of the target states, and returns the path to the first accepting state found. When applying the algorithm to the rule FSA the *waiting* state of the rule FSA will be set as the initial “*fromState*”. The returned path will be recommended to the service designer as the next steps of the business process.

```

1 FUNCTION findPathToFinalState(fromState, fsa, excludedStates)
2   IF NOT fsa.contains(aState) THEN
3     RETURN NULL
4   END IF
5   FOR EACH nState IN ruleFSA.getNextStates(fromState)
6     IF nState IN excludedStates THEN
7       CONTINUE
8     ELSE IF nState.isFinal() THEN
9       RETURN <nState>
10    ELSE
11      excludedStates.add(nState)
12      path = findPathToFinalState(nState,
                                   ruleFSA,
                                   excludedStates)
13    END IF
14    IF path != NULL THEN
15      RETURN <nState> + path
16    END IF
17  END FOR
18  RETURN NULL
19 END

```

Algorithm 6.2 Find path to DFSA final (accepting) state

6.4.2 Backward insertion

The backward insertion strategy is applied when the user has missed a sequence of activities. In this case, the *error-entry* state is typically *trapped*. The strategy attempts identifying previous missing steps in an error path by exploring the rule FSA from a state close to the error, and suggests their insertion in the BPEL schema.

Algorithm 6.3 and 6.4 together implement the backward insertion strategy. The *findPreviousMissingSteps* algorithm (algorithm 6.3) is the set up function. It backtracks the error path one step at a time and for each current state *cState* in the error path, it calls

the *missingStepsAfterState()* function (algorithm 6.4) to identify the missing steps between the previous state *pState* and the current state.

The *missingStepsAfterState()* is a recursive function that, given a current state *cState* of the rule FSA and an error path *errorPath*, returns a sequence of activities and the location to insert. It examines if the sequence fragment of *errorPath* after *pState* is acceptable at any state reachable from *pState*, and returns an activity sequence between *pState* and that state. The set *excludedStates* is used to prevent infinite execution loops. The function *isActSeqAcceptedFromState* tests if a given sequence leads the rule FSA from the given state to a final (accepting) state.

```

1 FUNCTION findPreviousMissingSteps(cState, errorPath, ruleFSA)
                                AS (cState, SeqToInsert)
2 BEGIN
3   IF NOT ruleFSA.contains(cState) OR cState.isInitial()
4     RETURN NULL
5   prevState = ruleFSA.getPreviousStateByPath(cState, errorPath)
6   postActSeq = ruleFSA.getActSeqAfterStateByPath(pState, errorPath)
7   actSeqToIns = missingStepsAfterState(pState, postActSeq, ruleFSA,  $\emptyset$ )
8   IF actSeqToIns != NULL
9     RETURN (cState, actSeqToIns)
10  ELSE
11    RETURN findPreviousMissingSteps(prevState, errorPath, ruleFSA)
12 END

```

Algorithm 6.3 Find previous missing steps

```

1 FUNCTION missingStepsAfterState(cState, actSeq, ruleFSA, excludedStates)
AS eventSequence
2 BEGIN
3   IF ruleFSA.isActSeqAcceptedFromState(cState, actSeq)
4     RETURN <>
5   excludedStates.add(cState)
6   FOR EACH (nEvent, nState) IN ruleFSA.getNextEventStatePairs(cState)
7     IF nState IN excludedStates
8       CONTINUE
9     END IF
10    IF ruleFSA.isActSeqAcceptedFromState(nState, actSeq)
11      RETURN <nEvent>
12    ELSE
13      actSeqToIns = missingStepsAfterState(nState, actSeq, ruleFSA)
14      IF actSeqToIns != NULL
15        RETURN <nEvent> + actSeqToIns
16      END IF
17    END IF
18  END FOR
19  RETURN NULL
20 END

```

Algorithm 6.4 Find missing steps after a state

Applying this algorithm to our violation scenario results in a suggestion that *Bank.ConfirmDeposit* be inserted before *Manufacturer.FulfilOrder*.

6.4.3 Backward correction

The backward correction strategy is applied when the user has composed a process skeleton and does not want to change the structure of that skeleton, that is, he wants to

maintain the order of all activities in the process. However, as he has missed some activities in the process, the composed process violates the rules. The backward correction strategy is a generalized and more comprehensive version of the backward insertion strategy of the previous section. Due to its high complexity, backward correction is applied only when backward insertion results in no solution. Similar to backward insertion, the backward correction algorithm works when the error state is trapped.

The backward correction strategy works by back tracking the rule FSA error path and for each state on that path, it attempts to “correct” the composition by filling in all the missing activities as required by the rule FSA. The back tracking process stops as soon as a solution can be found. Specifically, at each state, an alternate path that accepts all the subsequent events of the error paths is searched for. For example, if the error path is $\langle A1, A2, A3, A4, A5 \rangle$ then the algorithm with backtrack in order $\langle A5, A4, A3, A2, A1 \rangle$. At each step, such as at A3, it tries to search for an alternate path in the rule FSA that eventually accept the subsequent events, in this case, $\langle A4, A5 \rangle$ in that order. For example, if from state A3, there is a sequence of events $\langle A6, A4, A7, A5 \rangle$ found then that sequence is recommended to the service designer. Specifically, the service designer would need to insert activity A6 after A3 and before A4; and insert A7 after A4, before A5 so that he has a final sequence of activities $\langle A1, A2, A3, A6, A4, A7, A5 \rangle$ that supersedes his original sequence $\langle A1, A2, A3, A4, A5 \rangle$.

The implementation of the backward correction strategy uses algorithm 6.3 above and algorithm 6.5 in place of algorithm 6.4. Algorithm 6.5 is a customized recursive backtracking graph search function that attempts all the possible paths leading out of the current state of the FSA for the acceptance of the passed ordered events set. It explores paths until all the events in the event sequence, in that order, are accepted by the FSA. If not successful with a path, it backtracks, resets the variables, and attempts other paths.

```

1 FUNCTION findPathAcceptEvtSeq(from, ruleFSA, evtSeq,
excludedSet, index, path) AS BOOLEAN
2   FOR EACH (nEvent, nState) IN ruleFSA.getNStateEventPairs(from)
3     IF excludedSet.containsPair(nState, index) THEN
4       CONTINUE
5     ELSE
6       excludedSet.addPair(nState, index)
7     END IF
8     BOOLEAN foundAMatch = FALSE
9     BOOLEAN evtAddedToPath = FALSE
10    IF nEvent == evtSeq.get(index) THEN
11      index ++
12      path += nEvent
13      //found the match for the entire sequence,
14      //stop searching and return
15      IF index == (evtSeq.size() - 1) AND nState.isFinal() THEN
16        RETURN TRUE
17      END IF
18      foundAMatch = TRUE
19      evtAddedToPath = TRUE
20      //avoid duplication of event in the sequence
21      ELSE IF evtSeq.contains(e) THEN
22        CONTINUE
23        //need this event to go to other state
24      ELSE IF nState != from THEN
25        path += nEvent
26        evtAddedToPath = TRUE
27      END IF
28      IF findPathAcceptEvtSeq (nState, ruleFsa, evtSeq,
excludedSet, index, path) THEN
29        RETURN TRUE
30      ELSE
31        //no solution found while traversing that path
32        //remove added event in return path and
33        //decrement the index if incremented
34      IF evtAddedToPath THEN
35        path -= nEvent

```

```

29         END IF
           //a match was found but not the entire sequence
           //so decrement the index
30         IF foundAMatch THEN
31             index --
32         END IF
33     END IF
34 END FOR
35 RETURN FALSE
36 END

```

Algorithm 6.5 Find path accepting an events sequence

6.4.4 Deletion and reordering

Deletion is a straightforward strategy. It suggests the removal of the error-entry event from the BPEL schema. Typically, deletion is applied when there is no other possible remedy plan.

Reordering is also a simple strategy. It aims at bringing a blocked event out of the applicable scope by reordering the blocked event and the ending scope of the rule. A reordering algorithm attempts to backtrack the error path, search for the last state that accepts the ending scope and put the ending scope at this state. The benefit of this strategy is it does not force the designer to introduce new web services operations or remove some web services operations from his schema to remedy the violation

6.5 Pattern specific guidance

Apart from the automatic guidance generation method, we can apply the construct-specific remedies method. This method involves human effort in predefining the remedy plans for the violation of individual rules. The defined remedy plans are stored in a pattern repository and are referred to after the error analysis phase revealing which patterns are violated. In Table 6.1, we illustrate the principles for defining remedies by “*precedes*” and “*leads to*” pattern constructs involving two scopes: *globally* and *before*. The other constructs can be worked out in a similar fashion, and are omitted here due to

space limitations. In the FSA semantics, O refers to any other events/activities other than those explicitly mentioned in the FSA.

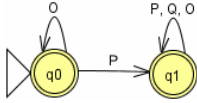
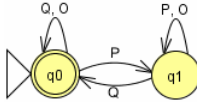
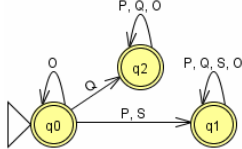
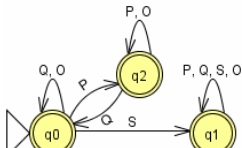
Rule patterns	FSA Semantics	Symptom/Diagnosis	Suggestions
P precedes Q globally		Violations must be caused by rejections, which occur only at state q0 in the FSA. Q is always the error-entry activity.	1) Remove the Q, or 2) Insert P before Q (as Q is only acceptable in state q1)
P leads to Q globally		Violations occur only when the FSA is at non-accepting state q1 (case 2). P is always the error-entry activity	1) Remove the P, or 2) Insert Q after P (to move the FSA out of state q1)
P precedes Q before S		Violations must be caused by rejections at state q2, i.e., P fails to appear before Q and S, and Q appears before S. S is always the error-entry activity.	1) Remove S, or 2) Insert P before the first Q (as sequence Q.S is only acceptable at states q1)
P leads to Q before S		Violations must be caused by rejections at state q2, i.e., Q fails to appear between P and S, and P appears before S. S is always the error-entry activity.	1) Remove S, or 2) Insert Q after the last P before S (as starting from state q2, S is only acceptable at q0)

Table 6.1 Example Predefined Remedies [22]

According to Table 6.1, the example violation scenarios will result in a suggestion either to “*REMOVE Bank.ConfirmPayment*” or to “*INSERT Bank.ConfirmDeposit before Manufacturer.IssueInvoice*”. This method of guidance generation has the advantage that it can make use of human knowledge and results in more general remedies than the

automatic guidance generation methods of section 6.4 . Also, there is little computational cost associated with this method. For pattern constructs with simple FSA semantics such as those listed in Table 1, the remedy writing can be straightforward. The automatic guidance generation method, on the other hand, can be used to generate remedy plans for complex rules, typically those involving logical compositions. These two methods thus complement each other.

6.6 Example Guidance Results

Figure 6.3 shows the final steps of the example composition process with guidance aid.

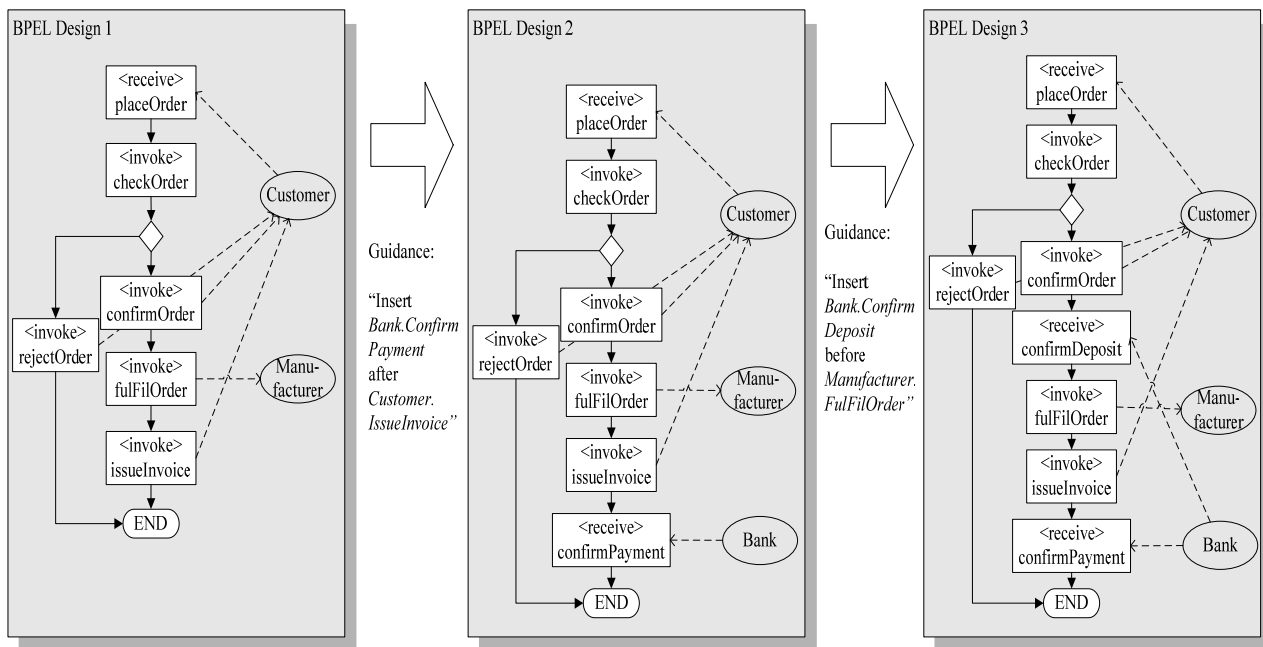


Figure 6.3 Composition progress with guidance aid [22]

The details of guidance given in each step are discussed below.

In Design 1, Figure 6.3, after inserting the activities *<invoke>issueInvoice* into the schema and saving the files, the service designer received a message informing that the rule “*Manufacturer.IssueInvoice* leads to *Bank.ConfirmPayment* globally” has been

violated. The guidance message given to the designer was to “*INSERT ConfirmPayment after IssueInvoice*” or to “*REMOVE IssueInvoice*” as shown in Figure 6.4.

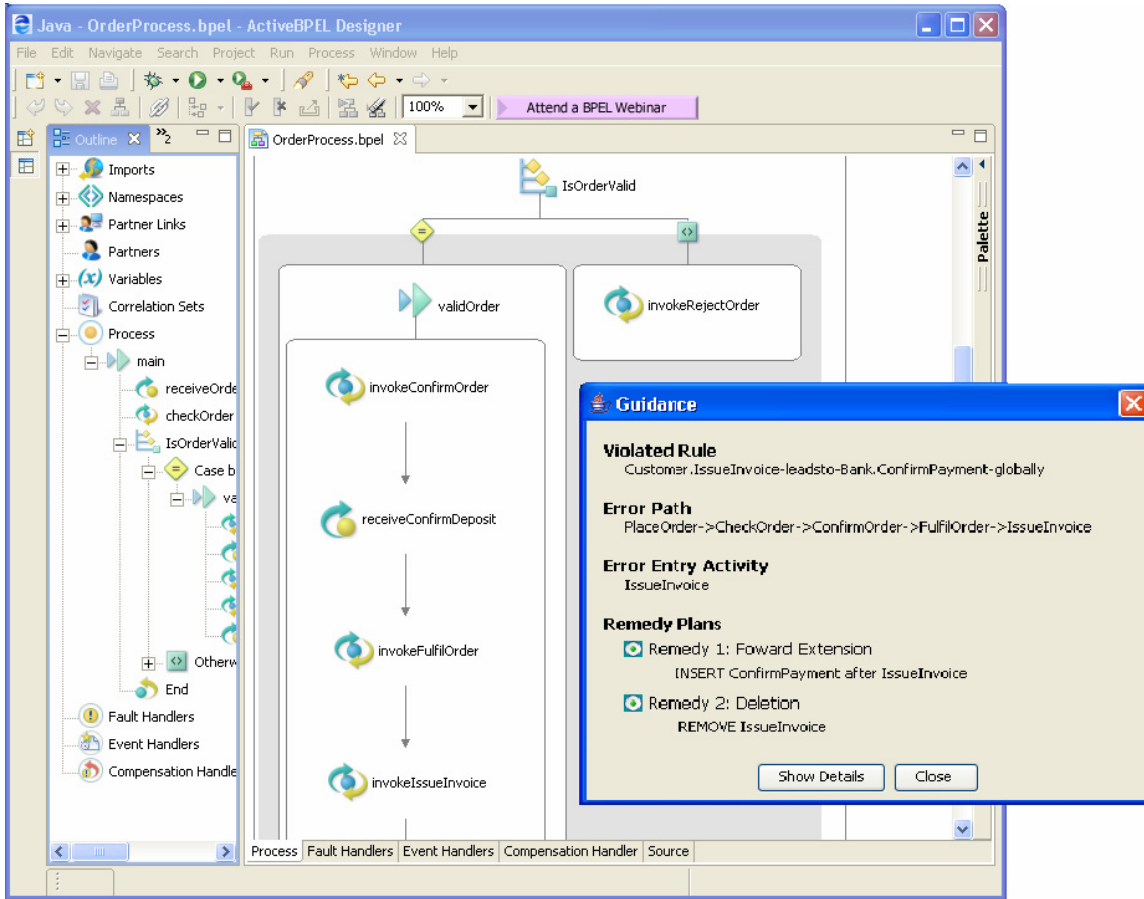


Figure 6.4 Forward extension guidance

The designer chooses to follow the first suggestion, resolves the *ConfirmPayment* business activity into the `<receive>ConfirmPayment()` web services activity and inserts it into the BPEL schema. The designer now has a schema as shown in Design 2 in Figure 6.3. However, another violation is detected. This time, the violated rule is “*Manufacturer.ConfirmOrder precedes Bank.ConfirmDeposit before Bank.ConfirmPayment*”. The guidance message suggests the service designer to “*INSERT ConfirmOrder Before ConfirmDeposit*” or to “*REMOVE ConfirmPayment*” as in Figure

6.5. The designer, again, follows the first suggestion, resolves *ConfirmOrder* into the web services activity `<invoke>confirmOrder()` and inserts into the BPEL schema at a location before the web services activity `<invoke>confirmDeposit()`.

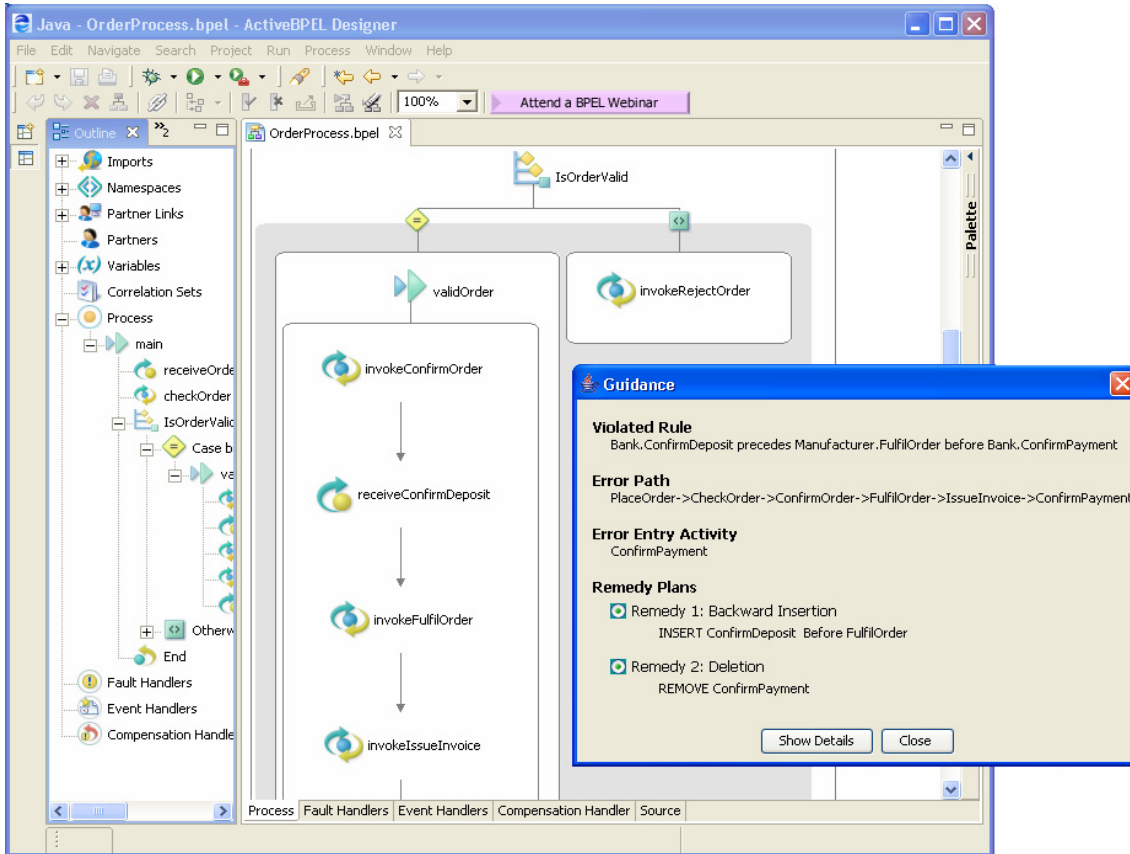


Figure 6.5 Backward insertion guidance

After saving the composition, the model-checking of the updated BPEL schema against the business rule shows no violations. The designer thus now has a complete and correct BPEL schema as shown in Design 3, Figure 6.3.

6.7 Summary

In this chapter, we have described the general approach to providing composition guidance, which utilizes the conformance checking result and generates remedy plans. A number of strategies including forward extension, backward insertion and correction, reordering and deletion are automated. This is complemented with the predefined remedy method. They together provide a comprehensive remedy generation framework.

7 Tool Implementation

This chapter presents the design and implementation of the Java prototype tool BPELCheck4Guide. It starts with an overview of the architecture followed by the detailed descriptions of its components.

7.1 Design

The general package structure of BPELCheck4Guide is given in Figure 7.1. The *GUI* package contains the graphical user interface implementation of the prototype. The internal representation and manipulation of FSAs are implemented in the *FSA* package. The verification functions are implemented in the *checker* package, while the *guider* package encapsulates all the guidance generation functionalities. These two packages both use the *FSA* package for internal representation of rules and BPEL process, and the *parser* package for parsing user inputs. An initial implementation of the PROPOLS language is also provided in the *PROPOLIS* package and its sub packages.

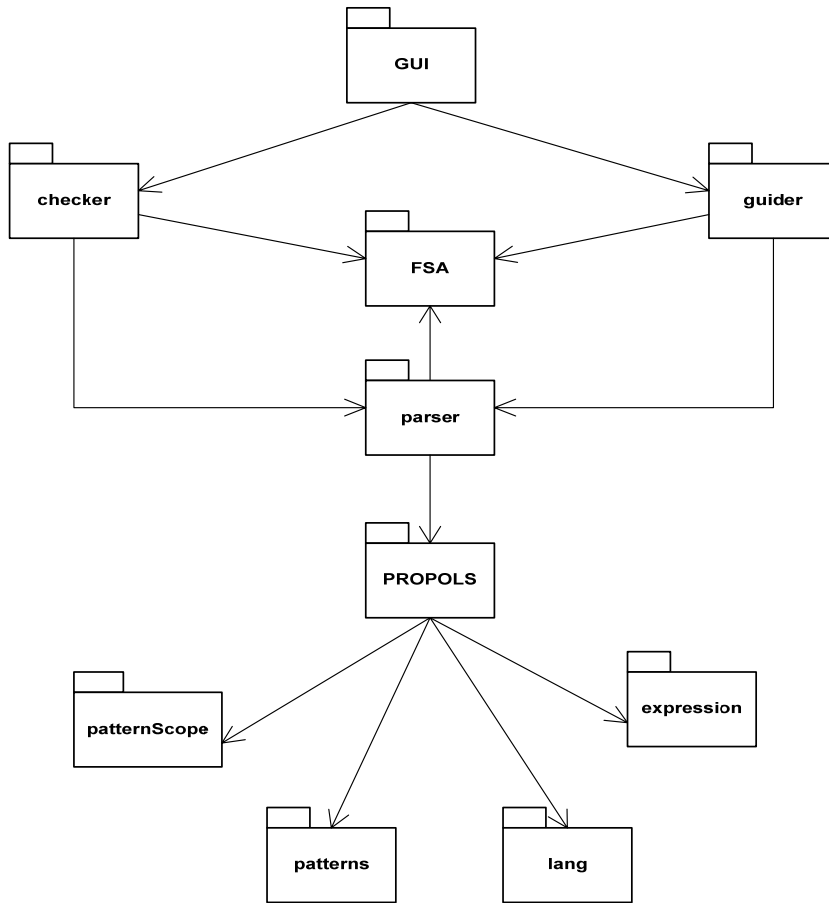


Figure 7.1 Design of BPELCheck4Guide

7.2 Package “FSA”

Figure 7.2 depicts the structure of the *FSA* package. The *DFSA* class represents a Deterministic FSA and implements the interface *FSA*, which names the collection of all methods in a general FSA. A *DFSA* instance contains a number of *FSAState*, *FSATransition* and *Event* objects. The *DFSA* class also contains methods to output its structure into the input language of the Graphviz Dot [17] program. The *FSAUtils* class provides a collection of algorithms and utilities to manipulate the *DFSA*.

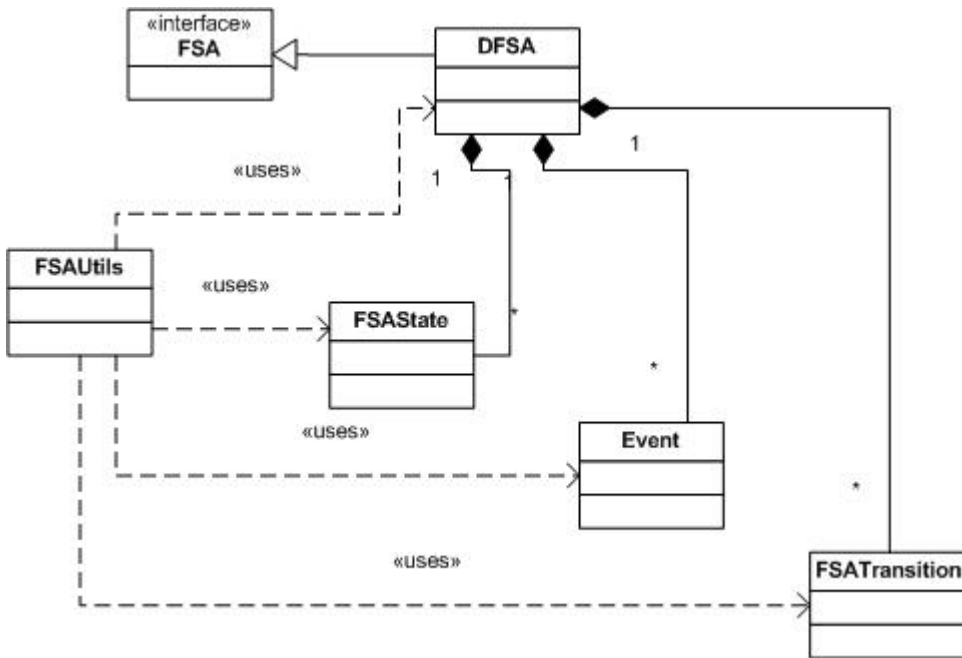


Figure 7.2 Structure of the “FSA” package

7.3 Package “Checker”

Figure 7.3 depicts the structure of the *checker* package. The main class *Checker* represents the conformance checking process for rule FSA and process FSA. It uses various FSA manipulation algorithms and set operations from the *FSAManipulator* and *SetManipulator* utility classes respectively.

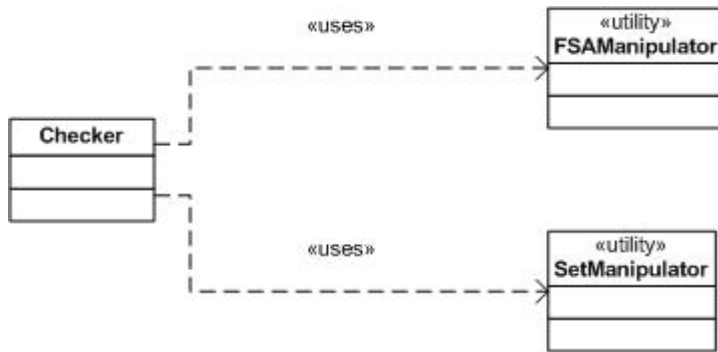


Figure 7.3 Structure of the “Checker” package

7.4 Package “Guider”

Figure 7.4 depicts the structure of the *guider* package. The *ErrorInformation* class represents a violation message with details about the violated rules, the violated event sequence and position. The *GuidanceMessage* class represents a guidance message, which contains information about the remedy plans, the which, when and where details of actions to be performed to remedy the error situation. The main class of the package is *Guider*, which provides methods to analyze and report the violations and then automatically generate remedy plans for a specific violation scenario. *Guider* class uses various utilities and algorithms provided in the *GuiderUtils* class to generate *ErrorInformation* and *GuidanceMessage* objects.

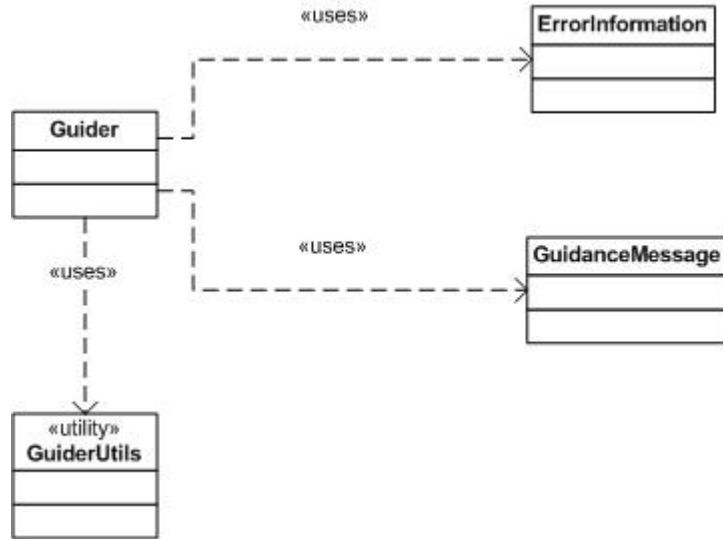


Figure 7.4 Structure of the “Guider” package

7.5 Package “GUI”

Figure 7.5 depicts the structure of the *GUI* package. The class *MainUI* is the main entry point into the BPELCheck4Guide software. It contains various graphical components for presenting the checking and guiding information to the end users and to collect user inputs. It utilizes the Graphviz Dot program for FSA visualization. The *GuidanceDialog* class presents violation details and remedy plans to the user. It also utilizes Graphviz Dot to visualize the composed BPEL schema as a graph. *ImagePanel* class can displays any DFSA graph as an image. Both *MainUI* and *GuidanceDialog* have an instance of *ImagePanel*. The *RemedyPlanPanel* class displays the information of a remedy plan.

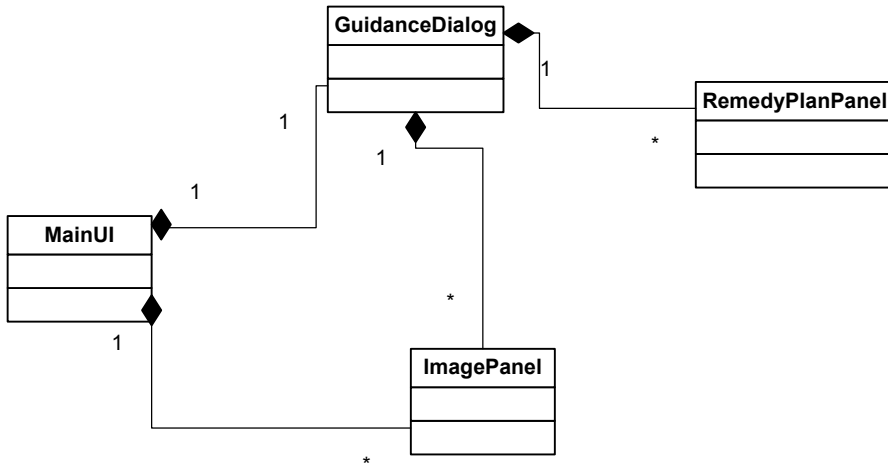


Figure 7.5 Structure of the “GUI” package

7.6 Summary

In this chapter, we have presented the architecture and design of the BPELCheck4Guide prototype. For the technical details of the tool, please refer to Appendix 2.

8 Conclusion

This chapter provides a summary of the goals and accomplishments of this research work. It discusses the limitations of the work and the areas that could be explored further.

8.1 Goals and Accomplishments

The essential goal of this thesis was to develop a framework for supporting service composition engineering including: property specification, conformance verification and composition guidance. The main accomplishments of this thesis are presented below.

Property specification

The development of the PROPOLS language as an extension to PPS allows practitioners to specify temporal properties for service composition. The language was designed as an ontology language to facilitate knowledge sharing. The language is based on OWL and has Finite State Automata based semantics.

Conformance checking

An approach to verifying a service composition (specifically a BPEL schema) against a set of properties expressed in PROPOLS has been developed. In this approach, the BPEL schema is converted to a TDFSA (called the BPEL TDFSA) utilizing Foster's BPEL to LTS method, and temporal business rules in PROPOLS are also converted to TDFSA representation (rule TDFSA). The conformance checking method is then treated as the testing for the non-emptiness of the inclusion between the complement of the rule TDFSA and the BPEL TDFSA.

Composition guidance

We have also presented a framework for providing guidance during the service composition process. While composing the service, whenever a new version of the BPEL schema is made available, it is model checked against the set of properties specified in

PROPOLS. If any violation is identified, the violation is then analyzed and guidance messages are generated based on some pre-defined remedies or some guidance generation algorithms.

Prototype Tool

A prototype tool named BPELCheck4Guide was developed to provide conformance checking and composition guidance. Apart from that, the tool also allows manipulating and visualizing FSAs.

8.2 Limitations and Future Work

Parameters consideration

Parameters make the temporal relationship between web services composition more meaningful and precise. For example, the rule “*IssueInvoice* leads to *ConfirmPayment* globally” makes more sense if both the invoice and the payment are for the same order. Therefore, it would be desirable to have the rule specified as “*IssueInvoice* leads to *ConfirmPayment* where *IssueInvoice.orderId = ConfirmPayment.orderId*”. However, in our current approach, the parameters are not considered for two reasons 1) Our work deals with composition time and pre-deployment time of the service composition when parameter values are unknown 2) Foster’s BPEL to LTS method does not consider web service operation parameters. Parameters consideration would be our priority for future work.

Runtime validation

Our conformance-checking model is static verification. It has some limitations that can only be solved when validating the service composition at runtime. One of the limitations is the support for the quantified *exists* property patterns. It is impossible to determine whether a rule like “*ConfirmPayment* exists at most twice globally” is violated given a BPEL schema that has *ConfirmPayment* positioning in the body of a loop as we can not be certain how many times the web services activities corresponding to *ConfirmPayment*

will be called at runtime. The runtime validation will complement our current verification and is the topic of another future work.

Forward guidance

Currently, guidance is only provided when there are errors encountered. It would be desirable to provide forward guidance – proactive guidance even when there is no violation. This can be done if the BPEL schema is parsed by a custom parser rather than relying on the BPEL2LTS tool, so that the latest composition activity can be identified.

User profile / user preferences consideration in guidance

As guidance involves human computer interaction, considering a user's profile would allow providing better guidance messages. The guidance messages can be filtered and classified according to the user's preferences so that only the most suitable guidance is recommended to the user.

The prototype implementation

The prototype has been implemented as a proof of concept. It has some components unimplemented and some components needs optimization. Specifically, a PROPOLS parser needs to be implemented. A BPEL2FSA module also needs to be developed once Foster's BPEL2LTS APIs are made available. The prototype implementation involves many graph algorithms. These algorithms are subject to optimization.

Summary

This chapter concludes the body of this thesis. We have outlined the achievements, which include the verification and composition guidance for service composition, and the development of PROPOLS and the prototype. We have also identified some areas that require further attention in order to increase the usability of our approach in a practical development environment.

Appendix 1: Formal Mathematical Background for Verification and Guidance

Label Transition System (LTS), Finite State Automata (FSA) are formalisms employed for internal representation of BPEL schema and rules in our approach. Appendix 1 presents a general introduction to LTS, FSA and their characteristics. It also presents the semantics for FSA composition and conformance verification.

Labeled Transition System

Definition 1 Labelled Transition System

A Labelled Transition System or LTS is a tuple (S, L, δ) where
 S is a finite set of state,
 L is a finite set of labels,
 $\delta: S \times L \rightarrow S$ is the transition relation,

Finite State Automata

Finite State Automata or FSA represent models of behavior composed of states and transitions. State stores information that reflects changes in the modeled system from its initial state. Transitions indicate state changes and the conditions or the triggers for those changes. FSA are characterized by a single initial state, a set of accepting states and a set of transitions between the states. The set of all transition labels is said to be the alphabet of the FSA.

Definition 1: Finite State Automata

A Finite State Automata or FSA is a tuple (S, s_0, L, δ, F) where
 S is a finite set of state,
 s_0 is the initial state with $s_0 \in S$,
 L is a finite set of labels,
 $\delta: S \times L \rightarrow S$ is the transition function,

F is a set of accepting states with $F \subseteq S$.

Given a FSA A , an event $e \in L$ is said to be *rejected* at a state $s \in S$ if $\delta(q; e)$ is undefined. Otherwise, e is *accepted* at s by A . Unlike LTS, FSA has a finite set of states and transitions and has one and only one initial state.

Total Deterministic FSA and Complementary of FSA

A FSA A is said to be a Total Deterministic Finite State Automata or TDFA if it satisfies the following restrictions.

1. It is total, which means at every state of a TDFA A , every label $l \in L$ is accepted.
2. It is deterministic, which means at every state $s \in S$, a label $l \in L$ would mark only a unique transition.

Definition 2: Total Deterministic FSA

A total Deterministic FSA or TDFA is a tuple (S, s_0, L, δ, F) where
 S is a finite set of state,
 s_0 is the initial state with $s_0 \in S$,
 L is a finite set of labels,
 $\delta: S \times L \rightarrow S$ is a total function called the transition function,
 F is a set of accepting states with $F \subseteq S$.

The complement of a TDFA A is a TDFA $\neg A$ with the same transition diagram as A but for every accepting state of A , that of $\neg A$ is non-accepting and vice versa. Therefore, the complement of a TDFA is constructed by making all non-accepting states accepting and vice versa. Note that this definition is only applicable to Total DFSA.

Definition 3: Complement

The complement of a TDFA $A=(S, s_0, L, \delta, F)$ is a TDFA $\neg A = (S', s'_0, L', \delta', F')$ where
 $S' = S, s'_0=s_0, L'=L, \delta'=\delta,$
 $F' = S-F.$

TDFSA intersection

The intersection of two TDFA is constructed by making a Cartesian product of their states, and only the product of two accepting states makes an accepting state in the intersection TDFA.

Definition 4(intersection):

The intersection of two TDFAs A and B is a TDFA $A \wedge B$ such that $A \wedge B = (S, s_0, L, \delta, F)$ where
 S is the Cartesian product $A.S \times B.S,$
 s_0 is the tuple $(A.s_0, B.s_0),$
 L is the union $A.L \cup B.L,$
 δ is $\{(a_1, b_1), l, (a_2, b_2) \mid (a_1, l, a_2) \in A.\delta \wedge (b_1, l, b_2) \in B.\delta\},$
 F is $\{(s_1, s_2) \in S \mid s_1 \in A.F \wedge s_2 \in B.F\}.$

Accordingly, the union, exclusive or, and imply of two TDFA are also defined as the Cartesian product of their states, with the acceptability of the composed accepting states stated as follows.

Union: F is $\{(s_1, s_2) \in S \mid s_1 \in A.F \vee s_2 \in B.F\}.$

In a union, the state of the composite FSA is the OR of the acceptability of the two component states. It is accepting if any of the two component states is accepting and is not accepting otherwise

Exclusive Or: F is $\{(s_1, s_2) \in S \mid s_1 \in A.F \oplus s_2 \in B.F\}.$

Appendix 1: Formal Mathematical Background for Verification And Guidance

In an exclusive or, the state of the composite FSA is the XOR of the acceptability of the two component states. It is accepting if any only if one of the two component states is accepting and the other is not; it is not accepting otherwise.

Imply: F is $\{(s1, s2) \in S \mid s1 \notin A.F \vee s2 \in B.F\}$.

In an imply, the state of the composite FSA is accepting if the first component state is not accepting and the second one is accepting. It is not accepting otherwise.

Language intersection

Theorem 1: $L(A) \cap L(B) = L(A \wedge B)$.

This theorem stated that the intersection of two language $L(A)$ and $L(B)$, represented by TDFSA A and B respectively, is equivalent to the intersection of the language defined by the intersection of TDFSA A and TDFSA B . The following is the proof for that theorem

Proof: Suppose $w: l0l1l2...ln-1$ is a word of two TDFAs A and B , we can find an accepting run of A $(a0, l0, a1)(a1, l1, a2)...(an-1, ln-1, an)$ and an accepting run of B $(b0, l0, b1)(b1, l1, b2)...(bn-1, ln-1, bn)$ where

$\forall 0 \leq i < n. (ai, li, ai+1) \in A.\delta$ and $\forall 0 \leq i < n. (bi, li, bi+1) \in B.\delta$, and

$an \in A.F$ and $bn \in B.F$;

According to Definition 1, $\forall 0 \leq i < n. ((ai, bi), li, (ai+1, bi+1)) \in (A \wedge B).\delta$, and $(an, bn) \in (A \wedge B).F$,

So $l0l1l2...ln-1$ is a word of $L(A \wedge B)$.

Vice versa, we can get that if $l0l1l2...ln-1$ is a word of TDFSA $A \wedge B$, then it is a word of two TDFAs A and B .

Definition 5: The TDFSA semantics of an And-composite pattern 'pr1 **And** pr2' is defined as:
 $TDFSA(\text{pr1 And pr2}) = TDFSA(\text{pr1}) \wedge TDFSA(\text{pr2})$.

Based on Theorem 1, $L(TDFSA(\text{pr1 And pr2})) = L(TDFSA(\text{pr1})) \cap L(TDFSA(\text{pr2}))$, that means any execution sequence $s1$ that is a word of (conforms to) a composite pattern property 'pr1 **And** pr2', is also a word of pr1 AND a word of pr2.

Accordingly, we have:

Appendix 1: Formal Mathematical Background for Verification And Guidance

$$- L(\text{T DFA}(\mathbf{Not} \text{ pr1})) = \neg \text{T DFA}(\text{pr1});$$

$$- L(\text{T DFA}(\text{pr1} \mathbf{Or} \text{ pr2})) = L(\text{T DFA}(\text{pr1})) \cup L(\text{T DFA}(\text{pr2}));$$

$$- L(\text{T DFA}(\text{pr1} \mathbf{XOR} \text{ pr2})) = (L(\text{T DFA}(\text{pr1})) \cap \neg L(\text{T DFA}(\text{pr2}))) \cup (\neg L(\text{T DFA}(\text{pr1})) \cap L(\text{T DFA}(\text{pr2})));$$

$$- L(\text{T DFA}(\text{pr1} \mathbf{Imply} \text{ pr2})) = \neg L(\text{T DFA}(\text{pr1})) \cup L(\text{T DFA}(\text{pr2})).$$

Appendix 2: Prototype Tool Technical Details

This appendix presents the technical details of BPELCheck4Guide prototype tools. It contains the information about the source files and the user instructions.

Source folder

Lib Contains the helper library	Absolute-Layout.jar, swing-layout-1.0.jar	Extra layout management utilities apart from those in J2SE 5.0.
	Graphviz	The graph visualization software from AT&T for rendering graph in the program
Xml FSA definition in XML format	Patterns	XML definition of pattern properties
	Process	XML definition of business process
Src The source files of the software	Manager	The manager module responsible for globally managing the checking and guiding environment
	Gui	The graphical user interface of the tool
	Fsa	The package for internal representation of FSA and utilities and algorithms for manipulating FSA
	Checker	The checking module
	Guider	The guidance module
	Lang	The PROPOLS language

	Parser	The XML parser and JFLAP Adapter
--	--------	----------------------------------

Run application

Set up

To run the application the user need to

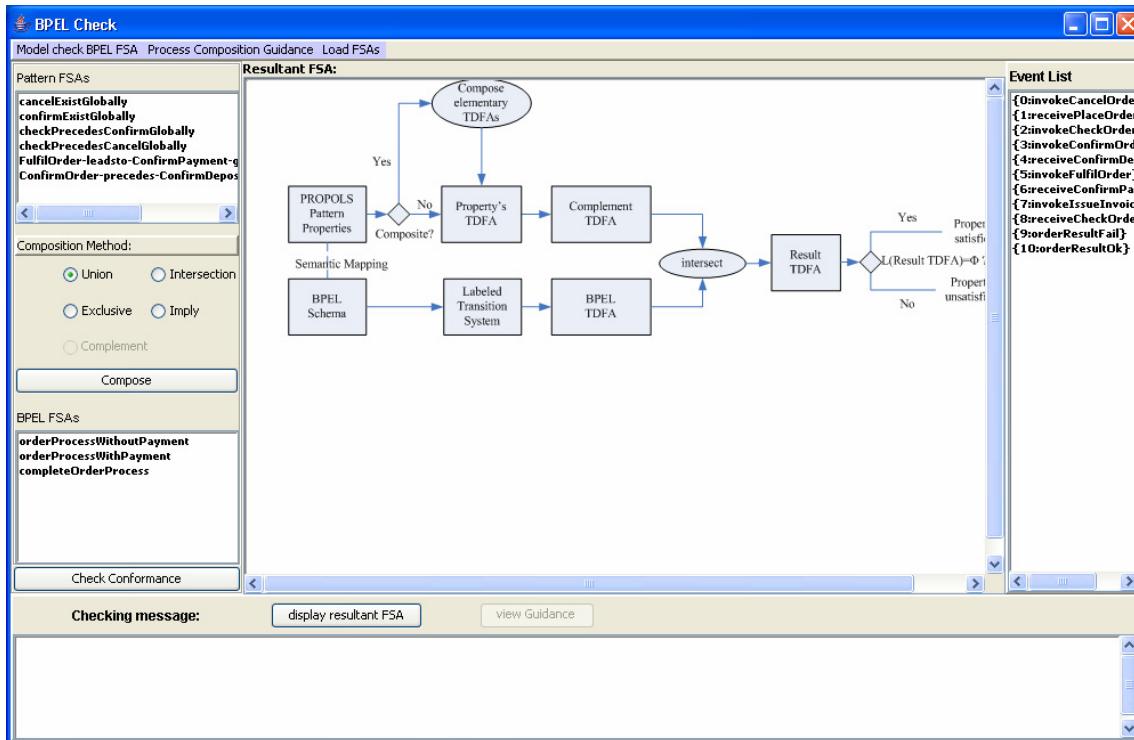
1. Install the Graphviz DOT software accompanied in the lib package
2. Edit the run.bat file to reflect your Java installation path
3. Run the run.bat file

The run.bat file will configure the necessary paths, allocate more heaps, stack space to the java process, and run the tool.

```
@echo OFF
set run="C:\Program Files\Java\jdk1.5.0_04\bin\java"
set stack=-Xss50m
set heap=-Xmx256m
set CP=-cp lib\AbsoluteLayout.jar;lib\swing-layout-1.0.jar;build\classes
%run% %stack% %aheap% %CP% gui.MainUi
PAUSE
```

When the program loads, it will load up the start up screen, as follows

Appendix 2: Prototype Tool Technical Details



Load FSA definitions

The program loads with the accompanied predefined patterns and process FSA. To load your own FSA definitions use the “Load FSA” menu. The tool has a JFLAP adapter which can read the output of JFLAP.

FSA display

All the loaded FSA are displayed in the left panel in two lists, pattern FSA list and process FSA list. When user clicks on any FSA, its graphical representation are displayed in the middle panel.

Compose FSA

To compose FSA, select two FSA from the Pattern FSA list and select the appropriate composition method radio button then click the “Compose” button. The resultant FSA will be displayed in the graph panel and will be automatically added to the pattern FSA list.

Appendix 2: Prototype Tool Technical Details

Check conformance

To check conformance, select a pattern FSA and a process FSA and click “*Check Conformance*” button. The resultant FSA graph will be displayed together with a popup showing the result of checking (conforming or not conforming).

View guidance

If the result of the checking is negative, user can press “*View Guidance*” button to view guidance. A dialog box will popup showing the compact view of guidance. If the user wants to view more details he can select the “*View Details*” option which will show the BPEL process FSA with the error paths highlighted as well.

Appendix 3: Sample XML Definitions of Rules and BPEL schemas

This appendix presents the XML definition of the rules and BPEL FSA. Without the PROPOLS parser and the BPEL2LTS library, the prototype tool can work if inputs are given in this format. The tool also has a JFLAP adapter that allows users to edit input graphically using the formal languages editor software JFLAP.

```
<?xml version="1.0" encoding="UTF-8"?>

<FSA>
  <FSA ID="rejectExistGlobally">
    <state name ="Q0" isFinal="false">0</state>
    <state name ="Q1" isFinal="true">1</state>
    <transition from="0" to="0" >
      <event>receivePlaceOrder</event>
      <event>invokeCheckOrder</event>
      <event>invokeConfirmOrder</event>
      <event>receiveConfirmDeposit</event>
      <event>invokeFulfilOrder</event>
      <event>receiveConfirmPayment</event>
      <event>invokeIssueInvoice</event>
    </transition>
    <transition from="0" to="1" >
      <event>invokeRejectOrder</event>
    </transition>
    <transition from="1" to="1" >
      <event>receivePlaceOrder</event>
      <event>invokeCheckOrder</event>
      <event>invokeRejectOrder</event>
      <event>invokeConfirmOrder</event>
      <event>receiveConfirmDeposit</event>
      <event>invokeFulfilOrder</event>
      <event>receiveConfirmPayment</event>
    </transition>
  </FSA>
</FSA>
```

Appendix 3: Sample XML Definitions of Rules and BPEL Schemas

```
        <event>invokeIssueInvoice</event>
    </transition>
    <guidanceMessage> Insert invokeRejectOrder in every error path
</guidanceMessage>
</FSA>

<FSA ID="confirmExistGlobally">
    <state name ="Q0" isFinal="false">0</state>
    <state name ="Q1" isFinal="true">1</state>
    <transition from="0" to="0" >
        <event>receivePlaceOrder</event>
        <event>invokeCheckOrder</event>
        <event>invokeRejectOrder</event>
        <event>receiveConfirmDeposit</event>
        <event>invokeFulfilOrder</event>
        <event>receiveConfirmPayment</event>
        <event>invokeIssueInvoice</event>
    </transition>
    <transition from="0" to="1" >
        <event>invokeConfirmOrder</event>
    </transition>
    <transition from="1" to="1" >
        <event>receivePlaceOrder</event>
        <event>invokeCheckOrder</event>
        <event>invokeRejectOrder</event>
        <event>invokeConfirmOrder</event>
        <event>receiveConfirmDeposit</event>
        <event>invokeFulfilOrder</event>
        <event>receiveConfirmPayment</event>
        <event>invokeIssueInvoice</event>
    </transition>
</FSA>

<FSA ID="checkPrecedesConfirmGlobally">
    <state name ="Q0" isFinal="true">0</state>
    <state name ="Q1" isFinal="true">1</state>
    <transition from="0" to="0" >
        <event>receivePlaceOrder</event>
```

Appendix 3: Sample XML Definitions of Rules and BPEL Schemas

```
        <event>invokeRejectOrder</event>
        <event>receiveConfirmDeposit</event>
        <event>invokeFulfilOrder</event>
        <event>receiveConfirmPayment</event>
        <event>invokeIssueInvoice</event>
    </transition>
<transition from="0" to="1" >
    <event>invokeCheckOrder</event>
</transition>
<transition from="1" to="1" >
    <event>receivePlaceOrder</event>
    <event>invokeCheckOrder</event>
    <event>invokeRejectOrder</event>
    <event>invokeConfirmOrder</event>
    <event>receiveConfirmDeposit</event>
    <event>invokeFulfilOrder</event>
    <event>receiveConfirmPayment</event>
    <event>invokeIssueInvoice</event>
</transition>
</FSA>

<FSA ID="checkPrecedesRejectGlobally">
    <state name ="Q0" isFinal="true">0</state>
    <state name ="Q1" isFinal="true">1</state>
    <transition from="0" to="0" >
        <event>receivePlaceOrder</event>
        <event>invokeConfirmOrder</event>
        <event>receiveConfirmDeposit</event>
        <event>invokeFulfilOrder</event>
        <event>receiveConfirmPayment</event>
        <event>invokeIssueInvoice</event>
    </transition>
    <transition from="0" to="1" >
        <event>invokeCheckOrder</event>
    </transition>
    <transition from="1" to="1" >
        <event>receivePlaceOrder</event>
        <event>invokeCheckOrder</event>
```

Appendix 3: Sample XML Definitions of Rules and BPEL Schemas

```
<event>invokeRejectOrder</event>
<event>invokeConfirmOrder</event>
<event>receiveConfirmDeposit</event>
<event>invokeFulfilOrder</event>
<event>receiveConfirmPayment</event>
<event>invokeIssueInvoice</event>
</transition>
</FSA>

<FSA ID="FulfilOrder-leadsto-ConfirmPayment-globally">
  <state name ="Q0" isFinal="true">0</state>
  <state name ="Q1" isFinal="false">1</state>
  <transition from="0" to="0" >
    <event>receivePlaceOrder</event>
    <event>invokeRejectOrder</event>
    <event>invokeConfirmOrder</event>
    <event>receiveConfirmPayment</event>
    <event>receiveConfirmDeposit</event>
    <event>invokeCheckOrder</event>
    <event>invokeIssueInvoice</event>
  </transition>
  <transition from="0" to="1" >
    <event>invokeFulfilOrder</event>
  </transition>
  <transition from="1" to="1" >
    <event>receivePlaceOrder</event>
    <event>invokeCheckOrder</event>
    <event>invokeRejectOrder</event>
    <event>invokeConfirmOrder</event>
    <event>receiveConfirmDeposit</event>
    <event>invokeCheckOrder</event>
    <event>invokeIssueInvoice</event>
  </transition>
  <transition from="1" to="0" >
    <event>receiveConfirmPayment</event>
  </transition>
</FSA>
```

Appendix 3: Sample XML Definitions of Rules and BPEL Schemas

```
<FSA ID="ConfirmOrder-precedes-ConfirmDeposit-before-ConfirmPayment">
  <state name ="Q0" isFinal="true">0</state>
  <state name ="Q1" isFinal="true">1</state>
  <state name ="Q2" isFinal="true">2</state>
  <transition from="0" to="0" >
    <event>receivePlaceOrder</event>
    <event>invokeRejectOrder</event>
    <event>invokeFulfilOrder</event>
    <event>invokeCheckOrder</event>
    <event>invokeIssueInvoice</event>
  </transition>
  <transition from="0" to="1" >
    <event>receiveConfirmDeposit</event>
  </transition>
  <transition from="1" to="1" >
    <event>receivePlaceOrder</event>
    <event>invokeCheckOrder</event>
    <event>invokeRejectOrder</event>
    <event>invokeConfirmOrder</event>
    <event>receiveConfirmDeposit</event>
    <event>invokeCheckOrder</event>
    <event>invokeFulfilOrder</event>
    <event>invokeIssueInvoice</event>
  </transition>
  <transition from="0" to="2" >
    <event>invokeConfirmOrder</event>
    <event>receiveConfirmPayment</event>
  </transition>
  <transition from="2" to="2" >
    <event>receivePlaceOrder</event>
    <event>invokeCheckOrder</event>
    <event>invokeRejectOrder</event>
    <event>invokeConfirmOrder</event>
    <event>receiveConfirmDeposit</event>
    <event>invokeCheckOrder</event>
    <event>receiveConfirmPayment</event>
    <event>invokeFulfilOrder</event>
  </transition>
</FSA>
```

Appendix 3: Sample XML Definitions of Rules and BPEL Schemas

```
        <event>receiveConfirmPayment</event>
        <event>invokeIssueInvoice</event>
    </transition>
</FSA>
</FSA>
```

Appendix 4: The Example WSDL File

Appendix 4 presents the WSDL file for the web services in the example.

```
The WSDL file
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by rth77
(rth77) -->
<definitions xmlns:tns="http://example.com"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:ns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.com" name="OnlineShop">
  <types>
    <schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://www.advantwise.com"
xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="OrderType">
        <complexType>
          <sequence>
            <element name="orderId" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="OrderResponseType">
        <complexType>
          <sequence>
            <element name="result" type="string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="Order">
    <part name="payload" element="" type="tns:OrderType"/>
  </message>
  <message name="OrderResponse">
```

Appendix 4: The Example WSDL file

```
        <part name="payload" element=""
type="tns:OrderResponseType"/>
    </message>
    <message name="CheckOrderResponse">
        <part name="payload" element="" type="ns:boolean"/>
    </message>
    <message name="ConfirmDepositNotif">
        <part name="payload" element="" type="ns:boolean"/>
    </message>
    <message name="FulfilOrderResponse">
        <part name="payload" element="" type="ns:boolean"/>
    </message>
    <message name="ConfirmPaymentNotif">
        <part name="payload" element="" type="ns:boolean"/>
    </message>
    <portType name="Shop">
        <operation name="placeOrder">
            <input message="tns:Order"/>
        </operation>
        <operation name="checkOrder">
            <input message="tns:Order"/>
            <output message="tns:CheckOrderResponse"/>
        </operation>
        <operation name="confirmDeposit">
            <input message="tns:ConfirmDepositNotif"/>
        </operation>
        <operation name="confirmPayment">
            <input message="tns:ConfirmPaymentNotif"/>
        </operation>
    </portType>
    <portType name="Customer">
        <operation name="confirmOrder">
            <input message="tns:Order"/>
        </operation>
        <operation name="rejectOrder">
            <input message="tns:Order"/>
        </operation>
    </portType>
</wsdl:binding>
</wsdl:service>
</wsdl:definitions>
```

Appendix 4: The Example WSDL file

```
<operation name="issueInvoice">
    <input message="tns:Order"/>
</operation>
</portType>

<portType name="Manufacturer">
    <operation name="filfulOrder">
        <input message="tns:Order"/>
    </operation>
</portType>
<portType name="Bank">
</portType>

<!--
~~~~~
TYPE DEFINITION - List of services participating in this BPEL
process
The default output of the BPEL designer uses strings as input and
output to the BPEL Process. But you can define or import any XML
Schema type and us them as part of the message types.
~~~~~
->
<!--
~~~~~
MESSAGE TYPE DEFINITION - Definition of the message types used as
part of the port type defintions
~~~~~
->
<!--
~~~~~
PORT TYPE DEFINITION - A port type groups a set of operations into
a logical service unit.
~~~~~
->
<!-- portType implemented by the ProcessOrder BPEL process -->
```

Appendix 4: The Example WSDL file

```
<!-- portType implemented by the requester of ProcessOrder BPEL
process
    for asynchronous callback purposes
    -->
<!--
~~~~~
PARTNER LINK TYPE DEFINITION
    the ProcessOrder partnerLinkType binds the provider and
    requester portType into an asynchronous conversation.
~~~~~ -
->
<plnk:partnerLinkType name="OrderProcessing">
    <plnk:role name="Requester">
        <plnk:portType name="tns:Customer"/>
    </plnk:role>
    <plnk:role name="Provider">
        <plnk:portType name="tns:Shop"/>
    </plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="GoodsSupply">
    <plnk:role name="Requester">
        <plnk:portType name="tns:Shop"/>
    </plnk:role>
    <plnk:role name="Provider">
        <plnk:portType name="tns:Manufacturer"/>
    </plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="FinanceService">
    <plnk:role name="Requester">
        <plnk:portType name="tns:Shop"/>
    </plnk:role>
    <plnk:role name="Provider">
        <plnk:portType name="tns:Bank"/>
    </plnk:role>
</plnk:partnerLinkType>
</definitions>
```

Appendix 5: The example BPEL schema

Appendix 5 presents the example BPEL schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveBPEL(tm) Designer Version 2.0.0 (http://www.active-
endpoints.com)
-->
<process name="ProcessOrder" suppressJoinFailure="yes"
  targetNamespace="http://example.com"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpelx="http://schemas.oracle.com/bpel/extension"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:ora="http://schemas.oracle.com/xpath/extension"
  xmlns:tns="http://example.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <partnerLinks>
    <partnerLink myRole="Provider" name="customer"
partnerLinkType="tns:OrderProcessing" partnerRole="Requester"/>
    <partnerLink myRole="Requester" name="manufacturer"
partnerLinkType="tns:GoodsSupply" partnerRole="Provider"/>
    <partnerLink myRole="Requester" name="me"
partnerLinkType="tns:OrderProcessing" partnerRole="Provider"/>
    <partnerLink myRole="Requester" name="bank"
partnerLinkType="tns:FinanceService" partnerRole="Provider"/>
  </partnerLinks>
  <variables>
    <variable messageType="tns:Order" name="inputOrder"/>
    <variable messageType="tns:CheckOrderResponse"
name="checkOrderResponse"/>
    <variable messageType="tns:ConfirmDepositNotif"
name="confirmDepositNotif"/>
    <variable messageType="tns:FulfilOrderResponse"
name="fulfilOrderResponse"/>
  </variables>
</process>
```

Appendix 5: The example BPEL schema

```
<variable messageType="tns:ConfirmPaymentNotif"
name="confirmPaymentNotif"/>
</variables>
<sequence name="main">
  <receive createInstance="yes" name="receiveOrder"
operation="placeOrder"
  partnerLink="customer" portType="tns:Shop"
variable="inputOrder"/>
  <invoke inputVariable="inputOrder" name="checkOrder"
operation="checkOrder"
  outputVariable="checkOrderResponse" partnerLink="me"
portType="tns:Shop"/>
  <switch name="IsOrderValid">
    <case
condition="bpws:getVariableData ("checkOrderResponse")">
      <sequence name="validOrder">
        <invoke name="invokeConfirmOrder"
operation="confirmOrder" partnerLink="customer" portType="tns:Customer"
inputVariable="inputOrder"/>
        <receive name="receiveConfirmDeposit"
operation="confirmDeposit" partnerLink="bank" portType="tns:Bank"
outputVariable="confirmDepositNotif"/>
        <invoke name="invokeFulfilOrder"
inputVariable="inputOrder" operation="fulfilOrder"
outputVariable="fulfilOrderResponse" partnerLink="manufacturer"
portType="tns:Manufacturer"/>
        <invoke name="invokeIssueInvoice"
operation="issueInvoice" inputVariable="inputOrder"
partnerLink="customer" portType="tns:Customer"/>
        <receive name="receiveConfirmPayment"
operation="confirmPayment" partnerLink="bank" portType="tns:Bank"
inputVariable="confirmPaymentNotif"/>
      </sequence>
    </case>
    <otherwise>
      <invoke inputVariable="inputOrder" name="invokeRejectOrder"
operation="rejectOrder" partnerLink="customer"
portType="tns:Customer"/>
    </otherwise>
  </switch>
</sequence>
```

Appendix 5: The example BPEL schema

```
        </otherwise>  
    </switch>  
</sequence>  
</process>
```

References

1. ActiveBPEL Designer. <http://www.active-endpoints.com/products/activebpeldes/>
2. G. Alonso, F. Casati, Grigori, H. Kuno, V. Machiraju. Web Services Concepts, Architectures and Applications. 2004, Springer-Verlag
3. T. Andrews, F. Curbera, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana,. Business Process Execution Language for Web Services version 1.1. 2003, <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>
4. R. Anzböck and S. Dustdar. Semi-automatic generation of Web services and BPEL processes - A model-driven approach. Proc. 3rd Int'l Conference on Business Process Management (BPM), LNCS 3649, pp. 64-79, 2005, Springer
5. B. Obrriens, J. Yang, M. Papazoglou. Model driven service composition. Proc. Service Oriented Computing (ICSOC), LNCS 2910, pp. 75-90, 2003, Springer-Verlag
6. BPMI. Business Process Modeling Language. 2002, <http://www.bpmi.org/>
7. B. Srivastava, J. Koehler. WS Composition current solution and open problems. 2003, IBM Developer Network, <http://www.zurich.ibm.com/pdf/ebizz/icaps-ws.pdf>
8. E.M. Clarke, O. Grumberg, and D. Peled. Model-checking. 1999, MIT Press
9. DAML.org. OWL-S: Semantics Markup for Web Services. <http://www.daml.org/services/owl-s/1.0/owl-s.pdf>, 2006
10. M.B. Dwyer, G.S. Avrunin, J.C. Corbett. Property Specification Patterns for Finite-State Verification. Proc. Workshop on Formal Methods in Software Practice, pp. 7 - 15, 1998, Clearwater Beach, FL, USA
11. M.B. Dwyer, G.S. Avrunin, J.C. Corbett. Patterns in Property Specifications for Finite state Verification. In Int. Conference on Software Engineering, pp. 411-420, 1999, Los Angeles, CA, USA
12. M.B. Dwyer, G.S. Avrunin, J.C. Corbett. A System of Specification Patterns. <http://www.cis.ksu.edu/santos/spec-patterns>,

Appendix 5: The example BPEL schema

13. F. Wang. Dynamic matching and binding mechanism for business service integration. In Int. Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS), 2002, Beijing China
14. H. Foster. LTSA WS-Engineering. 2006, <http://www.doc.ic.ac.uk/ltsa/BPEL/>
15. H. Foster. A Rigorous Approach to Engineering Web Services Compositions. PhD thesis, Imperial College London. 2006, <http://www.doc.ict.ac.uk/~hfl>
16. X. Fu, T. Bultan, J. Su. Analysis of Interacting BPEL Web Services. In 13th World Wide Web Conference, pp. 621-630, 2004, New York, USA
17. Graphviz. Graph visualization software. <http://www.graphviz.org>
18. J. Han, Y. Han, Y. Jin, J. Wang, and J. Yu. Personalized active service spaces for end-user service composition. Accepted for Int'l Conference on Service Computing (SCC), 2006, Chicago, USA.
19. Y. Han, H Geng, H. Li, J. Xiong, G. Li, B. Holtkamp, R. Gartmann, R. Wagner, N. Weissenberg. VINCA – A Visual and personalized Business level composition language for chaining web based services. In Int. Conference on Service Oriented Computing (ICSOC), 2003, Trento, Italy
20. L. Hao, Y. Han, G. Li, C. Zhang, Achieving context sensitivity of Service-Oriented applications with the Business-end programming language VINCA. In Grid and Cooperative Computing Int. Conference (GCC), 2004, Wuhan, China
21. Yan Jin and Jun Han. Runtime Validation of Behavioural Contracts for Component Software. In Proceedings of the 5th International Conference on Quality Software ([QSIC'05](#)), pp. 177-184, Melbourne, Australia, 19-20 September 2005, IEEE Computer Society Press
22. Y. Jin, J. Yu, M. T. Phan, J. Han. Using temporal business rules to guide service composition. Submitted to Service Oriented Computing (ICSOC), 2006, Chicago, USA
23. Z. Li, J. Han, Y. Jin. Pattern-Based Specification and Validation of Web Services Interaction Properties. Proc Int. Conference on Service Oriented Computing (ICSOC), 2005, Amsterdam, the Netherlands

Appendix 5: The example BPEL schema

24. M. Lin, H. Guo, J. Yin. Goal description languages for Semantic Web Service Automatic Composition. In Int. Sym. on Applications and the Internet (SAINT), 2005, Trento, Italy
25. K. Mantell. From UML to BPEL. IBM developerworks. September 2003, <http://www-106.ibm.com/developerworks/webservices/library/wsuml2bpel/>
26. OntoViz Tab. Visualizing Protégé Ontologies. 2005, <http://protege.stanford.edu/plugins/ontoviz/ontoviz.html>
27. C. Peltz. Web services orchestration, a review of emerging technologies, tools, and standards. 2003, HP White paper, http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf
28. M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. Proc. Int'l Joint Conference on Artificial Intelligence (IJCAI), pp. 1252-1259, 2005.
29. R. L. Smith, G.S. Avrunin, L.A. Clarke, L.J. Osterweil. PROPEL: An Approach Supporting Property Elucidation. In 24th International Conference on Software Engineering, pp. 11-21, 2002, Orlando, FL, USA
30. M. P. Singh, M. N. Huhns. Service-Oriented Computing, semantics, processes, agents. 2005, John Wiley & Sons, Ltd
31. C.A. Stahl. Petri Net Semantics for BPEL. Informatik-Berichte pp. 188, Humboldt-Universität zu Berlin, June, 2005
32. Sun Microsystems. Service Oriented Architecture. http://www.sun.com/soa/registry/faq/image1_lg.gif, 2006
33. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. Proc. Int'l Semantic Web Conference (ISWC), pp. 380-394, 2004.
34. W3C. Web Service Semantics - WSDL-S. www.w3.org/Submission/WSDL-S, 2005
35. J. Yu, M.T. Phan, J. Han, and Y. Jin. Pattern based property specification and verification for service composition, Submitted to Int. Conference on Web Information System Engineering (WISE), 2006, Wuhan, China

Appendix 5: The example BPEL schema

36. J. Yu, M.T. Phan, J. Han, and Y. Jin. Pattern based property specification and verification for service composition. Technical Report SUT.CeCSES-TR010, CeCSES, Swinburne University of Technology, May 2006, <http://www.it.swin.edu.au/centres/cecses/trs.htm>.