

FORMAL ANALYSIS OF WEB SERVICE COMPOSITION

CHEN MANMAN

(B.Sc., Nanjing University, 2011)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2015

Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in black ink, consisting of the Chinese characters '陈曼曼' (Chen Manman) in a cursive style.

CHEN MANMAN
29 September 2015

Acknowledgements

It would not be possible to complete my thesis without the encouragement and help of people around me, who give me valuable instructions and assistance during the whole of my Ph.D. journey.

I would like to express my special appreciation and thanks to my supervisor, Professor Dong Jin Song, for his supporting, advice and encouragement during these past four years. Professor Dong is someone you will instantly love and never forget once you meet him. His advice on both research as well as on my career have been invaluable.

I am also very grateful to to my mentors Dr. Sun Jun, Dr. Liu Yang and Dr. Tan Tian Huat, who act like friends and co-supervisors in the past four years. I thank them for their advice and knowledge and many insightful discussions and suggestions that I will benefit for my whole life. I would also like to thank my committee members, Dr Colin Tan and Dr Bimlesh Wadhwa for serving as my committee members and many helpful suggestions.

I would like to thank my seniors Dr. Shi Ling, Dr. Liu Yan, Dr. Song Songzheng, Dr. Gui Lin, Dr. Nguyen Truong Khanh, Dr. Liu Shuang and my fellow students Dr. Bai Guang Dong, Li Li for for your support and friendships through my Ph.D. study. And I am grateful to all my colleagues and friends in PAT group for their support and encouragement throughout, some of whom have already been named.

Lastly, I wish to express my deepest thanks to my parents for their love and support to help me go through all the difficulties. Their love is the greatest power in my life. Without them, I can not make it this far.

Contents

List of Tables	xi
List of Figures	xiii
List of Algorithms	xvii
1 Introduction	1
1.1 Thesis Overview	2
1.2 Thesis Outline	6
1.3 Publications of Our Works	7
2 Background	9
2.1 Service Oriented Architecture	9
2.2 Web Service Composition	11
2.3 Model Checking	12
3 Automated Synthesis of Local Time Requirement for Service composition	15
3.1 A BPEL Example with Timed Requirements	18
3.2 Overall Approach	20
3.3 Runtime Refinement of Local Time Requirement	22
3.3.1 Motivation	22

3.3.2	Runtime Adaptation of a BPEL Process	23
3.3.3	Algorithm for Runtime Refinement	24
3.3.4	Satisfiability Checking	26
3.3.5	Termination and Soundness	27
3.3.6	Discussion	29
3.4	Evaluation	30
3.4.1	Case Studies	31
3.4.2	Synthesis of Local Time Requirement	33
3.4.3	Runtime Adaptation	34
3.4.4	Threat to Validity	38
3.5	Related Work	39
3.6	Chapter Summary	41
4	Dynamic Ranking Optimization for QoS-Aware Service Composition	43
4.1	QoS-Aware Compositional Model	46
4.1.1	QoS Attributes	47
4.1.2	QoS for Composite Services	48
4.1.3	Optimality Function	50
4.1.4	Problem Statement	52
4.2	Dynamic Ranking Optimization	53
4.2.1	Service Preprocessing	54
4.2.2	Service Ranking	57
4.2.3	Dynamic Service Selection	60
4.2.4	Solving for Optimal Selection	64
4.3	Evaluation	64
4.4	Evaluation	65

4.4.1	Evaluation with a Synthetic Dataset	69
4.4.2	Evaluation with QWS Dataset	73
4.5	Related Work	74
4.6	Chapter Summary	76
5	Optimizing Selection of Competing Features via Feedback-directed Evolutionary Algorithms	77
5.1	Background	80
5.1.1	Software Product Line	80
5.1.2	Feature Model and its Semantics	80
5.1.3	Multi-objective Optimization Problem	83
5.2	Feedback-directed Evolutionary Algorithm	84
5.2.1	Preliminaries of Evolutionary Algorithms	85
5.2.2	Preprocessing of Feature Model	86
5.2.3	Genetic Encoding of the Feature Set	87
5.2.4	Feedback-directed Evolutionary Operators	88
5.3	Related Work	92
5.4	Conclusion	95
6	Verification of Functional and Non-functional Requirements of Web Service Composition	97
6.1	Motivation Example	100
6.1.1	Computer Purchasing Services (CPS)	100
6.1.2	BPEL Notations	102
6.2	QOS-AWARE COMPOSITIONAL MODEL	102
6.2.1	QoS Attributes	103
6.2.2	QoS for Composite Services	104

6.2.3	Labeled Transition System	105
6.3	Verification	108
6.3.1	Verification of Functional Requirement	108
6.3.2	Integration of Non-Functional Requirement	109
6.3.3	Integration of Availability and Cost	109
6.3.4	Integration of Response Time	110
6.3.5	Discussion	114
6.4	Experiment	114
6.4.1	Computer Purchasing Service (CPS)	114
6.4.2	Loan Service (LS)	115
6.4.3	Travel Agency Service (TAS)	116
6.5	Related Work	117
6.5.1	Verification of Web Service Composition	117
6.5.2	Constraint Synthesis of Web Service Composition	120
6.6	Chapter Summary	121
7	Tool Implementation: VeriWS	123
7.1	VeriWS	125
7.1.1	Architecture and Implementation	125
7.1.2	Aggregator	127
7.1.3	Verifier	128
7.1.4	Simulator	129
7.1.5	Comparison with Existing Tools	129
7.2	Demonstration	131
7.2.1	Computer Purchasing Service (CPS)	131
7.2.2	Requirements for Verification	132
7.3	Chapter Summary	133

8	Automated Runtime Recovery for QoS-based Service Composition	135
8.1	Motivating Example	138
8.2	QoS-aware Compositional Model	140
8.2.1	Labeled Transition System	141
8.2.2	Example: Transport Booking Service	142
8.2.3	Backward Actions	143
8.2.4	Monitoring Automata	144
8.2.5	Recovery Plan	145
8.3	Service Recovery as a GA Problem	147
8.3.1	Preliminaries of Genetic Algorithms	147
8.3.2	Architecture	149
8.3.3	Genetic Encoding of a Recovery Plan	149
8.3.4	Genetic Operators	151
8.3.5	Calculating the Fitness Value	153
8.3.6	QoS Optimality	153
8.3.7	Global Optimality	156
8.3.8	Fitness Function	157
8.3.9	Enhanced Initial Population Policy	158
8.3.10	rGA Algorithm	161
8.4	Evaluation	162
8.5	Related Work	167
8.6	Chapter Summary	169
9	Conclusion and Future Works	171
9.1	Conclusion	171
9.2	Future Work	172

Bibliography	175
A Appendix of Chapter 3	187
A.1 A Formal Model for Parametric Composite Services	187
A.1.1 Variables, Clocks, Parameters, and Constraints	188
A.1.2 Syntax of Composite Services	189
A.1.3 Parametric Composite Services	190
A.1.4 Bad Activity	191
A.2 A Formal Semantics for Parametric Composite Services	192
A.2.1 Labeled Transition Systems	192
A.2.2 Symbolic States	193
A.2.3 Implicit Clocks	193
A.2.4 Operational Semantics	194
A.2.5 Application to an Example	197
A.3 Synthesis of Design-time LTC	199
A.3.1 Addressing the Good States	199
A.3.2 Addressing the Bad States	200
A.3.3 Synthesis Algorithms	201
A.3.4 Application to the Running Example	202
A.3.5 Service Selection	203
A.3.6 Termination and Soundness	203
A.3.7 Incompleteness of dLTC	206
B Appendix of Chapter 5	209
B.1 Evaluation for Optimizing Selection of Competing Features	209
B.1.1 Setup	209
B.1.2 Evaluation with SPLOT	215

B.1.3	Evaluation with LVAT	217
B.1.4	Threats to validity	218

Summary

Web service technologies have emerged as a de-facto standard for integrating disparate applications and systems using open, XML-based standards. In addition to building Web service interfaces to existing applications, a number of standards (e.g. WS-BPEL) have been proposed to compose these Web services together to form a more meaningful business processes. In this thesis, we focus on the verification and analysis of the composition of Web services.

We present a fully automated technique for the synthesis of the local time requirement to help the service composition conform to the time requirement. The approach is implementation independent, therefore can be applied at the design stage of service composition. Based on the synthesis requirements, we propose a new approach to select a set of component services to compose a composite service such that it could satisfy the non-functional requirements, and we also extend the work to find a set of features that do not have inconsistency or conflict, yet optimize multiple objectives (e.g., minimizing cost and maximizing number of features), for service-based product lines. To guarantee the requirements of the service composition at the design time, we propose a method to verify the service composition against combined functional and non-functional requirements. We capture the semantics of Web service composition using labelled transition systems (LTSs) and verify the Web service composition directly without building intermediate or abstract models before applying verification approaches. We have also developed a tool to implement our proposed approach. To help the service composition conform to requirements during runtime, we propose an automated approach based on a genetic algorithm to generate the recovery plan. Our approach has been evaluated on real-world case studies, and has

shown promising results.

Key words: **Web Service, Web Service Composition, Model Checking, Formal Verification**

List of Tables

2.1	Standards used by Web Services	10
4.1	Aggregation Function	51
5.1	Constraints of <i>JCS</i>	83
6.1	QoS Attribute Values	103
6.2	Aggregation Function	105
6.3	Experiment Results	115
7.1	Aggregation Function	127
7.2	Web Service Verification Tools	129
B.1	Brief overview of EAs	210
B.2	Feature Models	211
B.3	Evaluation with SPLOT	214
B.4	Evaluation with LVAT	215

B.5	Improvement of EAs on SPLOT	216
B.6	Improvement of EAs on LVAT	216

List of Figures

1.1 Overall Picture of My Work	5
3.1 Stock Market Indices Service	19
3.2 Synthesis of Local Time Requirement	21
3.3 Service adaptation framework	23
3.4 Computer Purchasing Service (CPS)	30
3.5 Travel Booking Service (TBS)	31
3.6 Experiment results on synthesis	33
3.7 Synthesized dLTC	34
3.8 Overhead of runtime monitoring	35
3.9 Improvement of runtime conformance	36
4.1 Travel Agency Service (TAS)	44
4.2 Compositional structure	48
4.3 TAS Example	52

4.4	Dynamic Ranking Optimization (DRO)	55
4.5	A scenario of dynamic service selection	58
4.6	Experiment results for synthetic dataset	65
4.7	Worst-case performance for synthetic dataset	66
4.8	Attributes for synthetic/QWS dataset	66
4.9	Preprocess results for synthetic dataset	67
4.10	Preprocess results for QWS dataset	67
4.11	Experiment results for QWS dataset	68
5.1	The feature model of <i>JCS</i>	83
5.2	Typical flow of evolutionary algorithms	85
5.3	Feedback-directed mutation operator	90
5.4	Feedback-directed crossover operator	90
6.1	Computer Purchasing Service	101
6.2	LTS of CPS	107
6.3	LTS of CPS with Availability and Cost	110
6.4	LTS of CPS with Response Time, Availability and Cost	111
7.1	VeriWS Architecture	126
7.2	WS-BPEL Description for CPS	131

8.1	Transport Booking Service (TBS)	139
8.2	LTS of TBS example	142
8.3	Monitoring Automata	145
8.4	Typical Flow of Genetic Algorithms	147
8.5	Service Monitoring and Recovery Framework	148
8.6	Genetic Encoding of Recovery Plan	150
8.7	Genetic Operations	152
8.8	Convergence Rate	163
8.9	Experiment with rGA	164
A.1	Syntax of composite service processes	190
A.2	Activation function	194
A.3	Idling function	195
A.4	Set of rules for the transition relation \hookrightarrow	196
A.5	LTS of service CS	197
A.6	LTS of composite service CS	200
A.7	LTS of composite service CS'	200
A.8	LTS of the SMIS	202
A.9	dLTC of SMIS	203
A.10	dLTC of SMIS after simplification	203

List of Algorithms

1	Algorithm $rLTC(CS, s)$	25
2	Algorithm $\text{CheckSat}(\mathcal{LCS}, s_a, r, T_{CS})$	26
3	Service Preprocessing (DROPreprocess)	57
4	Optimization with DRO	60
5	Dynamic service selection (DROSelect)	61
6	PrunableFeatures	87
7	FMutation	91
8	ErrPos	92
9	FCrossover	93
10	Algorithm $\text{TagTime}(P, x)$	112
11	Algorithm $\text{CalculateTime}(P)$	113
12	Crossover	153
13	Mutation	153

14	Fitness	155
15	Initial Population	159
16	GA Algorithm	161
17	Algorithm <i>LocalTimeConstraint(CS)</i>	201
18	Algorithm <i>synConsLTS(s)</i>	201

Chapter 1

Introduction

Service Oriented Architecture (SOA) has been an important software design architecture that aims to enhance the efficiency, agility and productivity of an enterprise. This allows enterprises to outsource part of their processes to external services, which produces a lower cost of ownership for the enterprises over time. Services exist as physically independent software programs and each service has its own distinct functionality.

Web services make use of open standards, such as WSDL [15] and SOAP [92], that enable the interaction among heterogeneous applications. A Web service that are composed by other services is called *composite services*. Services that the composite service makes use of are called *component services*. Following the SOA, a composite service contains a set of *abstract services* (e.g., a hotel booking service) which have their interfaces defined, and *concrete services* (e.g., the Hilton Hotel booking service) are selected to realize the interfaces of abstract services during runtime. There are two kinds of requirements that are crucial to composite services, i.e., functional requirements and non-functional requirements. Functional requirements concern about the functionality of the composite service. Non-functional requirements concern about the quality of the service (QoS), and are usually

specified in a contract, called service-level agreements (SLAs).

Service composition is inevitably rich in concurrency and it is not a simple task for programmers to utilize concurrency as they have to deal with multi-threads and critical regions. Therefore, it is desirable to verify Web services with automated verification techniques at the design time, as the complexity of service composition continues to escalate.

1.1 Thesis Overview

Although during decades of enthusiasm across the research community of analyzing Web service composition, there are still some research gaps briefly listed as following:

- Given the response time requirement of a composite service, there do not exist any approaches that could synthesize the response time requirement for component services that will be used to compose the composite service.
- Non-functional requirements are crucial for Web service composition, and they often become clauses of service level agreement (SLA) among the service providers and users. Therefore, it is important to choose a set of component Web services that can maximize the overall Quality of Service (QoS) of Web service composition and, at the same time, satisfy all the compositional level constraints specified in the SLA.
- Both functional and non-functional requirements are important to the Web service composition. However, existing works cannot support verification of combined functional and non-functional requirements. They usually are just focused on one aspect. And combined functional and non-functional requirements can be used to check more complicated properties, e.g., the system can always reply to users within 5 seconds.

- The composite service operates in a highly dynamic environment; hence, it can fail at any time due to the failure of component services. Existing works are required to explore all state space to generate recovery plans and the generated recovery plans cannot guarantee the QoS.

However, it is highly non-trivial to solve the problem due to the following challenges.

- Web service composition contains complex timing constructs and control flow structures such as concurrency. Such a combination of timing constructs, concurrent calls to external services, and complex control structures makes it a challenge to synthesize the local time requirement.
- Given a composite service with 10 abstract services executed in a sequential manner, with each abstract service having 10 concrete service candidates, there are 10^{10} combinations to explore. In fact, it has been shown that the problem is NP-hard [33]. Therefore, it is impossible in practice to exhaustively search through all possible combinations of concrete services.
- There are many kinds of non-functional requirements of a Web service composition, and different non-functional requirements might have different aggregation functions for different compositional structures.
- Web service composition supports compensation mechanism. One of the problems is that it is uncertain whether the compensation will lead to a system state that could satisfy the functional properties of the composite service.

In this thesis, we address these challenges on analyzing Web service composition. The contributions can be summarized as follows:

- Given a composite service, we develop a refinement procedure for the design-time synthesized local time requirement based on runtime information. In addition, we

develop a fully automated tool to evaluate the proposed method and apply it to real-world case studies.

- We propose a novel approach, called dynamic ranking optimization (DRO) to select the component services to compose the composite service. Firstly, we introduce a pruning method called constraint pruning, that could effectively discard the service candidates that cannot satisfy the global constraints. Secondly, we introduce a ranking method based on both the overall local optimality and the constraint satisfaction probability of a service.
- We capture the semantics of Web service composition using Labelled Transition Systems (LTS) (LTSs) and verify the Web service composition directly. To the best of our knowledge, we are the first work on verification of combined functional and non-functional requirements. We also develop a tool that supports verification on different kinds of combined functional and non-functional properties of Web service composition.
- We propose a new method (*rGA*) based on genetic algorithms by making use of dynamic-length chromosomes to represent the recovery plans. *rGA* does not require the generation of full state space beforehand. State space is generated on-the-fly during recovery plan exploration. And the near-optimal QoS recovery plan would be selected by *rGA*.

In the following, we will introduce the overall picture of our work.

Our work focuses on formal analysis of Web service composition as shown in Figure 1.1. Functional and non-functional requirements are two important kinds of requirements of Web service composition. Given the requirements of the Web service composition, at the design time, we synthesize the response time requirement for each component service by given the response time requirement of the composite service. We propose a new

Overview of My Works

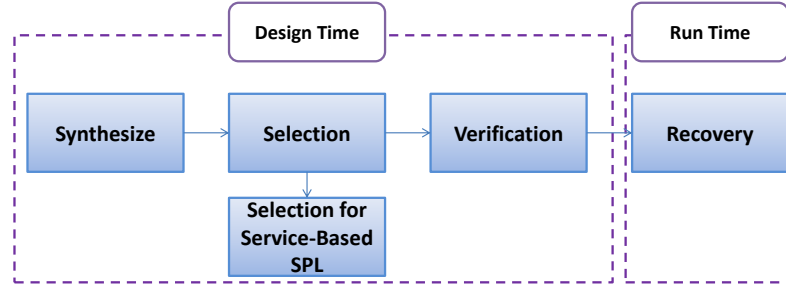


Figure 1.1: Overall Picture of My Work

technique to select a set of component services to compose a composite service such that it could satisfy the non-functional requirements. However, the two kinds of requirements are crucial to Web service composition, in order to guarantee both aspects at the same time, we check the combined functional and non-functional requirements. We extend the work on service selection to feature selection to select features for service-based product lines (SPL).

At runtime, component services could behave differently after being modified by service providers, or could fail due to various reasons such as network problems, software bugs, hardware failure, etc. Therefore, we propose an automated approach to generate the recovery plan once detecting the failure.

1.2 Thesis Outline

In this section, we briefly present the outline of thesis and the overview for each chapter.

Chapter 2 describes the background of our work. We first introduce the main features of Web service composition, including the service language and two kinds of requirements. Then we introduce the main concept of the model checking.

Chapter 3 proposes a novel technique for synthesizing design-time local time constraint (dLTC) for component services, in the form of a constraint on the local response times, that guarantees the global response time requirement. Our approach is based on the analysis of the LTS of a composite service by making use of parameterized timed techniques. Then, during the runtime of composite service, we propose the usage of the runtime information to weaken the dLTC, which becomes the runtime local time constraint (rLTC). It has been implemented and evaluated with real-world case studies.

Chapter 4 addresses the problem of QoS service composition by proposing a new technique, namely the dynamic ranking optimization (DRO). The technique considerably improves the current service selection approaches, by considering only a subset of representatives that are likely to succeed, before exploring a larger search space.

Chapter 5 proposes to incorporate a novel feedback-directed mechanism into evolutionary algorithms (EAs) to find a set of features that do not have inconsistency or conflict, yet optimize multiple objectives (e.g., minimizing cost and maximizing number of features) for service-based product lines.

Chapter 6 proposes an automated approach to verify combined functional and non-functional requirements directly based on the semantics of Web service composition. Our approach has been implemented and evaluated on the real-world case studies, which demonstrate the effectiveness of our method.

Chapter 7 presents VeriWS, a tool to verify combined functional and non-functional requirements of Web service composition. VeriWS captures the semantics of Web service composition and verifies it directly based on the semantics. We also show how to describe Web service composition and properties using VeriWS.

Chapter 8 proposes an automated approach based on a genetic algorithm to generate the recovery plan that could guarantee the satisfaction of functional properties of the composite service after recovery. Given a composite service with large state space, the proposed method does not require exploring the full state space of the composite service; therefore, it allows efficient selection of recovery plan. In addition, the selection of recovery plans is based on their quality of service (QoS). A QoS-optimal recovery plan allows effective recovery from the state of failure. Our approach has been evaluated on five real-world case studies, and has shown promising results.

Chapter 9 concludes the thesis and discusses the future work.

1.3 Publications of Our Works

Most of the work presented in this thesis has been published in international conference proceedings.

- Chapter 3 consists parts of a paper submitted to the IEEE Transaction on Software Engineering (TSE) [152]. I have made partial contribution to this submission, therefore Chapter 3 introduces the work that I have contributed to, and other related parts are listed in the Appendix A.
- Chapter 4 has been submitted to The 17th International Conference on Formal Engineering Methods (ICFEM 2015) [155].

- Chapter 5 has been accepted to International Symposium on Software Testing and Analysis (ISSTA 2015) [157]. I have made partial contribution to this publication, therefore Chapter 5 introduces the work that I have contributed to, and other related parts are listed in the Appendix B.
- Chapter 6 was published at the 15th International Conference on Formal Engineering Methods (ICFEM 2013) [54].
- Chapter 7 was published at the 36th International Conference on Software Engineering (ICSE 2014 Demo Track) [53].
- Chapter 8 was published at the 23rd International World Wide Web Conference (WWW 2014) [154].

Chapter 2

Background

2.1 Service Oriented Architecture

Enterprise applications are heterogenous in terms of operating systems, and development architecture. It is non-trivial to combine these heterogenous application to form a business process. On the other hand, existing applications are frequently tightly associated with the existing business processes; therefore, it is infeasible to build a new application starting from scratch. Service Oriented Architecture (SOA) is introduced to tackle this problem.

SOA represents a popular architectural paradigm for applications, with Web Services as probably the most visible way. It is a set of design principles for system development and integration, which establishes an architectural model to enhance the efficiency, agility, and the productivity of an enterprise.

SOA advocates an approach in which a software component provides its functionality as a service that can be leveraged by other software components. *Services* exist as physically independent software programs with distinct design characteristics. Each *service* is a piece of application's business logic or individual functions that are modularized and presented

Composition	WS-CDL,WS-BPEL
Description	WSDL
Message	SOAP
Transmission	HTTP, FTP, SMTP

Table 2.1: Standards used by Web Services

to consumer applications. The major advantage of services is their loosely coupled nature, therefore, services are suitable for invoking by external consumer programs via a published service contract (much like a traditional API).

Web service technologies are a realization of SOA based on internet protocols. It is formally defined as a software system designed to support interoperable machine-to-machine interaction over a network [10].

Web services technology is designed to offer a communication bridge between the heterogeneous computational environments. This allows organizations to communicate data without the intimate knowledge of each other's internal systems. Furthermore, since the communication between clients and servers is done through the World Wide Web, Web services could leverage on the ubiquitous internet connectivity for universal reach. To achieve this goal, Web service technology makes use of a number of protocols based on open and accepted standards as listed in Figure 2.1. For example, at the transmission level Web services take advantage of HTTP, which is supported by most Web browsers and servers. Another enabling technology is Extensible Markup Language (XML) [11]. XML is a widely accepted standard used to encode all communications to a Web service. Simple Object Access Protocol (SOAP) [13] and Web Services Description Language (WSDL) [14] are core standards used by Web services and both of them are specified in XML format. SOAP is a protocol specification for the information communication of Web services in computer networks. It relies on standard internet protocols (e.g., HTTP, SMTP). WSDL is

an XML-based interface definition language used to specify the syntax of messages that enter or leave Web services; therefore the consumer applications know the functionality offered by Web services and how to access them. For example, Paypal SOAP API [7], Salesforce SOAP API [8] are based on open standards, which include SOAP and WSDL.

However, description level only specifies the syntax of messages, the order of messages have to be exchanged between services is defined in the composition level. There are a number of WS-* specifications [6] (e.g., WS-BPEL, WS-Addressing, WS-Security, WS-Resource) proposed to handle other aspects of Web services (e.g., composition, addressing, security, resource states). In this thesis, we will focus on formal analysis of Web service composition. We will discuss Web service composition in the next section.

2.2 Web Service Composition

Composition of Web services has received much interest to support enterprise integration since it remains a challenge to integrate multiple services for complex interactions when the technology for creating services and interconnecting them with a point-to-point basis has achieved a certain degree of maturity. Web service composition makes use of existing services to form complex services in order to achieve a business goal. The de-facto standard for Web service composition is Web Services Business Process Execution Language (WS-BPEL) [102]. WS-BPEL is an XML-based orchestration business process language, which describes Web service composition by specifying the workflow of actions within business processes. It provides basic activities such as service invocation, and compositional activities such as sequential and conditional composition to describe composition of Web services.

There are two kinds of requirements of Web service composition, i.e., functional and non-functional requirements. Functional requirements focus on the functionalities of the Web

service composition, which are described as a set of inputs, the behavior and the output of the Web service composition. The non-functional requirements are concerned with the Quality of Service (QoS). They are often recorded in service-level agreements (SLAs), which is a contract specified between service providers and customers. Typical non-functional requirements include response time, availability, cost and so on. Nowadays, non-functional requirements are becoming more and more important as they have played a significant role in the user experience. Given a booking service, an example of functional requirement is that finally the booking result will be returned to users. An example of non-functional requirements is that the service will respond to the user within 3 seconds.

2.3 Model Checking

Testing, deductive reasoning, simulation, and model checking are principle techniques for formally analysis complex system behaviors. Testing approach tests system outputs with certain inputs against the expected results. Simulation approach simulates system's behaviors to compare with the expected one. However, both testing and simulation approaches are very expensive and infeasible for complex systems because most of them have various unexpected behaviors, and also not complete due to the fact that only a subset of behaviors are covered. Deductive verification is a manual approach which uses axioms and proof rules to prove the correctness of the systems. This approach can deal with infinite state systems, however, it is time consuming since it is manual method.

Model checking [58] is an automatic approach for verifying finite state systems, which exhaustively explores all possible system states. It is different from other methods in two crucial aspects, firstly, it does not aim of being fully general; secondly, it is fully algorithmic.

Basics of Model Checking Model checking is an automatic verification technique that explores all possible system states exhaustively. Given a model of a system, it exhaustively

and automatically checks whether this model meets a given specification. The specifications of the systems are specified as properties in proper logics. An example of logic specifications is temporal logic, which can assert how the behavior of the system evolves over time.

The process of model checking consists of two tasks. The first one is to abstract the model accepted by model checking tools from the original system. The second is that the verification of the system model against the specifications is generally conducted automatically by the model checker. The result will be returned with witness traces or counterexamples if the result is negative. The analysis of the error trace may require modifications to refine the model and repeat the model checking process.

Chapter 3

Automated Synthesis of Local Time Requirement for Service composition

Service-oriented architecture is a paradigm that promotes the building of software applications by using services as basic components. Services make their functionalities available through a set of operations accessible over a network infrastructure. To assemble a set of services to achieve a business goal, service composition languages such as BPEL (Business Process Execution Language) [26] have been proposed. A service that is composed by other services is called a *composite* service, and services that the composite service makes use of are called *component* services as presented in [78].

In business where timing is critical, a requirement on the service response time is often an important clause in service-level agreements (SLAs), which is the contractual basis between service consumers and service providers on the expected quality of service (QoS) level. Henceforth, we denote the response time requirement of composite services as *global time requirement*; and the set of constraints on the response times of the component services as *local time requirement*. The response time of a composite service is highly

dependent on that of individual component services. It is therefore important to derive local time requirements (i.e., requirements on the component services) from the global time requirement so as to identify component services which could be used to build the composite service while satisfying the response time requirement.

Consider an example of a stock indices service, which has an SLA with the subscribed users such that the stock indices would be returned in three seconds upon request. The stock indices service makes use of several component services, including a paid service, for requesting stock indices. The stock indices service provider would be interested in knowing the local time requirement of the component services.

BPEL is a de-facto standard for service composition. It supports control flow structures that involved complex timing constructs (e.g., `< pick >` control structure) and concurrent execution of activities (e.g., `< flow >` control structure). Such a combination of timing constructs, concurrent calls to external services, and complex control structures, makes it a challenge to synthesize the local time requirement.

In this chapter, we present a fully automated technique for the synthesis of the local time requirement in BPEL. The approach works by performing analysis on the behavior of the composite service based on its associated labeled transition systems (LTSs), using techniques on parameter synthesis for real-time systems. For the synthesized local time requirement to be useful, it needs to be as weak as possible, to avoid discarding any service candidates that might be a part of a feasible composition. This is particularly important, as often having a faster service would incur higher cost. Our synthesis approach does not only avoid bad scenarios in the service composition, but also guarantees the fulfillment of global time requirement.

In this chapter, the local time requirement of a composite service is represented as a constraint, which is called the *local time constraint* (LTC). During design time of a composite

service, since it is unknown which execution path will be executed at runtime, the LTC is synthesized based on *all* possible execution paths. The LTC of a composite service that is synthesized during the design time is called the *design-time local time constraint* (dLTC).

Due to the highly evolving and dynamic environment the composite service is running in, the design time assumptions for Web service composition, even if they are initially accurate, may later change at runtime. For example, the execution time of a component service could violate the dLTC due to reasons such as network congestion. Nevertheless, this does not necessarily imply that the composite service will not satisfy the global time requirement. Indeed, the dLTC is synthesized based on all possible execution paths at design-time, whereas only one path will be executed at runtime. At runtime, some of the execution paths can be eliminated. Therefore, we can use the runtime information to refine the dLTC to make it weaker – which results in a more relaxed constraint on the response times of the component services. We call the dLTC after runtime refinement a *runtime local time constraint* (rLTC). The rLTC is then used to decide whether the current composite service can still satisfy the global time requirement.

My contributions to this chapter are as follows.

1. We introduce a refinement procedure on the dLTC of a composite service based on runtime information, which results in a more relaxed rLTC. The rLTC can be used to verify whether the composite service may still eventually satisfy the global time requirement at runtime.
2. We develop a fully automated tool to evaluate the proposed methods and apply it to real-world case studies to show the effectiveness of our approach.

The synthesized local time requirement has multiple advantages. First, it allows the selection of feasible services from a large pool of services with similar functionalities but

different local response times. Second, the designer can avoid over-approximations on the local response times. An over-approximation may lead the service provider to purchase a service at a higher cost, while a service at a lower cost with a slower response time may be sufficient to guarantee the global time requirement. Third, the local requirements serve as a safe guideline when component services are to be replaced or new services are to be introduced.

Outline. The rest of this chapter is structured as follows. Section 3.1 introduces a timed BPEL running example. Section 3.2 provides an overall picture on our approach. Section 3.3 introduces rLTC, and its usage for runtime adaptation of a service composition. Section 3.4 evaluates our approach using three case studies. Section 3.5 reviews related works. Finally, Section 3.6 concludes the chapter. The definitions and terminologies are supplemented in Appendix A.1, the approach to analyze the BPEL process and the synthesis algorithms for dLTC are supplemented in Appendix A.2 and A.3 respectively for reference.

3.1 A BPEL Example with Timed Requirements

BPEL [26] is an industrial standard for implementing composition of existing Web services by specifying an executable workflow using predefined activities. In this work, we assume the composite service is specified using the BPEL language. Basic BPEL activities that communicate with component Web services are `< receive >`, `< invoke >`, and `< reply >`, which are used to receive messages, execute component Web services and return values respectively. We denote them as *communication activities*. The control flow of the service is defined using structural activities such as `< flow >`, `< sequence >`, `< pick >`, `< if >`, etc.

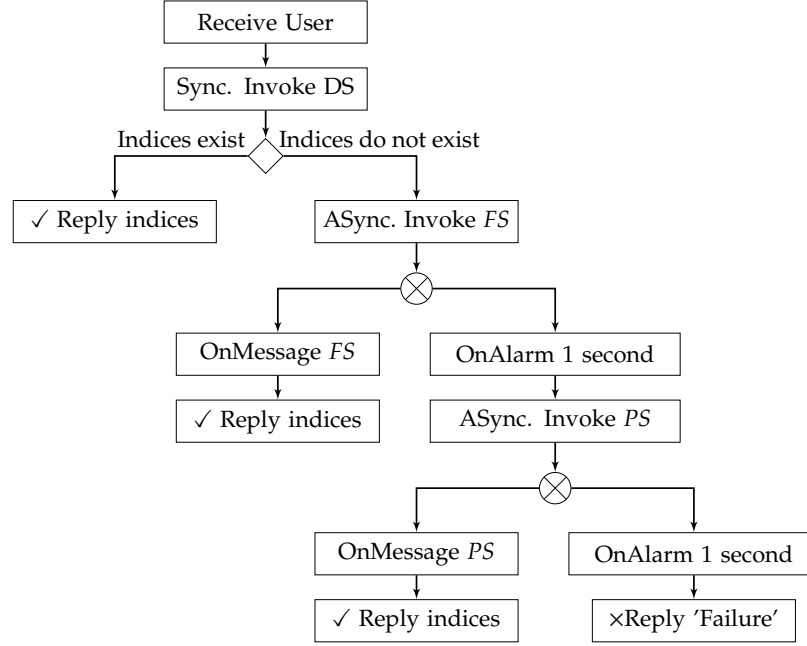


Figure 3.1: Stock Market Indices Service

Stock Market Indices Service

In this section, we illustrate a *Stock Market Indices Service* (SMIS) as a running example.

newsInfoShort is a paid service and its goal is to provide updated stock indices to the subscribed users. It provides service level agreement (SLA) to the subscribed users stating that it always responds within three seconds upon request.

SMIS has three component Web services: a database service (*DS*), a free news feed service (*FS*) and a paid news feed service (*PS*). The strategy of the SMIS is calling the free service *FS* before calling the paid service *PS* in order to minimize the cost. Upon returning the result to the user, the SMIS would also store the latest results in an external database service provided by *DS* (storage of the results is omitted here). The workflow of the SMIS is sketched in Figure 3.1 in the form of a tree. When a request is received from a subscribed

customer (Receive User), it would synchronously invoke (i.e., invoke and wait for reply) the database service (Sync. Invoke DS) to request for stock indices stored in the past minute. Upon receiving the response from *DS*, the process is followed by an `< if >` branch (denoted by \Diamond). If the indices are available (`indices exists`), then they are returned to the user (`Reply indices`). Otherwise, *FS* is invoked asynchronously (i.e., the system moves on after the invocation without waiting for the reply). A `< pick >` construct (denoted by \otimes) is used here to await incoming response (`< onMessage >`) from previous asynchronous invocation and timeout (`< onAlarm >`) if necessary. If the response from *FS* (`OnMessage FS`) is received within one second, then the result is returned to the user (`Reply indices`). Otherwise, the timeout occurs (`OnAlarm 1 second`), and SMIS stops waiting for the result from *FS* and calls *PS* instead (`ASync. Invoke PS`). Similar to *FS*, the result from *PS* is returned to user, if the response from *PS* is received within one second. Otherwise, it would notify the user regarding the failure of getting stock indices (`Reply 'Failure'`). The states marked with a \checkmark (resp. \times) represent desired (resp. undesired) end states.

The global time requirement for SMIS is that SMIS should respond within three seconds upon request. It is of particular interest to know the local time requirements for services *PS*, *FS*, and *DS*, so as to fulfill the global time requirement. This information could also help to choose a paid service *PS* which is both cheap and responds quickly enough.

3.2 Overall Approach

Figure 3.2 illustrates the main steps of our approach on synthesizing local time requirements. The required inputs are the specification of the composite service *CS*, and its global time requirement. These inputs are used to generate the labeled transition system representing the behavior of the composite service. The design-time local time constraint (dLTC) is subsequently generated based on the LTS of the composite service. The details of the LTS

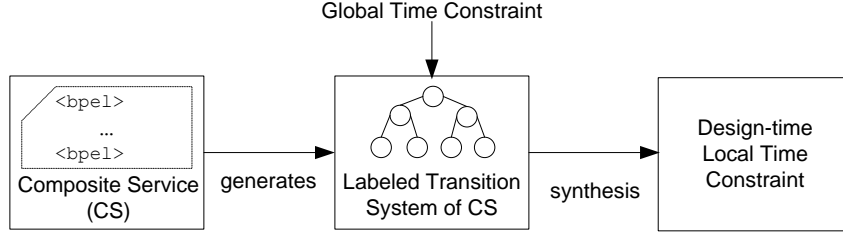


Figure 3.2: Synthesis of Local Time Requirement

generation are introduced in Appendix A.1 and Appendix A.2, and The details of synthesizing dLTC are given in Appendix A.3. The dLTC is then used to select a set of component services that collectively satisfy the global time requirement of the composite service. The component services are selected based on the upper bound of their response time, which could be specified by their SLA, using techniques from the research on worst-case execution time (WCET) (e.g., [67]), or based on estimations gathered from their past executions. We assume upper bounds of the response times of component services are known; how these values are collected or estimated is out of the scope of this chapter. We denote the estimated upper bound of a component service, as their *stipulated response time*. At runtime, the component service might not conform to its stipulated response time, due to the highly dynamic nature of the execution environment. If any component service is detected for violation of their stipulated response time, the runtime LTC (rLTC) is calculated based on the runtime information, such as the current active state of the LTS, and the total duration of the composite service. The rLTC, which is weaker than dLTC, is then used to judge whether the current execution could still satisfy the global time requirement. The details on rLTC are given in Section 3.3.

3.3 Runtime Refinement of Local Time Requirement

3.3.1 Motivation

Let us consider a composite service CS . Assume that we have selected a set of component services such that their stipulated response times fulfill the dLTC of CS . Since the composite service is executed under a highly evolving dynamic environment, the design time assumptions may evolve at runtime. For example, the response times of component services might be affected by network congestion – which might prevent some component services to conform to their stipulated response times. Nevertheless, this does not necessarily imply that the composite service will not satisfy the global time requirement. This is because the dLTC is synthesized at the design time to hold in *any* execution trace of CS ; whereas at runtime, the runtime information could be used to synthesize a more relaxed constraint for CS .

More specifically, given a composite service CS , we have two pieces of runtime information that could help in synthesizing a more relaxed constraint – the execution path that has been taken by CS , and the elapsed time of CS . The execution path that has been taken by CS could be used for *workflow simplification*. This is because in the midst of execution, some of the execution traces can be disregarded and therefore, a weaker LTC, that includes more parameter valuations, could be synthesized. In addition, the time elapsed of CS could be used to instantiate some of the response time parameters with real time constant; this makes the synthesized LTC contain less uncertainty and be more precise.

For example, consider the SMIS composite service, the LTS of which is depicted in Figure A.8. At runtime, after invocation of the component service DS , SMIS will be at state s_2 (see Figure A.8). Assume that DS does not conform to its stipulated response time. Therefore, it is desirable to check whether invoking FS could still satisfy the global time

requirement of CS . One can make use of dLTC for this purpose. Nevertheless, a better LTC could be synthesized at state s_2 .

The first observation is that from state s_2 , we can safely ignore the constraints from the good state s_5 , since it is not reachable from s_2 . The second observation is the delay from state s_0 to state s_2 (say r seconds, with $r \in \mathbb{R}_{\geq 0}$) is known. For this reason, we can substitute the delay component of state s_2 , which is the parametric response time t_{DS} , with the actual time delay r . This motivates the use of runtime information of BPEL process to refine the LTC. We refer to the runtime refined LTC as the runtime LTC (denoted by rLTC). By incorporating the runtime information, the resulting rLTC at state s_2 is:

$$\begin{aligned} (t_{FS} \leq 1) &\implies (r + t_{FS} \leq 3) \wedge \\ (t_{FS} \geq 1 \wedge t_{PS} \leq 1) &\implies (r + t_{PS} \leq 2) \wedge \\ \neg(t_{FS} \geq 1 \wedge t_{PS} \geq 1) \end{aligned}$$

3.3.2 Runtime Adaptation of a BPEL Process

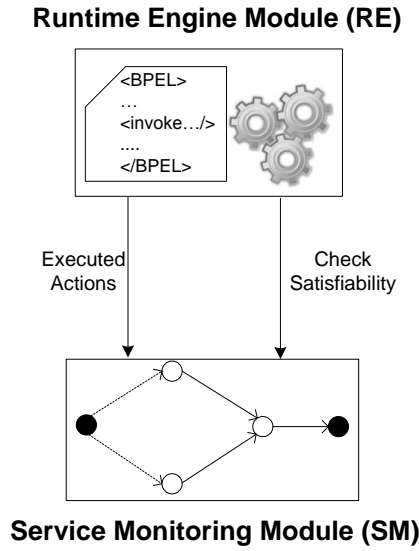


Figure 3.3: Service adaptation framework

In this section, we introduce a service adaptation framework to improve the conformance of global time requirement for a composite service. The framework makes use of rLTC and the architecture of the framework is shown in Figure 8.5. There are two modules in the framework – Runtime Engine Module (*RE*) and Service Monitoring Module (*SM*). The Runtime Engine Module (*RE*) provides an environment for the execution of a BPEL service; here, we use ApacheODE [3], an open source BPEL engine.

The Service Monitoring Module (*SM*) is used to monitor the execution of a BPEL service. During the deployment of a service *CS*, *SM* generates the LTS (Q, s_0, Σ, δ) of *CS* and stores it in the cache of *SM* so that it is available when *CS* is executing.

During the execution of the composite service *CS*, the executed action $a_e \in \Sigma$ from *RE* is used to update the active state $s_a \in Q$ of LTS stored in *SM*. The action a_e is also stored as part of the current execution run Π . *SM* also keeps track of the total execution time for execution run Π as well as the response time for each component service invocation.

Prior to the invocation of a component service *S*, *RE* will consult *SM* to check the satisfiability of rLTC. If the rLTC of s_a is satisfiable, then *SM* will instruct *RE* to continue invoking *S* as usual. Otherwise, some kind of mitigation procedure could be triggered. One of the possible mitigation procedures is to invoke a backup service of *S*, S_{bak} , which has faster stipulated response time than *S* (that may come with a cost). An example of *CS*, *S* and S_{bak} , are services *SMIS*, *FS* and *PS*. We introduce the details of synthesis of rLTC and satisfiability checking in Section 3.3.3 and Section 3.3.4 respectively.

3.3.3 Algorithm for Runtime Refinement

A way to calculate the rLTC is to run Algorithm 18 from a state *s* in the LTS. However, this requires traversing the state-space repeatedly for every calculation of the rLTC. To make it more efficient, we extend Algorithm 18 by calculating the rLTC for each state *s* during

Algorithm 1: Algorithm $rLTC(CS, s)$

input : Composite service CS

input : State s in LTS of CS

output: Constraint pair for sub-LTS of CS starting with s

```
1  $cons \leftarrow \emptyset$ ;  
2 if  $s$  is a good state then  
3    $cons \leftarrow (s.C \Rightarrow (s.D - d_1 + r_1 \leq T_G), true)$ ;  
4    $s.rLTC \leftarrow cons.g \wedge (d_1 = s.D)$ ;  
5 if  $s$  is a bad state then  
6    $cons \leftarrow (true, \neg(s.C))$ ;  
7    $s.rLTC \leftarrow cons.b$ ;  
8 if  $s$  is a non-terminal state then  
9    $SC \leftarrow \{rLTC(CS, s') | s' \in Enable(s)\}$ ;  
10   $a_1 \leftarrow \bigwedge \{c.g | c \in SC\}$  ;  
11   $a_2 \leftarrow \bigwedge \{c.b | c \in SC\}$  ;  
12   $cons \leftarrow (a_1, a_2)$ ;  
13   $s.rLTC \leftarrow cons.g \wedge cons.b \wedge (d_1 = s.D)$ ;  
14 return  $cons$ ;
```

the synthesis of the LTC at the design time. Therefore, during runtime, we only need to retrieve the synthesized $rLTC$ of the corresponding state for direct usage.

Algorithm 1 synthesizes the $rLTC$ for each state in the LTS. Given a composite service CS , and a state in \mathcal{LCS} , Algorithm 1 returns a constraint pair $c_s = (g, b)$, where $g, b \in C_U$. In this pair, g (resp. b) will be used to denote the constraint associated to a good (resp. bad) state. Given a constraint pair c_s , we use $c_s.g$ (resp. $c_s.b$) to refer to the first (resp. second) component of c_s . Variables d_1 and r_1 are free variables. Given a state s , free variables d_1 and r_1 in $s.rLTC$ are to be substituted by the delay component $s.D \in \mathcal{L}_U$ and the actual delay $r \in \mathbb{R}_{\geq 0}$ from the initial state to the state s respectively.

Given a good state s , $s.rLTC$ is assigned with value $cons.g$, with free variable d_1 substituted with $s.D$ (line 4). As an illustration, consider the good state s_{13} in the SMIS example. At runtime of SMIS, assume the active state is at state s_{13} , and assume that it takes $r \in \mathbb{R}_{\geq 0}$ seconds to execute from the initial state s_0 to state s_{13} . Therefore, the previously unknown

parametric response time in the delay component of state s_{13} , i.e., $t_{DS} + 1 + t_{PS}$, can be substituted with the real value r . To achieve this, at line 3, we subtract away the free variable d_1 , which is to be substituted with the response time parameter of state s_{13} , and add back the free variable r_1 , which is to be substituted with the real value r . We substitute the free variable d_1 at line 4. For free variable r_1 , it is only substituted in Algorithm 2 at runtime when the delay is known. In the case of the SMIS example, the $rLTC$ of state s_{13} after substituting free variable r_1 with value r (i.e., $s_{13}.rLTC \wedge (r_1 = r)$) is $((t_{PS} \leq 1 \wedge t_{FS} \geq 2) \implies (r \leq 3))$. The discussion when s is a bad state is similar.

Given a non-terminal state s , $s.rLTC$ is assigned with the conjunction of a_1 and a_2 , with free variable d_1 substituted with $s.D$ (line 13). Intuitively, the constraint a_1 (resp. a_2) is the conjunction of constraints synthesized from good (resp. bad) states. The reason for conjuncting both of them is to guarantee the reachability of at least one good state, and to avoid the reachability of all bad states from state s . Also note that the $rLTC$ of the initial state s_0 is the same as its $dLTC$, i.e., $s_0.rLTC = dLTC$. In fact, one can see Algorithm 1 as a generalization of Algorithm 17, in the sense that Algorithm 1 can be applied to any state (not only the initial one), and can benefit from the current partial execution.

3.3.4 Satisfiability Checking

Algorithm 2: Algorithm CheckSat($\mathcal{LCS}, s_a, r, T_{CS}$)

input : The LTS of the composite service CS , $\mathcal{LCS} = (Q, s_0, \Sigma, \delta)$

input : The active state $s_a \in Q$

input : Elapsed time for composite service, $r \in \mathbb{R}_{\geq 0}$

input : Stipulated response time information, $T_{CS} = \{(t_1, v_1), \dots, (t_n, v_n)\}$

output: Satisfiability of local time constraint at s_a

1 **return** $(\bigwedge_{1 \leq i \leq n} t_i \leq v_i) \implies (s_a.rLTC \wedge (r_1 = r));$

In this section, given a composite service CS with n component services $C = \{c_1, c_2, \dots, c_n\}$, we introduce how the satisfiability checking is performed prior to the invocation of a

component service $c_i \in C$. Let $\{t_1, t_2, \dots, t_n\}$ and $\{v_1, v_2, \dots, v_n\}$, where $t_i \in \mathcal{L}_U$ and $v_i \in \mathbb{R}_{\geq 0}$, be the set of parametric response times and stipulated response times for component services in C respectively. We denote by $T_{CS} = \{(t_1, v_1), \dots, (t_n, v_n)\}$ the stipulated response time information of component services CS . The algorithm to check the satisfiability of $rLTC$ at state $s_a \in Q$ is shown in Algorithm 2. With the assumption that all component services will reply within their stipulated response times ($\bigwedge_{1 \leq i \leq n} t_i \leq v_i$), it checks whether $rLTC$ at state s_a could be satisfied with free variables r_1 substituted with the actual elapsed time $r \in \mathbb{R}_{\geq 0}$.

3.3.5 Termination and Soundness

In this section, we show the termination and soundness of synthesis of $rLTC$.

3.3.5.1 Termination

Proposition 1. *Let CS be a service model, s be a state in \mathcal{LCS} . Then $rLTC(CS, s)$ terminates.*

Proof: *From Lemma 5, \mathcal{LCS} is acyclic. Algorithm 17 is obviously acyclic too. Now, Algorithm 18 is recursive (on line 9). However, due to the acyclic nature of \mathcal{LCS} , then no state is explored more than once. This ensures termination. \square*

3.3.5.2 Soundness

The following theorem formally states the correctness of our runtime refinement algorithm. It generalizes Theorem 11 to the case of runtime refinement.

Theorem 2. *Let CS be a service model. Let s be the current state and r be the current elapsed time. Let $\pi \models s.rLTC$. Let $LTS' = subLTS_{CS, s}$. Then:*

1. No bad activity is reachable in $LTS'_{CS[\pi]}$
2. There exists at least one reachable good state (V, P_g, C, d) in $LTS'_{CS[\pi]}$ and
3. For all good states (V, P_g, C, d) of $LTS'_{CS[\pi]}$, $d \leq T_G$.

Proof: The proof is similar to that of Theorem 11. We briefly prove the 3 items.

1. First note that we have conjuncted the bad state starting from s (line 6 and line 11 in Algorithm 1). Then, we can reuse the same proof as in Theorem 11 (Lemma 8), by using s instead of the initial state s_0 .
2. Again, we can reuse the same proof as in Theorem 11 (Lemma 9), since $subLTS_{CS,s}$ can be seen as a subtree of \mathcal{LCS} .
3. Assume s is a terminal state, then it must be a good state (by (1)). We have $s.rLTC = (s.C \Rightarrow s.D - d_1 + r_1 \leq T_G)$ (Algorithm 1, line 4). In addition, $d_1 = s.D$ (by Algorithm 1, line 4) and $r_1 = d$ (by Algorithm 2, line 1), where $s.D \in \mathcal{L}_U$ and $d \in \mathbb{R}_{\geq 0}$ are the parametric delay and real-value delay from the initial state to the good state s . After simplifying, we have $s.rLTC = (s.C \Rightarrow d \leq T_G)$. Hence, for any $\pi \models s.rLTC$, we have that $\pi \models (s.C \Rightarrow d \leq T_G)$. Therefore, the result holds. Assume s is a non-terminal state. For any reachable good state $s_g = (V, P_g, C, d)$ from s , the constraint $(s_g.C \Rightarrow (s_g.D - d_1 + r_1 \leq T_G))$ is included in Algorithm 1, line 10. And we have $d_1 = s.D$ (by Algorithm 1, line 13) and $r_1 = r$ (by Algorithm 2, line 1), where $s.D \in \mathcal{L}_U$ and $r \in \mathbb{R}_{\geq 0}$ are the parametric delay and real-value delay from the initial state to the non-terminal state s . The parametric delay $s_g.D - s.D$ represents the delay from the state s to the good state s_g . In short, $D = s_g.D - s.D + r$ represents the delay from the initial state to the good state s_g . Now, for any $\pi \models s.rLTC$, we have that $\pi \models (s_g.C \Rightarrow (D \leq T_G))$. Hence all reachable good states in $subLTS_{CS[\pi],s}$ are such that $d \leq T_G$.

□

3.3.6 Discussion

Termination. Our method is guaranteed to terminate. This is due to the fact that BPEL composite services do not support recursion, as well as our assumption on the loop activities such that the upper bound on the number of iterations and the time of execution is known. We discuss how to enforce such assumption if the loop activities exist. For the upper bound on the number of iterations, it could be either inferred by using loop bound analysis tool (e.g., [74]), or otherwise could be provided by the user. In the worst case, an option is also to set up a bound arbitrary but “large enough”. For the maximum time of loop executions, this could be enforced by using proper timeout mechanism in BPEL.

Time for internal operations. For simplicity, we do not account for the time taken for the internal operations of the system. In reality, the time taken by the internal operations might be significant especially when the process is large. In order to provide a more accurate synthesis of time constraint, an additional constraint $t_{overhead} \leq b$, where $t_{overhead} \in \mathbb{R}_{\geq 0}$ is a time overhead for an internal operation, and $b \in \mathbb{R}_{\geq 0}$ is a machine dependent upper bound for $t_{overhead}$, could be included for more precise analysis. The method in estimation of b is not the focus of this chapter; interested readers may refer to, e.g., [123].

Completeness of *rLTC*. *rLTC* is still incomplete in general, with the same reason for the incompleteness of *dLTC* as discussed in Section A.3.7. Nevertheless, it helps to mitigate the problem of incompleteness of *dLTC* with workflow simplification as illustrated in Section 3.3.1.

Bad Activity. The bad activities are the activities triggered when timeout occurs. For the running example SMIS, it is a reply activity that reports the user on the timeout of a composite service. As an additional example, it could also be an invocation activity to log

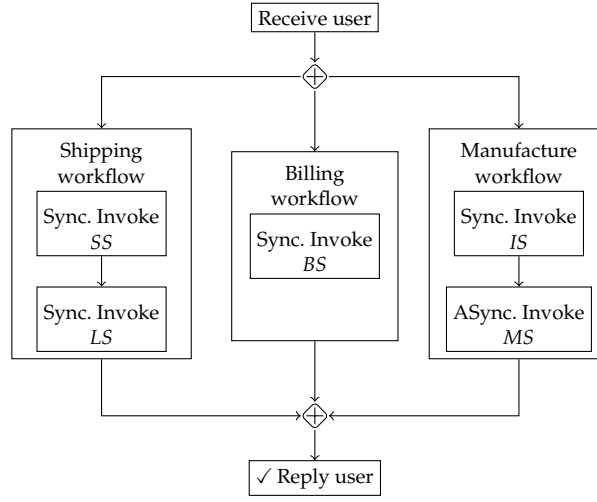


Figure 3.4: Computer Purchasing Service (CPS)

the timeout event upon the timeout of a composite service. With the rule of thumb that a bad activity is always triggered upon the timeout of a composite service, identifying a bad activity would become an obvious task; techniques for (semi-)automating this task is an interesting direction of future research. On the other hand, specifying bad activities is not mandatory. If the user cannot identify a bad activity in the composite service, (s)he has the option not to specify any.

3.4 Evaluation

We divide the evaluation of our approach into two parts. The first part (Section 3.4.2) focuses on the evaluation of synthesis of local time requirement at the design time. The second part (Section 3.4.3) focuses on the evaluation of runtime adaptation of a composite service. In the following, we first introduce the case studies that are used in the evaluation.

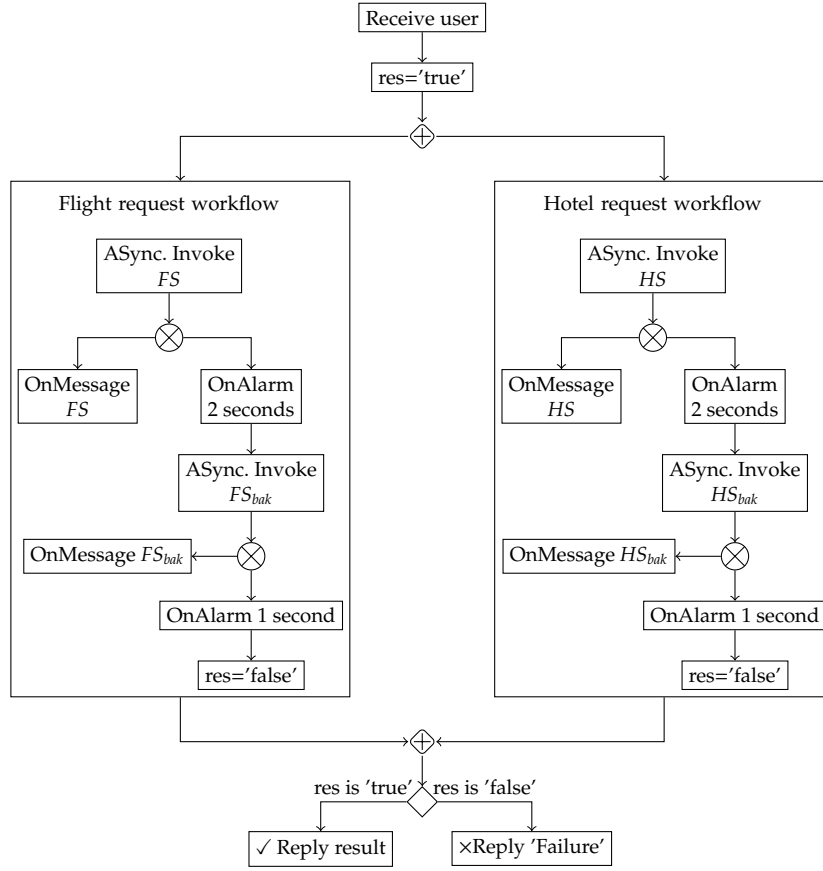


Figure 3.5: Travel Booking Service (TBS)

3.4.1 Case Studies

Stock Market Indices Service (SMIS). This is the running example introduced in Section 3.1.

Computer Purchasing Services (CPS). The goal of a CPS (e.g., Dell.com) is to allow a user to purchase a computer system online using credit cards. Our CPS makes use of five component services, namely Shipping Service (SS), Logistic Service (LS), Inventory Service (IS), Manufacture Service (MS), and Billing Service (BS). The global time requirement of the CPS is to respond within three seconds. The workflow of CPS is shown in Figure 3.4. A `< flow >` construct (denoted by \oplus) is introduced in the workflow for concurrent execution

of activities. The CPS starts upon receiving the purchase request from the client with credit card information, and the CPS spawns three workflows (viz., shipping workflow, manufacture workflow, and billing workflow) concurrently. In the shipping workflow, the shipping service provider is invoked synchronously for the shipping service on computer systems. Upon receiving the reply, *LS* which is a service provided by internal logistic department is invoked synchronously to record the shipping schedule. In the manufacture workflow, *IS* is invoked synchronously to check for the availability of the goods. Subsequently, *MS* is invoked asynchronously to update the manufacture department regarding the current inventory stock. In the billing workflow, the billing service which is offered by a third party merchant, is invoked synchronously for billing the customer with credit card information. Upon receiving the reply message from *LS* and *BS*, the result of the computer purchasing will be returned to the user.

Travel Booking Service (TBS). The goal of a travel booking service (TBS) (such as Booking.com) is to provide a combined flight and hotel booking service by integrating two independent existing services. TBS provides an SLA for its subscribed users, saying that it must respond within five seconds upon request. The travel booking system has four component services, Flight Service (*FS*), Backup Flight Service (*FS_{bak}*), Hotel Service (*HS*) and Backup Hotel Service (*HS_{bak}*). The workflow of TBS is shown in Figure 8.1a. Upon receiving the request from users, the variable *res* is assigned to true. After that, TBS spawns two workflows (viz., a flight request workflow, and a hotel request workflow) concurrently. In the flight request workflow, it starts by invoking *FS*, which is a service provided by a flight service booking agent. If service *FS* does not respond within two seconds, then *FS* is abandoned, and another backup flight service *FS_{bak}* is invoked. If *FS_{bak}* returns within one second, then the workflow is completed; otherwise the variable *res* is assigned to false. The hotel request workflow shares the same process as the flight request workflow, by replacing *FS* with *HS* and *FS_{bak}* with *HS_{bak}*. If *res* is true, the booking result will be returned to the

Case studies	#states	#transitions	dLTC (sec.)	rLTC (sec.)
SMIS	14	13	0.0076	0.0078
TBS	561	2386	1.004	1.05
CPS	120	119	0.0532	0.0562

Figure 3.6: Experiment results on synthesis

user, otherwise it will inform the user on the failure of travel booking.

3.4.2 Synthesis of Local Time Requirement

We synthesize the dLTC and rLTC for three case studies on a system using Intel Core I5 2410M CPU with 4 GB RAM. The details of the synthesis are shown in Figure 3.6. The *#states* and *#transitions* columns provide the information of number of states and transitions of the LTS respectively. The *dLTC (sec.)* and *rLTC (sec.)* columns provide the time (in seconds) spent for synthesizing dLTC (for the entire LTS), and rLTC (for each state in the LTS) respectively. TBS takes a longer time than SMIS and CPS for synthesizing dLTC and rLTC, as it contains a larger number of states and transitions compared to SMIS and CPS. Nevertheless, since both dLTC and rLTC are synthesized offline; the time for synthesizing the constraints (around one second) for TBS is considered to be reasonable.

The synthesized dLTC for SMIS is shown in Figure A.10, and dLTC for CPS and TBS are shown in Figure 3.7. All the dLTC are simplified and in DNF form. It is worth noticing the dLTC of CPS can be represented in one line representation after simplification. Note that t_{MS} does not appear in dLTC for CPS. The reason is that *MS* is invoked asynchronously without expecting a response; therefore its response time is irrelevant to the global time requirement of CPS.

For the synthesized rLTC, they are used for runtime adaptation during runtime. We evaluate the runtime adaptation of a composite service with rLTC in the next section.

$$(t_{SS} + t_{LS} + t_{IS} + t_{BS}) \leq 3$$

(a) Result of CPS

$$\begin{aligned} & ((2 \cdot t_{HSbak} < t_{FSbak}) \wedge (2 \cdot t_{FSbak} < t_{HSbak}) \wedge (t_{HSbak} < 1) \wedge (t_{FSbak} < 1)) \\ & \vee ((t_{HSbak} < 1) \wedge (t_{FSbak} < 1) \wedge (t_{FSbak} + t_{HSbak} \leq 1)) \\ & \vee ((t_{HSbak} < 1) \wedge (t_{FS} < 2)) \\ & \vee ((t_{HS} < 2) \wedge (t_{FSbak} < 1)) \\ & \vee ((t_{HS} < 2) \wedge (t_{FS} < 2)) \end{aligned}$$

(b) Result of TBS

Figure 3.7: Synthesized dLTC

3.4.3 Runtime Adaptation

In this section, we conduct two experiment to evaluate the *overhead* that caused by runtime adaptation, and the *improvement* provided by runtime adaptation on the conformance of the global time requirement.

3.4.3.1 Setup of Experiment

The evaluation was conducted using two different physical machines, which are connected by a 100 Mbit LAN. One machine is running ApacheODE [3] to host the RE module to execute the BPEL program, configured with Intel Core I5 2410M CPU with 4GiB RAM. The other machine is to host the SM module, configured with Intel I7 3520M CPU with 8GiB RAM.

To test the composite service under controlled situation, we introduce the notion of *execution configuration*. An execution configuration defines a particular execution scenario for the composite service. Formally, an execution configuration E is a tuple (M, R) , where M decides which path to choose for an `< if >` activity and R is a function that maps a component service $s_i \in S_{CS}$ to a real value $r \in \mathbb{R}_{\geq 0}$, which represents the response time of service s_i . We discuss how an execution configuration $E = (M, R)$ is generated. M is generated by

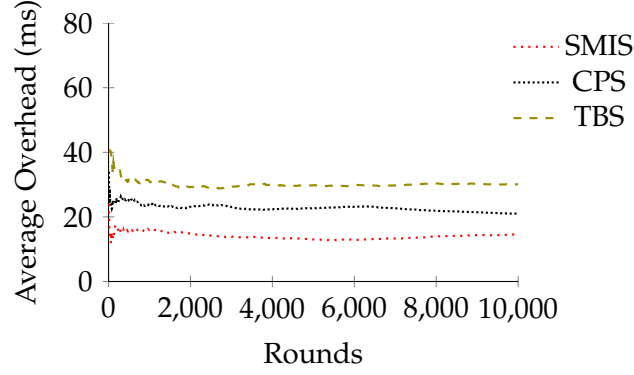


Figure 3.8: Overhead of runtime monitoring

choosing one of the branches of an `< if >` activities uniformly among all possible branches.

Let CS be a composite service, where a component service $s_i \in S_{CS}$ has a stipulated response time $v_i \in \mathbb{R}_{\geq 0}$. Then $R(s_i)$ will return a response time within the stipulated response time v_i with a probability of $p_c \in \mathbb{R}_{\geq 0} \cap [0, 1]$. p_c is the *response time conformance threshold*. More specifically, $R(s_i)$ will be assigned to a value in $[0, v_i]$ uniformly with a probability of p_c , and assigned to a value in $(v_i, v_i + t_e]$ uniformly with a probability of $1 - p_c$. $t_e \in \mathbb{R}_{\geq 0}$ is the *exceeding threshold*; and assume after $v_i + t_e$ seconds, the component service c_i will be automatically timeout by RE to prevent an infinite delay.

Given a composite service CS , and an execution configuration E , a *run* is denoted by $r(CS, A, E)$, where the first argument is the composite service CS that is running, the second argument $A \in \{rr, \emptyset\}$ is the adaptive mechanism where rr denotes the runtime adaptation, and \emptyset denotes no adaptation. Two runs $r(CS, A, E)$ and $r(CS', A', E')$ are equivalent if $CS = CS'$, $A = A'$ and $E = E'$. Note that all equivalent runs have the same execution paths and response times for all service invocations.

	p_c	N_{se}	N_e	Improvement (%)	Avg. Backup Service
SMIS	0.9	9441	8976	4.65	0.127
	0.8	9211	8374	8.37	0.352
	0.7	8109	6965	11.44	0.577
	0.6	7593	6348	12.45	0.702
TBS	0.9	10000	9743	2.57	0.384
	0.8	10000	9364	6.36	0.779
	0.7	10000	8460	15.40	0.948
	0.6	10000	7700	23.00	1.05
CPS	0.9	9523	8809	7.14	1.259
	0.8	9241	7156	20.85	1.589
	0.7	8504	6108	23.96	2.014
	0.6	8430	5650	27.80	2.578

Figure 3.9: Improvement of runtime conformance

3.4.3.2 Evaluation Results

We conducted two experiments, and we show the results and findings in the following. Each experiment goes through 10,000 rounds of simulation, and an execution configuration E is generated for each round of simulation. Given a composite service CS , we assume that for each component service c_i with a stipulated response time v_i , there exists a backup service c'_i , with a stipulated response time $v_i/2$ and a conformance threshold of 1. Suppose that before the invocation of a component service c_i , CS is at active state s_a . The satisfaction of the rLTC at s_a will be checked using Algorithm 1 before c_i is invoked. If it is satisfiable, then it will invoke c_i as usual. Otherwise, some kind of mitigation procedure will be used. The mitigation procedure used in the experiment is to invoke the backup service c'_i instead.

E1. Given a composite service CS , in order to measure the overhead, we use an execution configuration $E = (M, Q)$ for an adaptive run $r(CS, rr, E)$, and non-adaptive run $r(CS, \emptyset, E)$.

We have modified the adaptation mechanism for rr such that, if rLTC of the active state is checked to be unsatisfiable, component service c_i will still be used (instead of c'_i). The purpose for this modification is to make $r(CS, rr, E)$ and $r(CS, \emptyset, E)$ invoke the same set of component services, so that we can effectively compare the overhead of $r(CS, rr, E)$.

Results. Suppose at round k , the times spent for $r(CS, rr, E)$ and $r(CS, \emptyset, E)$ are $r_{rr}^k \in \mathbb{R}_{\geq 0}$ time units and $r_{\emptyset}^k \in \mathbb{R}_{\geq 0}$ time units respectively. The overhead O_k at round k is the time difference between r_{rr}^k and r_{\emptyset}^k , i.e., $O_k = r_{rr}^k - r_{\emptyset}^k$. The average overhead at round k is calculated using Equation 3.1.

$$\text{Average overhead} = (\sum_{i=1}^k O_i) / k \quad (3.1)$$

The main source of overhead for runtime adaptation comes from the satisfiability checking with Algorithm 2. We make use of the state-of-the-art SMT solver Z3 [63] for this purpose. Other sources of overhead include update of active state in SM , and communications between SM and RE .

The experiment results can be found in Figure 3.8. The average overheads of SMIS, CPS and TBS after 10,000 rounds are 15 ms, 21 ms, and 30 ms respectively. The results convey to us that the additional operations involved in the runtime adaptation, including the satisfiability checking, can be done efficiently.

E2. In this experiment, we measure the improvement for the conformance of global constraints due to rr . Given a composite service CS , an execution configuration E , two runs $r(CS, rr, E)$ and $r(CS, \emptyset, E)$ are conducted for each round of simulation. N_{se} is the number of executions that satisfy global constraints for composite service with rr , and N_e is the number of executions that satisfy global constraints for composite service without rr , the improvement is calculated by Equation 3.2.

$$\text{Improvement} = \frac{N_{se} - N_e}{10000} \quad (3.2)$$

Results. The experiment results can be found in Figure 3.9. The *Improvement (%)* column provides the information of improvement (in percentage) that is calculated using Equation 3.2. The *Avg. Backup Service* column provides the average number of backup service used (calculated by summing the number of backup services used for 10,000 rounds, and divided by 10,000).

The decrement of p_c represents the undesired situation where component services have a higher chance for not conforming to their stipulated response time. This could be due to situations such as poor network conditions. For each case study, the improvement provided by the runtime adaptation increases when p_c decreases. This shows that runtime adaptation improves the conformance of global time requirement. In addition, the average backup services usage increases when p_c decreases. This shows the adaptive nature of runtime adaptation with respect to different p_c – more corrective actions are likely to perform when the chances that component services do not satisfy their stipulated response time increase.

3.4.4 Threat to Validity

There are several threats to validity. The first threat to validity is due to the fact that we assume uniform distribution of response time for evaluation of runtime adaptation. To address this issue, more experimentations with real-world services should be performed. This said, our experiments on real case studies provide a first idea that our assumptions are realistic.

The second threat to validity is stemmed from our choice to use a few example values as experimental parameters, that include global constraints and termination thresholds, in order to cope with the combinatorial explosion of options. To resolve this problem, it is clear that even more experimentations with different case studies and experimental parameters should be performed, so that we could further investigate the effects that have

not been made obvious by our case studies and experimental parameters.

3.5 Related Work

This work shares common techniques with work for constraint synthesis for scheduling problems. The use of models such as parametric timed automata (PTAs) [24] and parametric time Petri nets (PTPNs) [158] for solving such problems has received recent attention. In particular, in [55, 110, 83], parametric constraints are inferred, guaranteeing the feasibility of a schedule using PTAs extended with stopwatches (see, e.g., [16]). In [28], we extended the “inverse method” (see, e.g., [30]) to the synthesis of parameters in a parametric, timed extension of CSP. Although PTAs or PTPNs might have been used to encode (part of) the BPEL language, our work is specifically adapted and optimized for synthesizing local timing constraint in the area of service composition.

Our method is related to using LTSs for analysis purpose in Web services. In [45], the authors propose an approach to obtain behavioral interfaces in the form of LTSs of external services by decomposing the global interface specification. It also has been used in model checking the safety and liveness properties of BPEL services. For example, Foster *et al.* [80] transform BPEL process into FSP [113], subsequently using a tool named “WS-Engineer” for checking safety and liveness properties. Simmonds *et al.* [142] propose a user-guided recovery framework for Web services based on LTSs. Our work uses LTSs in synthesizing local time requirement.

Our method is related to the finding of a suitable quality of service (QoS) for the system [168]. The authors of [168] propose two models for the QoS-based service composition problem: a combinatorial model and a graph model. The combinatorial model defines the problem as a multidimension multichoice 0-1 knapsack problem. The graph model defines the problem as a multiconstraint optimal path problem. A heuristic algorithm is proposed

for each model: the WS-HEU algorithm for the combinatorial model and the MCSP-K algorithm for the graph model. The authors of [33] model the service composition problem as a mixed integer linear problem where constraints of global and local component serviced can be specified. The difference with our work is that, in their work, the local constraint has been specified, whereas for ours, the local constraint is to be synthesized. An approach of decomposing the global QoS to local QoS has been proposed in [19]. It uses the mixed integer programming (MIP) to find optimal decomposition of QoS constraint. However, the approach only concerns for simplistic sequential composition of Web services method call, without considering complex control flow and timing requirement.

Our method is related to response time estimation. In [109], the authors propose to use linear regression method and a maximum likelihood technique for estimating the service demands of requests based on their response times. [120] has also discussed the impact of slow services on the overall response time on a transaction that use several services concurrently. Our work is focused on decomposing the global requirement to local requirement, which is orthogonal to these works.

Our method is related to service monitoring. Moser *et al.* [123] present VieDAME, a non-intrusive approach to monitoring. VieDAME allows monitoring of BPEL composite service on quality of service attributes, and existing component services are replaced based on different replacement strategies. They make use of the aspect-oriented approach (AOP); therefore the VieDAME engine adapter could be interwoven into the BPEL runtime engine at runtime. Baresi *et al.* [38] propose an idea of self-supervising BPEL processes by supporting both service monitoring and recovery for BPEL processes. They propose the use of Web Service Constraint Language (WScOL) to specify the monitoring directives to indicate properties need to be hold during the runtime of composite service. They also make use of the AOP approach to integrate their monitoring adapters with the BPEL runtime engine. Our work is orthogonal to the aforementioned works, as we do not assume any particular

service monitoring framework for monitoring the composite service, and those methods can be used to aid the monitoring approach, as discussed in Section 3.3.2.

Our previous work [111] complements with this work by proposing a method on building LTCs that under-approximate the dLTC of a composite service. The under-approximated LTCs consisting of independent constraints over components, which can be used to improve the design, monitoring and repair of component-based systems under time requirements.

3.6 Chapter Summary

We have presented a novel technique for synthesizing local time constraints for the component services of a composite service CS, knowing its global time requirement. Our approach is based on the analysis of the LTS of a composite service by making use of parameterized timed techniques. The synthesis algorithm utilizes the constraints from the LTS to synthesize design-time local time constraint (dLTC) for component services. The dLTC is used to select a set of component services that could collectively satisfy the global time requirement in design time. Then, during the runtime of composite service, we propose the usage of the runtime information to weaken the dLTC, which becomes the runtime local time constraint (rLTC). The rLTC is used to validate whether the composite service could satisfy the global time requirement at runtime. We have implemented the approach and applied it to three case studies.

Chapter 4

Dynamic Ranking Optimization for QoS-Aware Service Composition

Web services provide an affordable and adaptable framework that can produce a significantly lower cost of ownership for the enterprises over time. Figure 4.1 shows a simple composite service example, named Travel Agency Service (TAS). If the user requests to travel by road, the Car Booking Service (CBS) will be invoked to book a car for reaching the destination. If the user requests to travel by air, the Flight Booking Service (FBS) will be invoked to book the flight for reaching the nearest airport to the destination. In either case, the Hotel Booking Service (HBS) will be invoked to book the hotel. Finally, the system replies to the user with the status of booking.

Quality of Service (QoS) attributes, such as response time, availability, cost, etc., provide quantitative indicators on non-functional aspects of the composite service. The Service Level Agreement (SLA) is often used as an agreement between the composite service providers and the service users; it specifies the expected QoS level of the composite service. The SLA can be expressed as constraints over the QoS, e.g., the response time must be

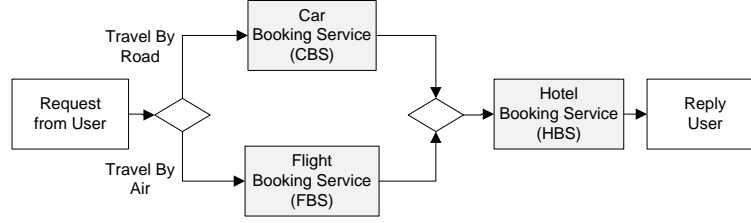


Figure 4.1: Travel Agency Service (TAS)

smaller than one second, or the availability must be higher than 99.99%.

Following the SOA, a composite service contains a set of abstract services (e.g., a hotel booking service) which have their interfaces defined, and concrete services (e.g., the Hilton Hotel booking service) are selected to realize the interfaces of abstract services during runtime. A composite service user typically has limited knowledge on the internal structure of the composite service, and component services that are involved. The service user is only concerned with the end-to-end QoS of composite service, i.e., the QoS at the composite service level. Therefore, the objective is to select a set of concrete services that could satisfy the global constraints on the composite service, while maximizing the end-to-end QoS.

The problem of selecting concrete services while maximizing end-to-end QoS, known as the *optimal selection problem*, has received considerable attentions over the last decade [22, 19, 35, 33]. The optimal selection problem becomes increasingly challenging as the number of concrete services increases. Given a composite service with 10 abstract services executed in a sequential manner, with each abstract service having 10 concrete service candidates, there are 10^{10} combinations to explore. In fact, it has been shown that the problem is NP-hard [33]. Therefore, it is impossible in practice to exhaustively search through all possible combinations of concrete services.

To address the scalability problem, approaches based on selecting representative component services have been proposed. We denote the optimality of QoS for a concrete component service with respect to other functional-equivalent services as *local optimality*. Existing

representative services approaches [22, 19] provide search space reduction based on local optimality of the services. Although it has been shown to provide good performance for a large number of candidates with less restrictive global constraints, it suffers from significant performance degradation when the global constraints become more restrictive because it requires more iterations of the approach to find the solution. The primary reason is that they do not take the global constraints into account during service selection.

In this chapter, we propose a novel approach, called *dynamic ranking optimization* (DRO), to address this problem. The key idea behind DRO is the use of both global constraints and local optimality to achieve search space reduction. DRO is divided into three stages, viz., service preprocessing stage, service ranking stage, and dynamic service selection stage. In the service preprocessing stage, the services that are verified impossible to be part of the optimal selection are pruned from the search space. Subsequently, in the service ranking stage, services are ranked according to both local optimality value and constraint satisfaction probability. Intuitively, local optimality value and constraint satisfaction probability of a service quantify the optimality of the service, and the likelihood that the service could contribute to the conformance of global constraints respectively. Following that, in the dynamic service selection stage, representative services are selected for the optimal end-to-end QoS that could satisfy global constraints. The number of representative services is decided dynamically, based on the constraint satisfaction probability of representative services.

DRO is a novel technique for optimizing the existing service selection approaches. Our main contributions are summarized below.

1. We introduce a pruning method called *constraint pruning*, that could effectively discard the service candidates that cannot satisfy the global constraints.
2. We introduce a ranking method based on both the overall local optimality and the

constraint satisfaction probability of a service. The service candidates that are ranked higher are more likely to be part of the optimal selection. We further propose a method for selecting the ranked representatives, where the number of representatives to be chosen is decided dynamically based on the constraint satisfaction probability of a service.

3. We evaluate the benefits brought by the DRO using a synthetically generated dataset and a publicly available dataset. The results have shown significant improvement on accuracy and performance over existing approaches.

Outline. Section 4.1 introduces the QoS compositional model and necessary terminologies. Section 4.2 presents DRO. Section 4.3 provides the evaluation of our approach. Section 4.5 reviews related works. Finally, Section 4.6 concludes the chapter.

4.1 QoS-Aware Compositional Model

In this section, we define the QoS compositional model used in this chapter. Following the SOA principles, service providers need to characterize their services to define both the offered functionalities and the offered quality. An abstract service specifies the functionality of the service without referring to any concrete service instance. An *abstract service* can be defined as a service class $S = \{s_1, \dots, s_n\}$ which contains n functionality equivalent concrete services s_i that can be used to realize the functionality specified by the abstract service. We use $|S|$ to denote the total number of concrete services in S . We use the notation \widehat{S} to denote an abstract service S' that is a subset of S , i.e., $S' \subseteq S$.

An *abstract composite service* CS_a specifies the compositional workflows using a set of abstract services $CS_a = \langle S_1, \dots, S_n \rangle$ for fulfilling the service requests. A *concrete composite service* $CS = \langle s_1, \dots, s_n \rangle$ is defined as an instantiation of abstract composite service $CS_a = \langle S_1, \dots, S_n \rangle$,

where each abstract service S_i is replaced by a concrete service $s_i \in S_i$. We use $|CS_a|$ to denote the total number of abstract services in CS_a .

For simplicity of illustration, we only consider the “travel by air” branch of the TAS example. This results in a composition where services FBS and HBS are running sequentially. We will use the modified TAS example as a running example in this chapter, and henceforth, simply refer to it as the TAS example. For the TAS example, there are two abstract services FBS and HBS. Suppose that the concrete services for FBS and HBS are $\{f_1, f_2, f_3, f_4\}$ and $\{h_1, h_2, h_3, h_4\}$ respectively. Then the abstract composite service TAS_a is $\langle FBS, HBS \rangle = \langle \{f_1, f_2, f_3, f_4\}, \{h_1, h_2, h_3, h_4\} \rangle$, and a possible concrete composite service of TAS_a could be $TAS = \langle f_1, h_2 \rangle$.

4.1.1 QoS Attributes

In this chapter, we deal with non-functional attributes that can be quantitatively measured using metrics. The values of QoS attributes can be solicited from service providers (e.g., cost), from users’ feedback (e.g., reliability), or based on past record of the execution (e.g., response time). We assume the values of QoS attributes are known; how these values can be collected or estimated is out of the scope of this chapter. There are two classes of attributes: positive ones (e.g., availability) and negative ones (e.g., response time). Positive attributes have positive effects on the QoS, and therefore they need to be maximized. Conversely, negative attributes need to be minimized. For simplicity, we only consider negative attributes in this work, since positive attributes can be transformed into negative attributes by multiplying their values with -1 . Given r QoS attributes of a concrete component service s , we use an attribute vector $Q_s = \langle q_1(s), \dots, q_r(s) \rangle$ to represent the QoS attribute values of service s , where $q_i(s)$ is the i th QoS attribute value of s . Similarly, given a concrete composite service CS , we use the attribute vector $Q_{CS} = \langle q'_1(CS), \dots, q'_r(CS) \rangle$ to represent it, where $q'_i(CS)$ is the i th end-to-end QoS attribute value of CS .

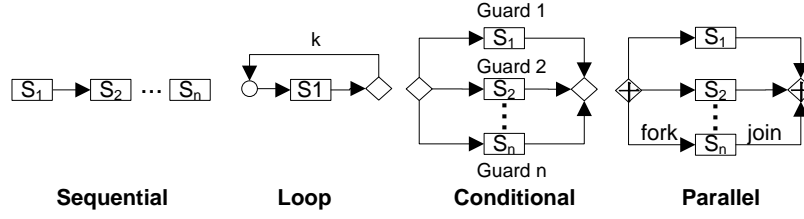


Figure 4.2: Compositional structure

The SLA often specifies a set of constraints on the end-to-end QoS. Such constraints define the lower bound for positive constraints (e.g., the availability must be higher than 99.99%) and the upper bound for negative constraint (e.g., the response time must be less than 500 ms). Given a composite service CS with QoS attribute vector $Q_{CS} = \langle q'_1(CS), \dots, q'_r(CS) \rangle$, the global constraints of CS can be represented as a vector $C_{CS} = \langle C_1, \dots, C_r \rangle$ where $C_i \in \mathbb{R}$ and $q'_i(CS) \leq C_i$. Without loss of generality, we use $C_i = \infty$ to denote the situation where $q'_i(CS)$ is unconstrained.

4.1.2 QoS for Composite Services

There are four elementary compositional structures for composing the component services: sequential, parallel, loop and conditional, as shown in Figure 4.2. Sequential composition of services $\{s_1, \dots, s_n\}$ executes the services sequentially, one after another. Parallel composition of services $\{s_1, \dots, s_n\}$ executes the services concurrently. A loop executes a service s_1 repeatedly up to k iterations. Conditional composition of services $\{s_1, \dots, s_n\}$ executes exactly one of the services according to the evaluation of the guard conditions, where the guards are mutually exclusive.

The end-to-end QoS is aggregated from the QoS on the component services, based on the service compositional structures, and the types of QoS attributes. Table 7.1 shows the aggregation functions of component services with respect to the compositional structures.

We use three QoS attributes, viz., response time, availability, and throughput to demon-

strate the aggregation functions. Other common QoS attributes have the similar aggregation functions as these three QoS attributes. For example, QoS attributes such as reliability, share the same QoS function as availability. The response time $r \in \mathbb{R}_{\geq 0}$ is the average delay between sending a request and receiving a response. For the sequential composition, the response time of the service composition is obtained by summing up the response time of the component services. For the parallel composition, it is equal to the maximum response time among the participating component services. For the loop composition, it is calculated by summing each of the involved component service k times, where k is the maximum number of loop iterations. For the conditional composition, since the evaluation of guards is not known at design time, the maximum response time of n services is chosen as the response time of the composite service. The availability $a \in \mathbb{R} \cap [0, 1]$ is the probability of the service being available. For the sequential composition, this implies that all the services are available during the sequential execution; therefore, the availability of the composite composition is the multiplication of the component services' availability. For other compositions, their aggregation functions are similar. The throughput $t \in \mathbb{R}_{\geq 0}$ is the number of invocations that can be handled by a composite service per second. The throughput of a composite service is decided by the minimum throughput of the component service, which is essentially the "bottleneck" of the throughput for the composite service.

Given the aggregation functions in Table 7.1, we define a function $agg : \mathbb{N} \mapsto \{sum, mult, min, max\}$, where $agg(k), k \in \mathbb{N}$ returns the type of aggregation functions – either summation (*sum*), multiplication (*mult*), minimum (*min*), or maximum (*max*) – for QoS attribute q'_k . For example, suppose the attribute vector of TAS example is in the form of $Q_{TAS} = \langle q'_1(TAS), q'_2(TAS) \rangle$, where attribute q'_1 provides the value of response time, and attribute q'_2 provides the value of availability, then we have $agg(1) = sum$ and $agg(2) = mult$ for TAS.

Given an abstract composite service CS_a , a composite service CS'_a is a *subset* of CS_a , denoted by $CS'_a \subseteq CS_a$, if CS'_a and CS_a have the same compositional structure, and every service class

of CS'_a is a subset of the corresponding service class of CS_a . Formally, $CS'_a \subseteq CS_a$ if CS'_a and CS_a share the same compositional structure C , with $|CS'_a| = |CS_a|$ and $\forall S_i \in CS'_a, \forall S_j \in CS_a : (i = j) \implies S_i \subseteq S_j$. Given a composite service CS_a , a *reduced abstract composite service* of CS_a , denoted by $\widehat{CS_a}$, is used to represent any composite service $CS'_a \subseteq CS_a$, e.g., an example of $\widehat{TAS_a}$ is $\langle \{f_1, f_2\}, \{h_1, h_2, h_3\} \rangle$.

4.1.3 Optimality Function

Concrete services have multi-dimensional attributes, and we need a methodology to facilitate their comparison in term of their QoS. In this work, we use a Simple Additive Weighting (SAW) technique [167] to obtain a score for multi-dimensional attributes. SAW uses two phases: normalization and weighting, for producing the score. The normalization stage normalizes the values of QoS attributes so that they are independent of their units and ranges to allow comparison. The weighting stage allows users to specify their preferences on different QoS attributes. In the normalization stage, a service compares its QoS attribute values with the maximum and minimum QoS attributes of other service candidates within a service class. A composite service compares its aggregated QoS attributes with the maximum and minimum aggregated QoS attributes. The maximum (resp. minimum) aggregated QoS attributes can be obtained by aggregating maximum (resp. minimum) QoS attributes from each service class. Formally, we have

$$\begin{aligned} G_{min}(k) &= F_{(k)}^n(L_{min}(i, k)) \\ G_{max}(k) &= F_{(k)}^n(L_{max}(i, k)) \end{aligned} \tag{4.1}$$

with

$$\begin{aligned} L_{min}(i, k) &= \min_{\forall s \in S_i} q_k(s) \\ L_{max}(i, k) &= \max_{\forall s \in S_i} q_k(s) \end{aligned} \tag{4.2}$$

QoS Attribute	Sequential	Parallel	Loop	Conditional
Response Time	$\sum_{i=1}^n q(s_i)$	$\max_{i=1}^n q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^n q(s_i)$
Availability	$\prod_{i=1}^n q(s_i)$	$\prod_{i=1}^n q(s_i)$	$(q(s_1))^k$	$\min_{i=1}^n q(s_i)$
Throughput	$\min_{i=1}^n q(s_i)$	$\min_{i=1}^n q(s_i)$	$q(s_1)$	$\min_{i=1}^n q(s_i)$

Table 4.1: Aggregation Function

where $G_{min}(k)$ and $G_{max}(k)$ are the minimum and maximum aggregated values for the k th QoS attribute for the composite service, $F_{(k)}$ is the QoS aggregation function for attribute k which is given in Table 7.1, $L_{min}(i, k)$ and $L_{max}(i, k)$ are the minimum and maximum aggregated values for the k th QoS attribute for service class i .

Suppose each service has r QoS attributes. The local optimality function $L_i(s)$ computes the local optimality value of a concrete service s within service class S_i , where $s \in S_i$, as follows.

$$L_i(s) = \sum_{k=1}^r v(i, k, s) \cdot w_k \quad (4.3)$$

with

$$v(i, k, s) = \begin{cases} \frac{L_{max}(i, k) - q_k(s)}{L_{max}(i, k) - L_{min}(i, k)} & \text{if } L_{max}(i, k) \neq L_{min}(i, k) \\ 1 & \text{if } L_{max}(i, k) = L_{min}(i, k) \end{cases}$$

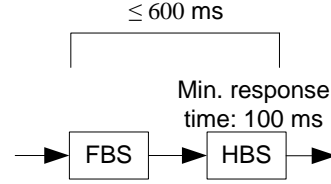
where $w_k \in \mathbb{R}_{\geq 0}$ is the weight of q_k and $\sum_{k=1}^r w_k = 1$.

The global optimality function $G(CS)$ computes the global optimality value of concrete composite service CS as follows.

$$G(CS) = \sum_{k=1}^r v'(k, CS) \cdot w_k \quad (4.4)$$

Concrete Services	Response Time (ms)	Availability
f_1, h_1	100	0.85
f_2, h_2	300	0.92
f_3, h_3	500	0.95
f_4	600	0.94
h_4	600	0.8

(a) Concrete services of TAS



(b) Service preprocessing

Figure 4.3: TAS Example

with

$$v'(k, CS) = \begin{cases} \frac{G_{max}(k) - q'_k(CS)}{G_{max}(k) - G_{min}(k)} & \text{if } G_{max}(k) \neq G_{min}(k) \\ 1 & \text{if } G_{max}(k) = G_{min}(k) \end{cases}$$

where $w_k \in \mathbb{R}_{\geq 0}$ is the weight of q'_k and $\sum_{k=1}^r w_k = 1$.

4.1.4 Problem Statement

We first define the notions of feasible and optimal selection.

Definition 1. Given an abstract composite service $CS_a = \langle S_1, \dots, S_n \rangle$, and global QoS constraints $C = \langle C_1, \dots, C_r \rangle$ for CS_a , a feasible selection is a selection of concrete services CS , such that CS contains exactly one service for each service class S_i in CS_a and CS satisfies the global QoS constraints C , i.e., $q'_k(CS) \leq C_k$.

Definition 2. An optimal selection is a feasible selection of concrete services CS that maximizes the global optimality value $G(CS)$.

Given an abstract composite service CS_a , and a set of global constraints C , we are interested in finding the optimal selection CS . To address this problem, a naive approach is to exhaustively explore all combinations of concrete services for each service class. However,

given n services in sequential composition, with each of them having l candidates, the total number of combinations is l^n . In fact this method can be modeled as an instance of a combinatorial problem, viz., the multi-dimensional multi-choice knapsack problem (MMKP) [131], which is NP-hard. To allow real-time QoS-aware service composition, it is desirable to find a near-optimal selection at an acceptable cost, rather than finding an exact solution to the optimal selection problem at a very high cost [33].

One way to mitigate this problem is to select a subset of service candidates each time, instead of working on all services candidates. This is efficient especially when the number of service candidates is large and hard to be handled. The question that arises is how to identify a set of representative service candidates that not only satisfy the constraints, but also contribute to high global optimality value. In this work, our goal is to address the optimal selection problem efficiently and effectively by progressively exploring subsets of candidates that are likely to contribute to the optimal or near-optimal selection.

TAS Example. We provide the details of non-functional properties on the TAS example, which will be used in the following sections. Each of the abstract services of TAS_a has four concrete services, where $FBS = \{f_1, f_2, f_3, f_4\}$ and $HBS = \{h_1, h_2, h_3, h_4\}$. The attribute vector of TAS example is in the form of $Q_{TAS} = \langle q'_1(TAS), q'_2(TAS) \rangle$, where attribute q'_1 provides the value of response time, and attribute q'_2 provides the value of availability. The values of response time and availability of all concrete services are given in Figure 4.3(a). In addition, the global constraints for response time and availability of TAS are 600 ms and 0.8 respectively.

4.2 Dynamic Ranking Optimization

As discussed in the previous section, given an abstract composite service, our goal is to search for a set of concrete services, one from each service class, that not only satisfy

all global constraints, but also maximize the global optimality value. A way to avoid exploring all combinations of component services is to select the most promising service candidates, which are referred to as *representative services*, from each service class, and to verify whether there is a feasible selection. One metric we can use for local selection is the local optimality value of a concrete service. Although the local optimality value provides an indicator for overall QoS of the concrete service, naively selecting the service with the highest local optimality value from each service class may not work, since this approach does not take into consideration of the satisfaction of global constraints. For example, given a composite service with only a component service with two candidate s_1 , and s_2 . The global QoS constraints for response time, cost, and reliability are 5 seconds, 2 dollars and 80%. The response time, cost and reliability of services s_1 and s_2 are 10, 1, 90% and 5, 2, 80%, respectively. Now, even if s_1 has the highest local optimality value, however, the response time of s_1 can not satisfy the requirements, s_2 should be selected. Therefore, besides the local optimality value, we also need to consider how likely a concrete service could contribute to the conformance of the global constraints. We denote this likelihood as the *constraint satisfaction probability*. With this observation, we propose in the following a new method, called *dynamic ranking optimization* (DRO), which is driven by both the local optimality value and constraint satisfaction probability to reduce the search space for the optimal selection. The workflow of DRO is illustrated in Figure 4.4. DRO is divided into three stages, viz., service preprocessing stage, service ranking stage, and dynamic service selection stage. The details of each of the stages are introduced in the following sections.

4.2.1 Service Preprocessing

The global constraints play an important role in selecting the services. For example, it has been shown that the heuristic method proposed in [22] has its performance degraded when the global constraints become more restrictive. In this section, we show that the

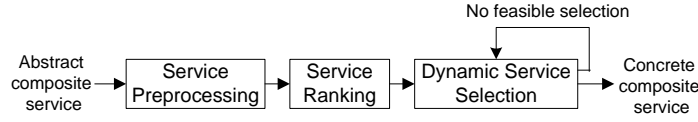


Figure 4.4: Dynamic Ranking Optimization (DRO)

global constraints could be used as an effective means for pruning service candidates. Consider the TAS example in Figure 4.3(b) where FBS and HBS are running sequentially, and the TAS global constraint for response time is 600 ms. We know that the concrete service that has the fastest response time for HBS is service s'_1 , which has response time of 100 ms. Therefore, a service $s \in \text{FBS}$ must have a response time smaller than 500 ms, in order to fulfill the global constraint for response time. We refer to a service that may fulfill the global constraints as a *constraint-satisfiable service*. Formally, given a composite service $CS = \langle S_1, \dots, S_n \rangle$, and global constraints $C_{CS} = \langle C_1, \dots, C_r \rangle$, a service $s_i \in S_i$ is a constraint-satisfiable service if the following condition holds.

$$\forall k : \quad F_{(k)}(v_j) \leq C_k \quad (4.5)$$

$$j \in \{1, \dots, n\}$$

with

$$v_j = \begin{cases} q_k(s_j) & \text{if } j = i \\ L_{min}(j, k) & \text{if } j \neq i \end{cases}$$

where $F_{(k)}$ is the QoS aggregation function for attribute k . For example if the QoS aggregation function is a summation, then the condition becomes

$$\forall k : \quad (q_k(s_i) + \sum_{j \in \{1, \dots, n\} \setminus i} L_{min}(j, k)) \leq C_k$$

We can safely prune all constraint-unsatisfiable services as they cannot satisfy the global constraints. For the TAS example, concrete services f_4 and h_4 are the only concrete-

unsatisfiable services, because some of their QoS attributes do not satisfy Condition (4.5). In particular, it is the response times of f_4 , h_4 and the availability of h_4 that do not satisfy the condition.

In addition to pruning using global constraints, we also include in the service preprocessing stage the pruning of non-skyline services [22]. Let us first recall the notion of *dominance*.

Definition 3 (Dominance). *Let S be a service class, and s, s' be two services, where $s, s' \in S$ and $s \neq s'$. Service s dominates service s' , denoted by $s < s'$, if the service s is at least as good as service s' in all QoS parameters and better than service s' in at least one QoS parameter, i.e., $\forall k \in \{1, \dots, |Q_S|\}: q_k(s) \leq q_k(s')$, and $\exists k \in \{1, \dots, |Q_S|\}: q_k(s) < q_k(s')$.*

A service $s \in S$ is denoted as a *skyline service* if there does not exist a service in service class S that dominates s (i.e., $\neg \exists s' \in S : s' < s$). In other words, a service $s \in S$ is a *non-skyline service*, if there exists a service $s' \in S$ that dominates s .

It can be shown that we can safely prune non-skyline services in service class CS , without affecting the result of optimal selection [19]. For our TAS example, concrete service h_4 is the only non-skyline service.

Henceforth, we denote the pruning of constraint-unsatisfiable services and non-skyline services as *constraint pruning* and *non-skyline pruning* respectively. The algorithm of service preprocessing is shown in Algorithm 3. The abstract composite service CS'_a is initialized with empty service classes (line 1), where \emptyset_i denotes i th empty service class. After that, each service class of CS_a will be undergone constraint pruning (line 3) and non-skyline pruning (line 4). The service class after pruning will be assigned to CS'_a (line 5). Finally, abstract composite service CS'_a which contains the preprocessed service classes will be returned (line 6). In the TAS example, the composite service TAS'_a after service preprocessing stage is $\langle \{f_1, f_2, f_3\}, \{h_1, h_2, h_3\} \rangle$.

Algorithm 3: Service Preprocessing (DROPreprocess)

input : Abstract composite service $CS_a = \langle S_1, \dots, S_n \rangle$

output: Preprocessed abstract composite service $CS'_a = \langle S'_1, \dots, S'_n \rangle$

```
1  $CS'_a \leftarrow \langle \emptyset_1, \emptyset_2, \dots, \emptyset_n \rangle;$ 
2 for  $i = 1$  to  $n$  do
3    $S''_i = \text{ConstraintPruning}(S_i);$ 
4    $S'_i = \text{NonSkylinePruning}(S''_i);$ 
5    $CS'_a[i] \leftarrow S'_i;$ 
6 return  $CS'_a;$ 
```

After the initial preprocessing, we propose in the following a ranking based method that could allow us to limit the selection to a smaller set of services that have higher chance contributing to the optimal selection.

4.2.2 Service Ranking

The local optimality value of a concrete service could reflect the quality of a service. Nevertheless, it does not provide a direct indication on how likely the concrete service can contribute to the conformance of the global constraints. To address this problem, we propose an estimation on the probability of the constraint satisfaction of a concrete service. First, we introduce the notion of *local constraint*, which is the average constraint value that has to be fulfilled by individual service classes. Given a global constraint C_i , we define the local constraint c_i as follows, where n is the number of service classes.

$$c_i = \begin{cases} C_i/n & \text{if } agg(i) = sum \\ C_i^{1/n} & \text{if } agg(i) = mult \wedge C_i \geq 0 \\ -|C_i|^{1/n} & \text{if } agg(i) = mult \wedge C_i < 0 \\ C_i & \text{if } agg(i) = min, max \end{cases} \quad (4.6)$$

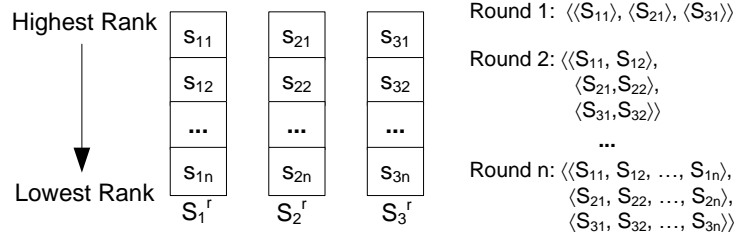


Figure 4.5: A scenario of dynamic service selection

Consider again the TAS example: since the global constraints for response time and availability are 600 ms and 0.8 respectively, the local constraints are $600/2 = 300$ ms and $-|-0.8|^{1/2} \approx -0.89$ respectively. Recall that we treat all positive QoS attributes as negative ones by multiplying their attribute values with -1.

Given a composite service $CS_a = \langle S_1, \dots, S_n \rangle$, and global constraints of CS_a as $C = \langle C_1, \dots, C_r \rangle$, the *constraint satisfaction probability* of a service $s \in S_i$, denoted by $P_i(s)$, is calculated as follows:

$$P_i(s) = \gamma \left(\sum_{k=1}^r p_k(s) \cdot w_k \right) \quad (4.7)$$

where $w_k \in \mathbb{R}_0^+$ is the weight of $p_k(s)$, $\sum_{i=1}^r w_k = 1$ and $\gamma \in \mathbb{R} \cap [0, 1]$ is the *credibility factor* (explained later). The calculation of $p_k(s)$ is divided into two cases. When $q_k(s) > c_k$, then

$$p_k(s) = \begin{cases} \frac{D_{max}(i, k) - (q_k(s) - c_k)}{D_{max}(i, k)} & \text{if } agg(k) = sum, mult \\ 1 & \text{if } agg(k) = min, max \end{cases} \quad (4.8)$$

with

$$D_{max}(i, k) = \max_{\forall s_i \in S_i} (q_k(s_i) - c_k)$$

and when $q_k(s) \leq c_k$, $p_k(s) = 1$. In the following, we explain the details of Equation 4.7 and Equation 4.8.

In Equation 4.7, $P_i(s)$ is calculated by using the SAW method that is introduced in Section 4.1.3, followed by multiplying with the credibility factor γ . The credibility factor γ is used to adjust the value of $P_i(s)$ based on how much we can trust $P_i(s)$. We set the value of credibility factor γ to $1/n$, where n is the number of service classes. The reason for this choice is that, with the increment of n , $P_i(s)$ would be more dependent on the choices made by other service classes. Because when we have more service classes n , we have more external factors. For example, given two service classes s_1, s_2 , when we are calculating S_1 , we only affect by s_2 . But for $s_1 \dots, s_{10}$, we are affecting by $s_2 \dots, s_{10}$, which makes us to have less certainty for the result we calculated. To estimate the probability more conservatively, we choose to lower the constraint satisfaction probability, given that the number of service class n increases.

In Equation 4.8, $q_k(s) > c_k$ signifies that $q_k(s)$ has violated the local constraint (reminded that negative attributes are assumed here). The value of $p_k(s)$ is decided by the value of $q_k(s)$. If $q_k(s)$ has a smaller value, it has a higher chance for satisfying the global constraints, hence the higher value of $p_k(s)$. Given a service class S_i , $D_{max}(i, k)$ calculates the maximum difference between $q_k(s_i)$ and the local constraint c_k . $D_{max}(i, k)$ serves the purpose of normalizing the value of probability $p_k(s)$, such that $p_k(s) \in \mathbb{R} \cap [0, 1]$.

For minimum and maximum aggregations, since the QoS values between each service class are independent to each other, therefore to be conservative, we set the value $p_k(s) = 1$. For the case of $q_k(s) \leq c_k$, it indicates the conformance of $q_k(s)$ to the local constraint, and we simply set $p_k(s) = 1$. In TAS, the constraint satisfaction probabilities for concrete services are $P_1(f_1)=P_2(h_1)=0.25$, $P_1(f_2)=P_2(h_2)=0.5$, and $P_1(f_3)=P_2(h_3)=0.25$.

Given a composite service $CS_a = \langle S_1, \dots, S_n \rangle$, the service ranking is performed by ordering the services in each service class S_i by the multiplication value of local optimality value and constraint satisfaction probability of each service $s \in S_i$, i.e., $L_i(s) \cdot P_i(s)$, in the descending order. We denote the ranked service class as $S_i^r = \langle s_1, \dots, s_n \rangle$, which is an ordered sequence

Algorithm 4: Optimization with DRO

input : Abstract composite service to be solved CS_a
input : Termination threshold for constraint-satisfiability $\epsilon \in \mathbb{R} \cap [0, 1]$
output: A feasible selection of concrete composite services CS

- 1 $CS'_a \leftarrow DROPreprocess(CS_a)$;
- 2 $CS^r_a \leftarrow DRORank(CS'_a)$;
- 3 $\widehat{CS^r_a} \leftarrow \langle \emptyset_1, \emptyset_2, \dots, \emptyset_n \rangle$;
- 4 $round \leftarrow 1$;
- 5 **repeat**
- 6 $\widehat{CS^r_a} \leftarrow DROSelect(CS^r_a, \widehat{CS^r_a}, \epsilon, round)$;
- 7 $CS \leftarrow OptimalSelection(\widehat{CS^r_a})$;
- 8 **if** $CS \neq \emptyset$ **then**
- 9 **return** CS ;
- 10 $round \leftarrow round + 1$;
- 11 **until** $\widehat{CS^r_a} = CS^r_a$;
- 12 **return** \emptyset ;

of all concrete services $s_i \in S_i$. The ranked composite service containing the ranked services as $CS^r_a = \langle S^r_1, \dots, S^r_n \rangle$. For TAS example, given the composite service TAS'_a returned by service preprocessing, the ranked composite service $TAS^r_a = \langle \langle f_2, f_1, f_3 \rangle, \langle h_2, h_1, h_3 \rangle \rangle$, where the values of $L_i(s) \cdot P_i(s)$ for concrete services $f_1(h_1)$, $f_2(h_2)$, and $f_3(h_3)$ are 0.125, 0.3, and 0.125 respectively.

4.2.3 Dynamic Service Selection

After obtaining the ranked composite service CS^r_a , the next stage is to perform service selection on CS^r_a . We illustrate the dynamic service selection with an example as shown in Figure 4.5. Consider that $CS^r_a = \langle S^r_1, S^r_2, S^r_3 \rangle$, where the services of each service class S^r_i can be found in Figure 4.5. The dynamic service selection uses multiple rounds of selection for selecting the service representatives. At the first round, one service is chosen from each service class; they form a *reduced* composite service $\widehat{CS^r_a} = \langle \widehat{S^r_1}, \widehat{S^r_2}, \widehat{S^r_3} \rangle$, where $\widehat{S^r_1} = \langle s_{11} \rangle$, $\widehat{S^r_2} = \langle s_{21} \rangle$, and $\widehat{S^r_3} = \langle s_{31} \rangle$. The reduced composite service $\widehat{CS^r_a}$ is then solved, e.g., by MIP

Algorithm 5: Dynamic service selection (DROSelect)

input : Abstract composite service $CS_a^r = \langle S_1^r, \dots, S_n^r \rangle$
input : Reduced abstract composite service $\widehat{CS}_a^r = \langle \widehat{S}_1^r, \dots, \widehat{S}_n^r \rangle$
input : Termination threshold for constraint-satisfiability $\epsilon \in \mathbb{R} \cap [0, 1]$
input : Current round, *round*
output: Abstract composite service $\widehat{CS}_a^{r'}$, where $\widehat{CS}_a^r \subseteq \widehat{CS}_a^{r'} \subseteq CS_a^r$

```
1  $\widehat{CS}_a^{r'} \leftarrow \langle \emptyset_1, \emptyset_2, \dots, \emptyset_n \rangle$ ;  
2 for  $i = 1$  to  $n$  do  
3    $serviceCount \leftarrow |\widehat{S}_i^r|$ ;  
4    $prob \leftarrow 1$  ;  
5   for  $j = |\widehat{S}_i^r|$  to  $|S_i^r|$  do  
6      $prob \leftarrow prob \cdot (1 - P_i(S_i^r[j]))$ ;  
7      $serviceCount \leftarrow serviceCount + 1$ ;  
8      $p \leftarrow 2^{round-1}$ ;  
9     if  $(1 - prob) \geq \epsilon^{1/p}$  then  
10      break ;  
11  $\widehat{CS}_a^{r'}[i] \leftarrow \{s_j \in S_i^r \mid j \in [1, serviceCount]\}$  ;  
12 return  $\widehat{CS}_a^{r'}$  ;
```

solver, for optimal selection (details will be given in Section 4.2.4). Note that the reduced composite service \widehat{CS}_a^r contains less services than CS_a^r . Therefore, the optimal selection can be performed more efficiently. If a solution is found, then the result is returned to the user. Otherwise, we proceed to next round by selecting more services from each service class. The process is repeated until a solution is found or until all services in the service classes have been explored. In the latter case, the selection approach guarantees that it does not miss out a solution if one exists.

The next problem is how to determine the number of representatives for each service class S_i^r at each round of selection: the number of services should be large enough for finding a solution to the service composition, while small enough to allow for efficient computation. We propose to use constraint satisfaction probability to address the problem. We first extend the definition of constraint satisfaction probability to a set of services. Given S

as a subset of services from service class S_i , i.e., $\$ \subseteq S_i$, we define constraint satisfaction probability of $\$$, denoted as $P_i(\$)$, as the probability that at least one of the services in $\$$ succeed in satisfying the global constraints after composition. $P_i(\$)$ is calculated as follows:

$$P_i(\$) = 1 - \prod_{s \in \$} (1 - P_i(s)) \quad (4.9)$$

We observe that the reason that the optimal selection cannot produce a solution is due to global constraint violation. Therefore, the capacity of the set of representatives $\$$ should be large enough, so that $P_i(\$)$ can exceed ϵ , which is a parameter provided by the user on the intended termination threshold.

Algorithm 4 presents the DRO algorithm, which is the entry algorithm for dynamic selection. Initially, the input abstract composite service CS_a passes through two initial stages of DRO: service preprocess and service ranking, using functions *DROPreprocess* and *DRORank* respectively (lines 1, 2). Function *DROPreprocess* has been introduced in Algorithm 3, and function *DRORank* sorts the services for each service class $S_i \in CS'_a$ by the value of $L_i(s) \cdot P_i(s)$, where $s \in S_i$, in the descending order. The details of *DRORank* can be found in Section 4.2.2. The reduced abstract composite service \widehat{CS}_a^r is initialized with empty service classes (line 3), and the current round number *round* is initialized to value 1 (line 4). After that, *DROSelect* (see Algorithm 5) is called to populate each service class of the reduced composite function \widehat{CS}_a^r with the representative services (line 6). Subsequently, it is passed to the *OptimalSelection* function (line 7) for optimal selection, where the details will be introduced in Section 4.2.4. The result of the optimal selection is stored in *CS*. If the optimal selection is successful, i.e., when *CS* is not empty, the result is returned (line 9); otherwise, we proceed to the next round of DRO. This continues until all concrete services are explored, i.e., when $\widehat{CS}_a^r = CS_a^r$ (line 11). If the optimal selection could not find a result, an empty result is returned (line 12).

Algorithm 5 presents the dynamic selection algorithm. Initially, the reduced abstract composite service \widehat{CS}_a^r is initialized with empty service classes. For each service class (line 2), the *serviceCount* is initialized to the number of services in service class \widehat{S}_i^r (line 3), and the accumulated probability *prob* is initialized with 1 (line 4). Starting from index $|\widehat{S}_i^r|$, we accumulate the constraint satisfaction probability of service $S_i^r[j]$, until it exceeds the threshold $\epsilon^{1/p}$, where $p = 2^{round-1}$ (lines 5–10). Note that the threshold $\epsilon^{1/p}$ increases with the number of rounds. The reason is based on the assumption that if there exists an optimal selection, concrete services in the optimal selection are likely to be ranked higher by the ranking algorithm in respective services classes. Therefore, at the starting round, choosing a smaller amount of services that are ranked higher from each service class will decrease the solving time for the optimal selection. Conversely, the chance of getting a feasible selection decreases when the number of rounds increases. Hence, choosing a larger amount of services at later rounds will lead to a faster exploration of all concrete services, especially in the case where there does not exist a feasible selection.

Afterwards, the respective service classes in \widehat{CS}_a^r are then populated with the representative services with the amount equals to *serviceCount* (line 11). Finally, we return the reduced composite service \widehat{CS}_a^r , which contains the newly inserted representatives (line 12).

For TAS, the number of services that are chosen at each round depends on the termination threshold for constraint-satisfiability ϵ . If $\epsilon = 0.4$, then for the first round, we get the reduced composite service $\widehat{TAS}_a^r = \langle \langle f_2 \rangle, \langle h_2 \rangle \rangle$. $OptimalSelection(\widehat{TAS}_a^r)$ will return a feasible selection $TAS = \langle f_2, h_2 \rangle$, which is also the optimal selection for TAS example. Since a feasible selection is found, therefore TAS terminates at the first round of *DROSelect*. If $\epsilon = 0.9$, for the first round, we get $\widehat{TAS}_a^r = \langle \langle f_2, f_1, f_3 \rangle, \langle h_2, h_1, h_3 \rangle \rangle = TAS_a^r$. $OptimalSelection(\widehat{TAS}_a^r)$ will return the same optimal solution as in the former case where $\epsilon = 0.4$. However, since there are more concrete services, it might take longer time to solve for the latter case where $\epsilon = 0.9$. We will investigate how different values of ϵ affect the service selection process

in Section 4.3.

4.2.4 Solving for Optimal Selection

In this section, we introduce the use of Mixed Integer Programming (MIP) to realize the *OptimalSelection* at line 7 of Algorithm 4. Mixed Integer Programming (MIP) is a technique for minimization or maximization of an objective function subjected to a set of constraints. A binary decision variable x_{ij} is used to represent the selection of service candidates s_{ij} . If a service candidate is selected, then x_{ij} is set to 1, and to 0 otherwise. By Equation 4.3 and Equation 8.2, an MIP model can be specified as a maximization of the objective function below:

$$\sum_{k=1}^r \frac{G_{max}(k) - F_{(k)} \sum_{j=1}^n \sum_{i=1}^{|S_j|} q_k(s_{ij}) \cdot x_{ij}}{G_{max}(k) - G_{min}(k)} \cdot w_k \quad (4.10)$$

subjected to the QoS constraints

$$(F_{(k)} \sum_{j=1}^n \sum_{i=1}^{|S_j|} q_k(s_{ij}) \cdot x_{ij}) \leq C_k, 1 \leq k \leq r \quad (4.11)$$

where r and n are the total number of attributes and service classes respectively. In addition, since we only choose one service per service class, therefore the following constraint must be hold.

$$\sum_{i=1}^{|S_j|} x_{ij} = 1, 1 \leq j \leq n \quad (4.12)$$

4.3 Evaluation

There are several tools or libraries that could be used to solve the MIP model, examples are Gurobi solver [95] and lpsolve solver [43]. Note that to solve the above MIP model requires the linearization of the objective function and the QoS constraints, we omit the technical

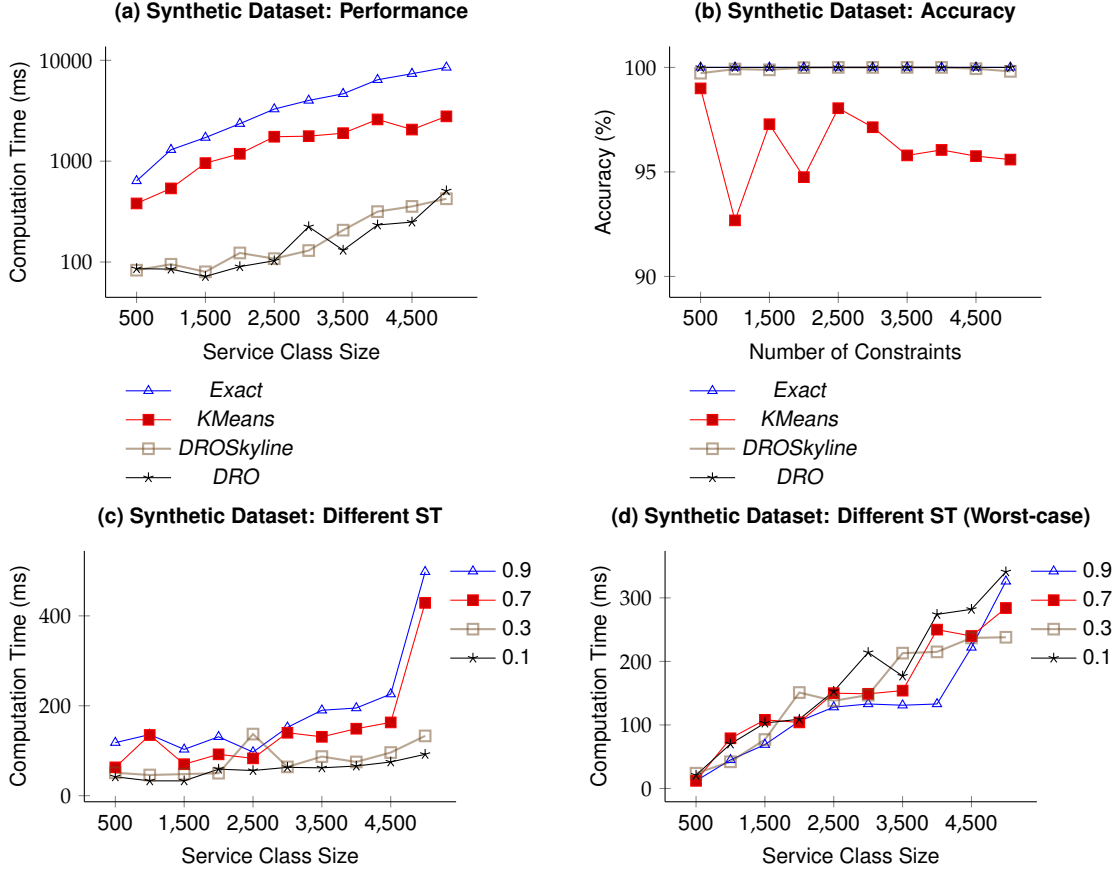


Figure 4.6: Experiment results for synthetic dataset

details here, and refer the readers to [21].

4.4 Evaluation

We conducted experiments to evaluate our approach to service selection using DRO. Specifically, we attempted to answer the following research questions.

RQ1. How is the *performance* of DRO in comparison to the state-of-the-art?

We analyze how different stages contribute to the performance of DRO. Firstly, we investigate how well service preprocessing performs. In particular, what is the number of services

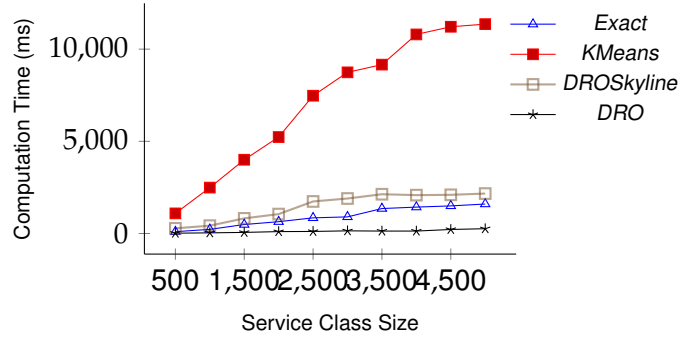


Figure 4.7: Worst-case performance for synthetic dataset

Name	Agg.	Type	Name	Agg.	Type
Response Time	Sum	-	Compliance	Mult	+
Availability	Mult	+	Best Practices	Mult	+
Throughput	Min	+	Latency	Sum	-
Successability	Mult	+	Documentation	Mult	+
Reliability	Mult	+			

Figure 4.8: Attributes for synthetic/QWS dataset

that are pruned in the service preprocessing stage. Secondly, we also look into the performance of DRO algorithm without the constraint pruning preprocessing. The reason is to facilitate the comparison with the state-of-the-art, as well as to enable us to evaluate how well the service ranking and dynamic service ranking work without utilizing the constraint pruning preprocessing. Lastly, we evaluate the performance of DRO as a whole.

RQ2. How is the *accuracy* of the concrete services that are selected by DRO?

We measure the accuracy using the formula

$$accuracy = \frac{G(CS_{heu})}{G(CS_{exact})} \quad (4.13)$$

where $G(CS_{heu})$ and $G(CS_{exact})$ are the global optimality values of concrete services returned by heuristic method and exact method respectively.

RQ3. How *scalable* is DRO?

Size	1000	2000	3000	4000	5000
NonSkyline (%)	23.1	29.7	34.4	38.0	40.2
Constraint (%)	91.8	91.1	91.0	90.9	90.6
DROPreprocess (%)	92.1	91.5	91.7	91.8	91.6

Figure 4.9: Preprocess results for synthetic dataset

Size	500	1000	1500	2000	2500
NonSkyline (%)	70.2	86.5	88.9	89.6	91.0
Constraint (%)	94.6	94.6	94.4	94.3	93.9
DROPreprocess (%)	96.2	97.0	97.2	97.6	98.0

Figure 4.10: Preprocess results for QWS dataset

To evaluate the scalability of DRO, we use a synthetically generated dataset generated using a publicly available dataset generator¹.

RQ4. How do different termination threshold values ϵ influence the selection process of DRO?

The performance of DRO may depend on the threshold value ϵ . We investigate how DRO performs under different threshold values.

Implementation. We implemented all algorithms in C#. For solving the mixed integer programming models, we used the Gurobi solver [95]. The experiments were conducted on an Intel Core I5 2410M CPU with 4 GB RAM, running on Windows 7.

Experimental Setup. To answer the previous research questions, we evaluate DRO using a synthetically generated dataset and a real-world dataset. All datasets that are used in the experiments are publicly available². We compare our methods with [22] which is the existing state-of-the-art for the optimal selection of services, to the best of our knowledge. For effective comparison, we choose to use dataset of anti-correlated distribution, as it presents

¹<http://pgfoundry.org/projects/randddataset>

²<http://sites.google.com/site/droselection>

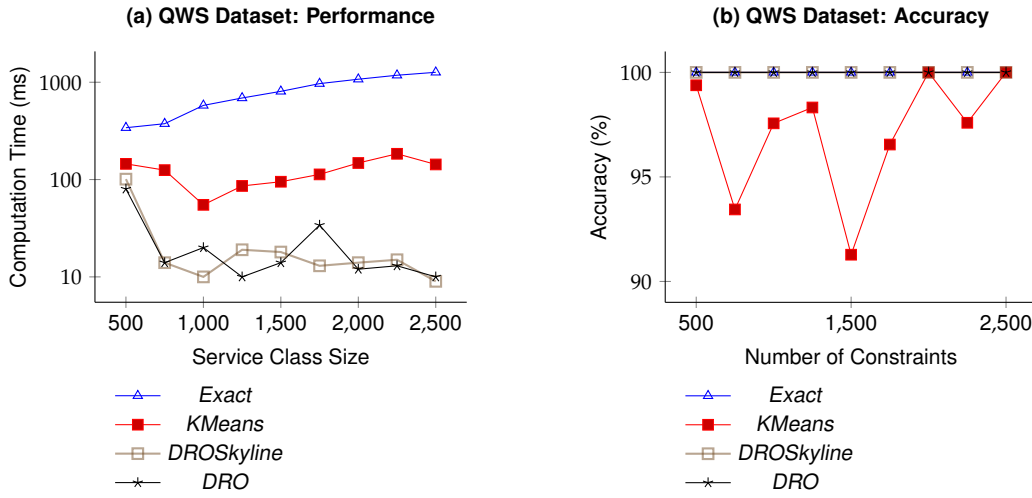


Figure 4.11: Experiment results for QWS dataset

the most challenging dataset for methods that make use of non-skyline pruning [22]. Note that both [22] and our methods utilize non-skyline pruning in the preprocessing stage. We compare the efficiency of the following QoS-based composition methods:

Exact The standard global optimization problem for all service candidates represented in the MIP model.

KMeans The hierarchical k -means clustering method proposed in [22].

DROSkyline Our method described in Section 4.2, with the modification that only non-skyline pruning is used in the preprocessing stage.

DRO Our method described in Section 4.2.

The reason for adding *DROSkyline* is because *KMeans* is also using non-skyline pruning as their preprocessing method. Using the same preprocessing method will allow both methods to reduce the same amount of services at the preprocessing stage, therefore enable us to compare the strength of the service selection algorithms for these two methods.

4.4.1 Evaluation with a Synthetic Dataset

We use a synthetic dataset generated using a publicly available dataset generator¹.

The synthetic dataset contains 50,000 QoS vectors, with 9 QoS attributes. The details of these QoS attributes can be found in Figure 4.8. In Figure 4.8, there are three columns. The *Name* and *Agg.* columns provide the name of the QoS attributes and the aggregation functions (Summation (*Sum*), Multiplication (*Mult*), Minimum (*Min*)) for those attributes. The *Type* column shows that whether these attributes are negative (–) or positive (+). We randomly partition the dataset into 10 service classes and the QoS constraints are set randomly. We conduct several experiments (E1 – E6) and report our findings in the following. For experiments E1 to E4, we set the termination threshold at 0.9 and for experiments E1 to E3, there exists a feasible selection that could satisfy the global constraints.

E1: We record the number of services that have been pruned using non-skyline pruning (*NonSkyline*), constraint pruning (*Constraint*), and the combination of both pruning methods (*DROPreprocess*). The numbers reported are the percentage of services that are pruned with respect to the total number of services. The experiment results can be found in Figure 4.9.

Results. The non-skyline pruning has achieved 23.1%-40.2% reduction rate. As the dataset for experiment has anti-correlated distribution, and it is known that the non-skyline pruning does not perform well in such dataset [22], therefore we can expect the method could perform better for datasets of different distributions (e.g., correlated distribution). For constraint pruning, it has achieved 90.6%-91.8% reduction rate. The high reduction rate is due to a seemingly lenient global constraint of composite service could result in a high expectation of the QoS attribute of component services. For example, the global constraint of availability is set to 0.1, and since there are 10 service classes, the average availability of the involved component services that need to fulfill the global constraint is $(0.1)^{1/10} \approx 0.8$.

Therefore, it results in high reduction rate when combined with the constraint reduction using response time attribute. Another observation is that, when these two pruning methods are combined (*DROPreprocess*), it can achieve a reduction rate that is greater than any individual pruning method applies alone.

E2: We compare the computation time with respect to the number of services for each service class, which varies from 500 to 5000. The experiment results can be found in Figure 4.6(a).

Results. We notice that both *DRO* and *DROSkyline* have significant improvement over *Exact* and *KMeans* methods. Since *DROSkyline* and *KMeans* are both using the same preprocessing method (non-skyline pruning), therefore, this result suggests that service ranking and dynamic service selection of *DRO* have provided faster performance over *KMeans*. Interestingly, although substantial number of services have been pruned by constraint pruning for *DRO*, *DRO* has only achieved slight improvement of performance over *DROSkyline*. This result suggested that the services that are pruned by constraint pruning are ranked lower by service ranking algorithm, and services that are part of the feasible selection are ranked higher. Therefore, the dynamic selection algorithm could effectively locate a feasible selection by using a significant smaller subset of the services in a service class.

E3: We compare the accuracy with respect to the number of services for each service class, which varies from 500 to 5000. The experiment results can be found in Figure 4.6(b).

Results. We notice that all methods achieve accuracy higher than 90%. For *DRO* and *DROSkyline*, both have outperformed *KMeans* by achieving almost 100% accuracy. This result shows that the service ranking algorithm of *DRO* has accurately ranked the service, such that the feasible selection that has been chosen by the dynamic service selection algorithm is near-optimal or optimal selection.

E4: We compare the computation time with respect to the number of services for each

service class, which varies from 500 to 5000, in the case where it does not exist a selection that can satisfy the global constraints. The experiment results can be found in Figure 4.7.

Results. We notice that *KMeans* has fast-growing computation time. The reason is that *KMeans* applies *k*-means clustering at each round of selection, and it is known that *k*-means clustering is NP-hard in the worst case. Therefore, this makes the method sensitive to the number of services. We also notice that *DROSkyline* outperforms *KMeans* significantly. There are two reasons for this. Firstly, *DROSkyline* only incurs sorting at the start of the selection, and it does not incur extra operations between rounds of selection. Secondly, the number of services that are used for selection increases significantly at each round; therefore, it could effectively explore all services in the service classes in a few number of rounds. Hence, *DROSkyline* has provided significant improvement over *KMeans*. *DROSkyline* is slower than *Exact* due to the extra time that it spent on multiple rounds of service selections before all concrete services in the service classes are explored at the final round – while for *Exact*, it explores all services from the beginning. We also observe that *DRO* outperforms *Exact*. This is attributed to the constraint pruning preprocessing in *DRO*, since most of the services are pruned by constraint pruning due to the strict global constraints that make no feasible selection, and this let *DRO* achieve higher performance than *Exact*.

E5, E6: We compare *DRO* using different termination thresholds with respect to the number of services for each service class, which varies from 500 to 5000, in the case where it exists (E5) and does not exist (E6) a selection that can satisfy the global constraints respectively. The results of experiments E5 and E6 are shown in Figure 4.6(c) and Figure 4.6(d) respectively.

Results. It suggests that the smaller the value of termination threshold, the faster it tends to complete. For example, the dynamic selection algorithm with termination threshold of 0.1 tends to complete faster than the other termination thresholds. This result is due to the fact that the smaller the termination threshold, the fewer elements will be chosen in

each round by the dynamic selection algorithm. We illustrate why this could end up in a faster searching time using an example. Suppose the services that are part of a feasible selection are all ranked at fifth position in their service classes. Dynamic selection algorithm with termination threshold of 0.1 could choose fewer services (say five services) at a single round, while dynamic selection algorithm with termination threshold of 0.9 could choose more services (say ten services) at single round. Although the services selected by dynamic selection algorithm for termination thresholds 0.1 and 0.9 both contain the feasible services, but dynamic selection algorithm for termination thresholds 0.1 will be solved faster by the MIP solver since it contains fewer services. Nevertheless, there is a disadvantage for choosing a smaller termination threshold. Experiment E6 has shown that the dynamic selection algorithm with smaller termination threshold tends to finish slower, since fewer items that are chosen at each round will lead to more rounds to iterate before all concrete services in each service class are explored. It is therefore a tradeoff to choose a smaller termination threshold over a larger one.

Answer to Research Questions. To answer the research questions RQ1–RQ3, we can see that both *DRO* and *DROSkyline* outperform *KMeans* in terms of performance, accuracy and scalability from experiments E1–E4. Research question RQ4 is answered by the analyses of experiments E5 and E6.

Threats to Validity. There are several threats to validity. The first threat of validity is due to the fact that experiment inputs were randomly generated. The second threat of validity is stemmed from our choice to use a few example values as experimental parameters, that include global constraints and termination thresholds, in order to cope with the combinatorial explosion of options. To address these threats, it is clear that more experimentations with different dataset and experimental parameters are required, so that we could further investigate the effects that has not been made obvious by our dataset

and experimental parameters. To mitigate these threats to validity, we further conduct an evaluation on real-world Web services in order to confirm the results with the synthetic dataset.

4.4.2 Evaluation with QWS Dataset

In this evaluation, we use the QWS dataset, which is a public dataset³ collected from public registries, search engines and service portals, using a specialized Web crawler. The dataset contains 2507 Web services and there are a total of 9 QoS attributes measured using commercial benchmark tools. More details on the dataset can be found at [17, 18]. In our experiments, there are five service classes, where the concrete services for the service classes are chosen from the QWS dataset. The QoS constraints are set randomly. The experiments (E1' – E3') that we have conducted are listed as follows. For all experiments, we set the termination threshold at 0.9 and assume there exists a feasible selection that could satisfy the global constraints.

E1': We record the number of services that has been pruned using non-skyline method, constraint method, and the combination of both pruning methods (*DROPreprocess*). The experiment results can be found in Figure 4.10.

Results. The non-skyline pruning offers a better pruning result compare to experiment E1. This result is expected, as it is known that non-skyline pruning does not work well in anti-correlated dataset that is used in experiment E1.

E2', E3': We compare the computation time and accuracy with respect to the number of services for each service class, which varies from 500 to 2500. The experiment results can

³<http://www.uoguelph.ca/~qmahmoud/qws/>

be found in Figure 4.11(a) and Figure 4.11(b) respectively.

Results. We observe similar tendency compared to experiments E2 and E3, where both *DROSkyline* and *DRO* provide better performance and accuracy over *KMeans* and *Exact* methods, and *DROSkyline* and *DRO* have similar performance and accuracy.

As a conclusion, the results of the evaluation with real-world Web services conform to the results with the synthetic dataset.

4.5 Related Work

The problem of QoS-aware Web service selection and composition has received considerable attention during recent years. In [170], [171], the authors present an approach that makes use of global planning to search dynamically for the best concrete services for service composition. Their approach involves the use of mixed integer programming (MIP) techniques to find the optimal selection of component services. Ardagna *et al.* [35] extend the MIP methods to include local constraints. Cardellini *et al.* [49] propose a methodology to integrate different adaptation mechanisms for combining concrete services to an abstract service, in order to achieve a greater flexibility in facing different operating environments. Our work is orthogonal to aforementioned works, as it does not assume particular formulation of the MIP problems. Although the method in aforementioned works efficiently for small case studies, it suffers from scalability problems when the size of the case studies becomes larger, since the time required grows exponentially with the size of concrete services.

Yu *et al.* [168] propose a heuristic algorithm that could be used to find a near-optimal solution. The authors propose two QoS compositional models, a combinatorial model and a graph model. The time complexity for the combinatorial model is polynomial, while the

time complexity for the graph model is exponential. However the algorithm does not scale with the increasing number of Web services. To address this problem, Alrifai *et al.* [22] present an approach that prunes the search space using skyline methods, and they make use of a hierarchical k -means clustering method for representative selection. The work of Alrifai *et al.* is the closest to ours. Our approach has several advantages over their approach. Firstly, their work does not take into account of provided global constraints for representative selection. Therefore, it does not scale well with respect to the number of attributes, and the performance can be significantly degraded by providing restrictive constraints. Secondly, making use of k -means clustering for the purpose of representative selection can be expensive since the operation is NP-hard in general, while in our work, the worst-case performance of representative selection is bounded by $O(n \cdot \log n)$, where n is the total number of concrete services.

Dionysis *et al.* [37] propose a method that allows users to specify their perception of quality in terms of user-defined quality model. Their method is based on k -means approach to match the user defined quality model to the search engine's quality model automatically. Their work focuses on providing intuitive quality abstraction, and is not related to the optimal selection of services. Stephan *et al.* [165] propose a QoS-based service ranking and selection approach. Their approach ranks the services according to their satisfactory scores and selects the optimal service that satisfies users' QoS requirements. Raed *et al.* [104] propose a method that makes use of analytical network process (ANP) to calculate the weight associated with each QoS attribute and rank the service based on users' satisfaction degrees. [165] and [104] could only be used to choose for a single optimal service that could satisfy the users' QoS requirements. In contrast, our work aims to select a set of services that are optimal for a service composition. In our previous work [153], we have proposed an approach to synthesize the local time requirement for component services given the global time requirement of composite service. For the component services that satisfy the local

time requirement, it is guaranteed to satisfy the global time requirement of the composite services. In contrast, in this work, we focus on the set of component services that could not only satisfy the global QoS constraint, but also provide the overall optimal QoS for the composite service.

4.6 Chapter Summary

In this chapter, we have addressed the problem of QoS service composition by proposing a new technique, namely the dynamic ranking optimization (DRO). The technique considerably improves the current service selection approaches, by considering only a subset of representatives that are likely to succeed, before exploring a larger search space. The full search space will be explored only if all the smaller search spaces have failed to produce a result. The evaluation has shown great improvement over the existing methods.

In future, we consider selecting services in the cloud computing environment, as it is known that each cloud computing environment has more components involved. Quinton *et al* [134] propose to model the cloud computing using Software Product Lines principles (SPL). Therefore, we propose the work in the next chapter to select a set of competing features for SPL as a start.

Chapter 5

Optimizing Selection of Competing Features via Feedback-directed Evolutionary Algorithms

To reduce development costs, shorten development cycles, and improve flexibility and reusability, industries usually need to develop and maintain a set of similar products in a systematic and reuse-based way [101]. In software family or Software Product Line (SPL) [103], feature model is proposed to model commonalities and competing variabilities among similar yet different products. Based on the feature model, different features are carefully selected to meet the requirements of customers and to avoid possible conflicts or compatibility problems [127]. In the era of a thriving market of mobile and serviced-based applications, vendors are required to continually reconfigure their applications promptly, to retain and extend their customer base. Therefore, it is desirable to automatically derive features that could meet the requirements of customers, and avoid all possible conflicts of

features.

Feature model provides a representation of software product lines (SPLs), that could be used to facilitate the reasoning and configuration of SPLs [103]. Common SPLs consist of hundreds or even thousands of features. For instance, as reported in [140], the Linux X86 kernel contains 6888 features, and 343944 constraints. In addition, the features are usually associated with quality attributes such as cost and reliability. This complexity provides challenges for the reasoning and configuration of feature models. It is hard for the vendor to select a set of features that complies with the feature model, and meanwhile optimizes the quality attributes according to user preferences. This is called the *optimal feature selection* problem [93].

Existing works [93, 138, 136, 137] have adopted the evolutionary algorithms (EAs) for feature selection with resource constraints and product generation based on the value of user preferences, respectively. Guo *et al.* [93] proposed a genetic algorithm (GA) approach for tackling the optimal feature selection problem. In their work, a repair operator is used to fix each candidate solution, so that it is fully compatible with the feature model after each round of crossover and mutation operations. This approach might be non-terminating, and furthermore, it does not take advantage of the automatic correction that brought by the GA. In addition, GA combines all objectives into a single fitness function with respective weights. This only gives users a solution that is specific to the weights used in the objective formula.

To address this problem, Sayyad *et al.* [138, 136] proposed an approach that uses EAs that support multi-objective optimization, and a range of optimal solutions (i.e., a Pareto front) is returned to the user as a result. They investigated seven EAs and discovered that the Indicator-Based Evolutionary Algorithm (IBEA) [172] yields the best results among the seven tested EAs in terms of time, correctness and satisfaction to user preferences. In [137], they made use of static method to prune features before execution of IBEA for

reducing search space. They also introduced a “seeding method” by pre-computing a correct solution, which was subsequently used by IBEA to generate more correct solutions.

Our work complements existing works by introducing a novel feedback-directed mechanism to existing EAs. In our approach, the feature model is first preprocessed based on SAT solving to remove the prunable features, before the execution of an EA. We have shown that we always prune more features compared to pruning method in [137]. During each round of executing EA, the violated constraints would be analyzed. The analyzed results are used as feedback to guide evolutionary operators (i.e., crossover and mutation) for producing offsprings for next round. Our evaluation has shown that our method produces more promising offsprings (that have less violated constraints), which has led to faster convergence and resulted in more valid solutions in a significantly shorter amount of time.

We make use of both SPLOT [122] and LVAT [5] repositories to evaluate our work. SPLOT is a repository of feature models used by many researchers as a benchmark, and LVAT contains the real-world feature models which have large feature sizes, including the aforementioned Linux X86 kernel model which contains 6888 features.

My contribution for this Chapter is summarized here. We introduce a feedback-directed mechanism into existing EAs. In a feedback-directed EA, solutions are analyzed by their violated constraints. The information is used as feedback for evolutionary operators to produce offsprings that are more likely to satisfy more constraints.

Outline. Section 5.1 introduces the background of this chapter. Section 5.2 presents our feedback-directed EA. Section B.1 provides the evaluation of our approach. Section 5.3 reviews related works. Finally, Section 5.4 concludes this chapter.

5.1 Background

In this section, we provide the background knowledge on software product line, feature model, and multi-objective optimization problem.

5.1.1 Software Product Line

Software product line engineering (SPLE) is architecture-centric and feature-oriented, as SPLE adopts feature-oriented domain analysis [103] for requirements analysis and builds core assets architecture for reuse [59]. Technically, SPLE is a two-phase approach composed of domain engineering and application engineering. The task of domain engineering is to build the software product line (SPL) architecture consisting of a core-asset base and the variant features, while the application engineering focuses on derivation of new products by different customizations of variant features applied onto the core-asset base. Thus, automation of processing and verification of product derivation is a fundamental problem in SPLE. Exploring an efficient and scalable approach for the optimal feature selection problem is critical to the success of SPLE.

5.1.2 Feature Model and its Semantics

The concept of feature model in domain engineering is to represent the features within the product family as well as the structural and semantic (require or exclude) relationships between those features [103]. Since the proposal of SPL, feature model has even been characterized as “the greatest contribution of domain engineering to software engineering” [61].

A feature model is a tree-like hierarchy of features. The structural and semantic relationships between a parent (or compound) feature and its child features (or subfeatures) can

be specified as:

- *Alternative* – If the parent feature is selected, only one among the exclusive subfeatures should be selected,
- *Or* – If the parent feature is selected, at least one or at more all subfeatures must be selected,
- *Mandatory* – A mandatory feature must be selected if its parent is selected,
- *Optional* – An optional feature is optional to be selected.

Besides the above structure or parental relationships between features, cross-tree constraints (CTCs) are also often adopted to represent the mutual relationship for features across the feature model. There are three types of common CTCs:

- f_a *requires* f_b – The inclusion of feature f_a implies the inclusion of feature f_b in the same product.
- f_a *excludes* f_b – The inclusion of feature f_a implies the exclusion of feature f_b in the same product, and vice versa.
- f_a *iff* f_b – The inclusion of feature f_a implies the inclusion of feature f_b in the same product, and vice versa.

In Figure 5.1, the feature model of a Java Chat System (JCS) is illustrated. The root feature of the feature model is *Chat*, which has a mandatory subfeature (*Output*) and several optional subfeatures (e.g., *Encryption*). Since the feature *Output* is mandatory, exactly one of its subfeatures (*GUI*, *CMD*, and *GUI2*) must be selected. In addition, if the *Encryption* feature is selected, at least one of its subfeatures (*Caesar* and *Reverse*) needs to be selected. There is

a CTC for JCS which is of the form $f_a \text{ iff } f_b$ – *Encryption_OR* is selected if and only if *Caesar* or *Reverse* is selected.

The feature model listed in Figure 5.1 can be captured by the constraints that are listed in Table 5.1. The constraints are specified according to the semantics of feature model. Constraint c(1) specifies that the root feature must be present, to prevent a trivial feature model with no selected feature. Constraint c(2) specifies the mandatory feature *Output* and constraints c(3) – c(7) specify constraints on the other five optional subfeatures. The subfeatures of *Output* are in an *Alternative* relationship. This is specified using constraints c(8) – c(11). Constraint c(8) states that *Output* is selected, if and only if at least one of *CMD*, *GUI* and *GUI2* is selected. Constraints c(9) – c(11) specify that at most one feature from *CMD*, *GUI* and *GUI2* can be chosen. The subfeatures of *Encryption* are in *Or* relationship. The constraint c(12) denotes if *Encryption* is selected, then at least one feature from *Caesar* and *Reverse* needs to be selected, and vice versa. The only CTC of JCS is captured in the constraint c(13). Constraints c(1) – c(12) are called *tree constraints*, since they are related to the tree structure of the feature model. Henceforth, given a feature model M , we simply refer tree constraints and CTCs of the M , as the *constraints* of M . We denote the conjunction of constraints of M as $\text{conj}(M)$. We use $\text{Fea}(M)$ to denote the set of entire features of the feature model M . For the JCS example, $\text{Fea}(\text{JCS}) = \{\text{Chat}, \dots\}$ and $|\text{Fea}(\text{JCS})|=12$.

Definition 4 (feasible feature set). *Given a feature model M , a feasible feature set for M is a non-empty feature set $F \subseteq \text{Fea}(M)$, such that F satisfies the constraints of M .*

We write $F \models M$ if $F \subseteq \text{Fea}(M)$ is a feasible feature set of the feature model M .

Example. We use JCS as an example. $F = \{\text{Chat}, \text{Output}, \text{GUI}\}$ is a feasible feature set of JCS, i.e., $F \models \text{JCS}$.

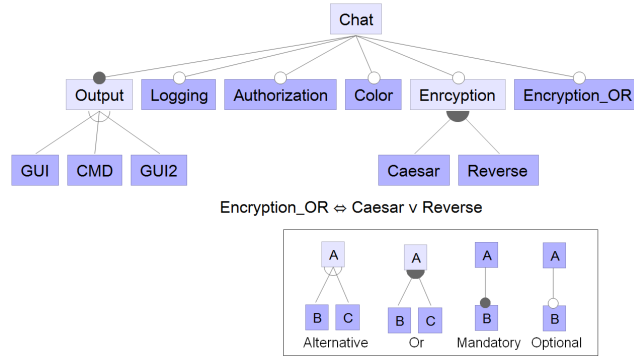


Figure 5.1: The feature model of *JCS*

<i>Chat</i>	c(1)
<i>Output</i> \iff <i>Chat</i>	c(2)
<i>Logging</i> \implies <i>Chat</i>	c(3)
<i>Authorization</i> \implies <i>Chat</i>	c(4)
<i>Color</i> \implies <i>Chat</i>	c(5)
<i>Encryption</i> \implies <i>Chat</i>	c(6)
<i>Encryption_OR</i> \implies <i>Chat</i>	c(7)
$(GUI \vee CMD \vee GUI2) \iff Output$	c(8)
$\neg(GUI \wedge CMD)$	c(9)
$\neg(GUI \wedge GUI2)$	c(10)
$\neg(CMD \wedge GUI2)$	c(11)
$(Caesar \vee Reverse) \iff Encryption$	c(12)
$Encryption_OR \iff (Caesar \vee Reverse)$	c(13)

Table 5.1: Constraints of *JCS*

5.1.3 Multi-objective Optimization Problem

Many real-world problems have multiple objectives that need to be optimized simultaneously. However, these objectives usually conflict with each other, which prevents optimizing all objectives simultaneously. A remedy is to have a set of optimal trade-offs between the conflicting objectives.

A k -objective optimization problem could be written in the following form:

$$\begin{aligned}
 \text{Minimize } Obj(F) &= (Obj_1(F), Obj_2(F), \dots, Obj_k(F)) \\
 &\text{subject to } F \models M
 \end{aligned}
 \tag{5.1}$$

where $Obj(F)$ is a k -dimensional objective vector for F and $Obj_i(F)$ is the value of F for i th objective.

Given $F_1, F_2 \models M$, F_1 can be viewed as better than F_2 for the minimization problem in Equation 5.1, if Equation 5.2 holds.

$$\forall i : Obj_i(F_1) \leq Obj_i(F_2) \wedge \exists j : Obj_j(F_1) < Obj_j(F_2) \quad (5.2)$$

where $i, j \in \{1, \dots, k\}$.

In such a case, we say that F_1 *dominates* F_2 . F_1 is called a *Pareto-optimal solution* if F_1 is not dominated by any other $F \models M$. We denote all Pareto-optimal solutions as the *Pareto front*.

Many evolutionary algorithms (e.g., IBEA [172], NSGA-II [64], ssNSGA-II [72], MO-Cell [126]) are proposed to find a set of non-dominated solutions that approximate the Pareto front for solving the multi-objective optimization problem. **Problem Statement.** Our work addresses the *optimal feature selection*, which aims at searching for feasible feature sets that approximate the Pareto front to solve the multi-objective optimization problem.

5.2 Feedback-directed Evolutionary Algorithm

In this section, we elaborate our approach in addressing the optimal feature selection problem. First, we introduce a preprocessing method to filter out prunable features before the execution of an EA, in order to reduce the search space. Second, we illustrate feedback-directed evolutionary operators that are used in this chapter to guide an EA for the optimal feature selection.

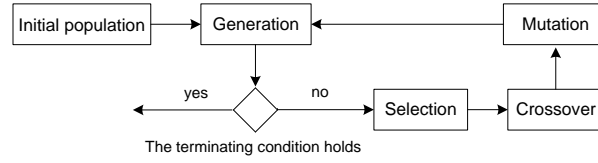


Figure 5.2: Typical flow of evolutionary algorithms

5.2.1 Preliminaries of Evolutionary Algorithms

Evolutionary algorithms (EAs), inspired by the “survival of the fittest” principle of the Darwinian theory of natural evolution, are stochastic search methods based on principles of the biological evolution. By applying the EA, a problem is encoded into a simple chromosome-like data structure, and then evolutionary operators (e.g., selection, crossover, and mutation) are applied on these data structures to preserve “the fittest” information, which is analogous to “survival of the fittest” in the natural world. EAs often perform well in approximating solutions, and therefore EAs are typically suitable for the optimization problems especially if the search space of the problem is large and complex.

A typical workflow of EAs is described in Figure 8.4. An EA begins with an initial generation of chromosomes, which we denote as *initial population*. Typically, the initial population is generated randomly. Evolutionary operators are then applied on a generation to evolve into a new generation of chromosomes. Different EAs have different dominating criteria, which will be introduced in Section B.1.1. The chromosomes that are ranked higher according to the dominating criteria of the EA have a higher chance to proceed to the next generation. The evolutionary process continues until the termination condition is met. An example of the termination condition might be that the number of generations exceeds a predefined upper bound $n \in \mathbb{Z}_{>0}$.

5.2.2 Preprocessing of Feature Model

In the following, we introduce the features that could be pruned from $Fea(M)$ before the execution of an EA. By doing this, the search space of the EA would be reduced, which could make the optimal feature selection more efficient.

Our approach of preprocessing is by exploiting the *commonalities* [46] of the products. Observed that some features must be present in all products derived from M . For example in JCS , the feature set $\{Chat, Output\}$ is shared by all derived products, and we call these features as *common features*. Similarly, we call the set of features that must not be used in all derived products as *dead features*. Dead features do not present in JCS but they are common in feature models of real systems (e.g., Linux X86 kernel, FreeBSD operating system and eCos operating system). Henceforth, we denote common features and dead features as F_c and F_d respectively, where $F_c, F_d \subseteq Fea(M)$, and $F_c \cap F_d = \emptyset$. The preprocessed features that are passed to the execution of EAs is $Fea(M) \setminus (F_c \cup F_d)$, and we denote $F_c \cup F_d$ as *prunable features*.

The function *PrunableFeatures* (Algorithm 6) is used to find common and dead features. Recall that $conj(M)$ represents the conjunction of all tree constraints and CTCs of feature model M , and SAT is a function that is used to check the satisfiability of the constraints. Note that SAT function is readily provided by many off-the-shelf SAT solvers (e.g., SAT4J [9]). We assume that $conj(M)$ is satisfiable, i.e., there exists at least a valid product from the feature model M . If $conj(M) \wedge \neg f$ is unsatisfiable (line 4), it implies that feature f must exist in all derived products of M . Therefore, feature f is added to common features F_c (line 5). This is similar to the detection of dead features in lines 6 and 7.

Algorithm 6: PrunableFeatures

input : Feature model M
output: Common features $F_c \subseteq Fea(M)$
output: Dead features $F_d \subseteq Fea(M)$

```
1  $F_c \leftarrow \emptyset$ ;  
2  $F_d \leftarrow \emptyset$ ;  
3 foreach  $f \in Fea(M)$  do  
4   if  $\neg SAT(conj(M) \wedge \neg f)$  then  
5      $F_c = F_c \cup f$ ;  
6   else if  $\neg SAT(conj(M) \wedge f)$  then  
7      $F_d = F_d \cup f$ ;  
8 return  $(F_c, F_d)$ ;
```

5.2.3 Genetic Encoding of the Feature Set

The selected features of a feature model is encoded using an array-based chromosome as shown in Figure 8.7b. Given a chromosome of length n , array indices are numbered from 0 to $n - 1$. Each feature is assigned with an array index starting from 0. Each value on the chromosome ranges over $\{0, 1\}$, where 0 (resp. 1) represents the absence (resp. presence) of the feature. Given a feature model M , we define a function $f_M : Fea(M) \rightarrow \{\mathbb{Z}, \perp\}$ that maps each feature f of the feature model M to an array index. $f_M(f_1) = \perp$ denotes that there is no array index that is assigned for the feature f_1 . Similarly, we define $f_M^{-1} : \mathbb{Z} \rightarrow Fea(M)$ as a function that maps a given array index to the feature it represents.

Example. We show how a feature set on the *JCS* is encoded. Note that features *Chat* and *Output* have been pruned by the preprocessing algorithm in Algorithm 6; therefore, they are not contained in the chromosome (i.e., $f_M(Chat) = f_M(Output) = \perp$). The features are indexed level by level, and their indexes have been listed in Figure 8.7b (e.g., $f_M(Logg-ing) = 0$). The chromosome in Figure 8.7b represents the feature set $\{Encryption, GUI, Caesar, Reverse\}$.

5.2.4 Feedback-directed Evolutionary Operators

The violated constraints of a chromosome C_i provide an important clue on which features on the chromosome C_i need to be modified. If we focus on these features, we may converge faster on the optimal feature selection.

We incorporate this feedback into the crossover and mutation operations, which are the main evolutionary operators common for almost all EAs. The feedback-directed crossover and mutation operators provide an effective guidance for EAs to perform the optimal feature selection.

5.2.4.1 Feedback-directed Mutation

The objective of *mutation* operator is to change some values in a selected chromosome leading to additional genetic diversity to help the search process escape from local optimal traps.

We introduce how the *feedback-directed mutation* operator works. Before the mutation, the feedback-directed mutation analyzes the selected chromosome on the violated constraints. We denote the corresponding positions on the chromosomes for the features that are contained in the violated constraints as *error positions*.

Example. We illustrate the feedback-directed mutation operator, using the *JCS* example shown in Figure 8.7b. Given the values of the chromosome as shown in Figure 8.7b, we can easily check that it violates the constraint c_{13} . The constraint c_{13} contains three features, which are *Encryption_OR*, *Caesar*, and *Reverse*. The corresponding array positions of these three features are shaded on the chromosome in Figure 8.7b. These shaded positions are the *error positions*.

The algorithm *FMutation* for feedback-directed mutation are given in Algorithm 7. At line 1,

an offspring chromosome C is initialized with values in the chromosome P , and $n \in \mathbb{Z}$ is initialized with the length of the chromosome P (line 2). At line 3, $Err \in \mathcal{P}(\mathbb{Z})$ is assigned with the set of integers that is returned from $ErrPos(C)$ (which will be introduced later). The set of integers returned by $ErrPos(C)$ represents the error positions on the chromosome C . Each position on the chromosome is iterated (line 4). The function $rand(a, b)$ (resp., $randInt(a, b)$), with $a > b$, chooses a real (resp., integer) number between numbers a and b . At line 5, if the current position i is an error position, and the random number is less than the error mutation probability P_{emut} , then the value in the i th-position on the chromosome is mutated by randomly choosing an integer between 0 and 1 (line 7). On the other hand, if the position does not belong to any error position, and the random number is less than P_{mut} (line 6), the value in the i th-position is mutated. Note that the probability P_{emut} will be set with a value that is far larger than P_{mut} , so that the mutation occurs more frequently on error positions. For P_{emut} and P_{mut} , example values could be 1.0 and 0.0000001. Note that we set P_{mut} much lower than classic mutation probability (e.g. 0.001-0.05 [144]). This is because lower P_{mut} with higher P_{emut} would lead to faster convergence, since it allows faster correction of constraint violations by minimizing the changes of non-error positions and focusing on the changes of error positions. This is demonstrated in Section B.1.3.

We now introduce the $ErrPos$ function described in Algorithm 8. At line 1, $ePos$ is initialized with an empty set. The valuation function $\Pi : Fea(M) \rightarrow \{true, false\}$ (line 3) maps each feature f of the feature model M to a Boolean value that denotes whether the corresponding feature is selected. The mappings in Π are populated according to the values on the chromosome (line 5). Subsequently, prunable features are added to the mappings in Π with value *true* (line 7), since they must belong to any feasible feature set of feature model M as explained in Section 5.2.2. At line 9, $\Pi \not\models constraint$ holds iff replacing each feature f contained in the *constraint* with $\Pi(f)$ evaluates to false. In other words, $\Pi \not\models constraint$ means that the selection represented by chromosome C violates the constraint *constraint*.

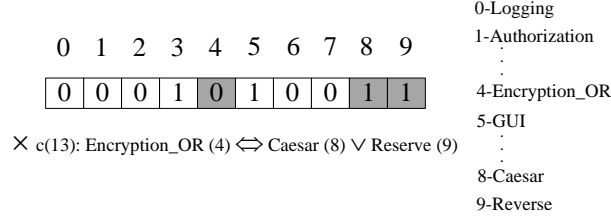


Figure 5.3: Feedback-directed mutation operator

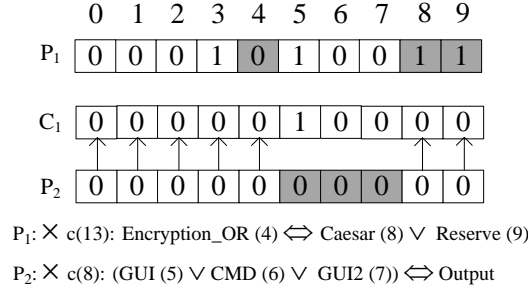


Figure 5.4: Feedback-directed crossover operator

In such a case, the function $\text{getFeatures}(c)$ is used to get the features that are contained in the constraint c (line 10). For example, given the constraint $c(13)$ in Table 5.1 for JCS as an input, getFeatures will return $\{4, 8, 9\}$. These array indexes that represent the error positions will be included in $ePos$.

5.2.4.2 Feedback-directed Crossover

The crossover operation is used to generate offsprings by exchanging values in a pair of chromosomes chosen from the population, and it happens with a probability P_{cross} (the crossover probability). The feedback-directed crossover operator uses values in the non-error positions to crossover. The objective for using values from non-error positions is to pass the “good genes” to offsprings.

Example. We demonstrate the feedback-directed crossover operator, using the JCS example shown in Figure 8.7a. Suppose the chromosomes P_1 and P_2 have violated constraints $c(13)$ and $c(8)$ respectively. The offspring chromosome C_1 is first initialized as the same

Algorithm 7: FMutation

input : Chromosome P
input : Error mutation probability P_{emut}
input : Mutation probability P_{mut}
output: Chromosome C

```
1  $C \leftarrow P$ ;  
2  $n \leftarrow |P|$ ;  
3  $Err \leftarrow ErrPos(C)$ ;  
4 for  $i = 0$  to  $n - 1$  do  
5   if  $(i \in Err \wedge rand(0, 1) < P_{emut}) \vee$   
6    $(i \notin Err \wedge rand(0, 1) < P_{mut})$  then  
7      $C[i] \leftarrow randInt(0, 1)$ ;  
8 return  $C$ ;
```

values with the chromosome P_1 . We now show that how the feedback-directed crossover is performed. The values from non-error positions of the chromosome P_2 are copied to the chromosome C_1 (shown by the arrows). This results in the chromosome C_1 that is shown in Figure 8.7a. Note that, in the given example, chromosome C_1 is now representing a feasible feature selection after the feedback-directed crossover. The production of the chromosome C_2 (not shown in the graph) is symmetric to the production of the chromosome C_1 .

The algorithm *FCrossover* of feedback-directed crossover operator is given in Algorithm 12. The chromosomes C_1 and C_2 are initialized with the values from chromosomes P_1 and P_2 respectively (lines 1, 2). If the generated random number is smaller than the crossover probability P_{cross} (line 4), then it will perform the crossover operation. First, it verifies whether there exists any error position in chromosomes P_1 and P_2 , by checking whether the size of their error positions is greater than 0 (line 5). If it is, then the feedback-directed crossover will be performed. The algorithm iterates through the chromosome (line 6), and copies the values of non-error positions from chromosome P_1 (resp., P_2) to the corresponding positions in chromosome C_2 (resp., C_1) (lines 7–10).

Algorithm 8: ErrPos

input : Chromosome C
input : A set of constraints $constraints$
output: A set of integers $ePos$

```
1  $ePos \leftarrow \emptyset$ ;  
2  $n \leftarrow |C_1|$ ;  
3  $\Pi \leftarrow \emptyset$ ;  
4 for  $i = 0$  to  $n - 1$  do  
5    $\Pi \leftarrow \Pi \cup \{f_M^{-1}(i) \mapsto (C[i] \neq 0)\}$ ;  
6 foreach  $feature \in F_m$  do  
7    $\Pi \leftarrow \Pi \cup \{F_m \mapsto true\}$ ;  
8 foreach  $constraint \in constraints$  do  
9   if  $\Pi \not\models constraint$  then  
10     $ePos \leftarrow ePos \cup getFeatures(constraint)$ ;  
11 return  $ePos$ ;
```

Otherwise, if both chromosomes P_1 and P_2 do not have any error position, the classic single point crossover operator is applied. First, an array index, $crossIndex \in \{0, \dots, n - 1\}$, is randomly selected. Subsequently, all values starting from position $crossIndex$ are copied from chromosome P_2 (resp., P_1) to chromosome C_1 (resp., C_2) (lines 12–15).

5.3 Related Work

Our work is related to the feasible feature selection. In [162], White *et al.* reduced the feature selection problem in SPL to a multidimensional multi-choice knapsack problem (MMKP). They proposed a polynomial time approximation algorithm, called Filtered Cartesian Flattening (FCF), to derive an optimal feature configuration subject to resource constraints. Their evaluation showed that FCF can stably achieve the optimality above 90% even when the number of resources increases up to 91, while the optimality of Constraint Satisfaction Problem (CSP) based Feature Selection in [39] drops down to 30% when there are 91 resources.

Algorithm 9: FCrossover

```
input : Chromosome  $P_1$ 
input : Chromosome  $P_2$ 
input : Crossover probability  $P_{cross}$ 
output: Chromosomes  $C_1, C_2$ 

1  $C_1 \leftarrow P_1$ ;
2  $C_2 \leftarrow P_2$ ;
3  $n \leftarrow |P_1|$ ;
4 if  $rand(0, 1) < P_{cross}$  then
5   if  $|ErrPos(P_1)| > 0 \wedge |ErrPos(P_2)| > 0$  then
6     for  $i = 0$  to  $n - 1$  do
7       if  $i \notin ErrPos(P_1)$  then
8          $C_2[i] \leftarrow P_1[i]$ ;
9       if  $i \notin ErrPos(P_2)$  then
10         $C_1[i] \leftarrow P_2[i]$ ;
11   else
12      $crossIndex \leftarrow randInt(0, n-1)$ ;
13     for  $i = crossIndex$  to  $n-1$  do
14        $C_1[i] \leftarrow P_2[i]$ ;
15        $C_2[i] \leftarrow P_1[i]$ ;
16 return  $(C_1, C_2)$ ;
```

Although FCF in [162] can achieve a highly optimal solution, but it requires significant computing time. To address the problem of scalability, Guo *et al.* [93] presented GAFES (a genetic algorithm based approach). The rationale is that GAs are quite suitable for the highly constrained problems, such as the feature selection (product derivation) problem. GAFES integrated a new *repair* operator for feature selection and also defined a *penalty* function for resource constraints. The evaluation showed GAFES may not beat the FCF and CSP in optimality, but it scaled up to large-scale models with a reasonable optimality.

Genetic algorithm only allows single objective function, and in addition, the method proposed in [93] repairs each solution explicitly, and does not take advantage of the evolution of GA algorithm for repairing. To address this problem, Sayyad *et al.* [138] investigated the use of different types of EAs that support multi-objective function for the optimal feature

selection. They adopted 7 types of EAs, such as IBEA, NSGA-II and MOCell, to search for the optimal product. The results have shown that IBEA performs much better than other 6 EAs in terms of time, correctness and satisfaction to user preferences. In [136], Sayyad *et al.* improved [138] by turning down the crossover probability from 0.9 to 0.1 and mutation probability from 0.05 to 0.01, and they reported HV-mean and spread mean may increase by 5% to 10% in most cases. In [137], Sayyad *et al.* proposed the use of EA with simple heuristic in larger product lines from LVAT repository. They proposed the use of static analysis to identify prunable features for reducing search space, and the use of seeded techniques to find more correct products from Linux X86 Kernel. Our method has improved the method proposed in [138, 136, 137] by incorporating feedback-directed mechanism for EAs (cf. Section 3.4 for the evaluation). We also show that our method for finding prunable feature with Algorithm 6 is always not lesser than the method proposed in [137] (cf. Section B.1.1.3 for the explanation and evaluation).

Our work is relevant to searching valid features for a feature model. In [132], an experiment for measuring the efficiency of BDD, SAT, and CSP solvers is conducted using feature models from SPLOT repository. They reported long run times for certain operations, and certain runs are cancelled if exceeded three hours. They also reported an exponential runtime increase with the number of features for non-BDD solvers on the “valid” operations. In [133], the state-of-art solvers, e.g., JavaBDD BDD solver, the JaCoP CSP solver and the SAT4J SAT solver, were used to answer the questions such as “derive one valid product from a feature model” and “number of products”. They found that CSP and SAT solvers have exponential runtime increase as the feature size of feature model increases, and BDD requires a maximum of 28 seconds to derive a valid product for web-portal, even without considering the quality of feature attributes. Thus, these automated reasoning techniques can be precise, but generally not scalable for large feature models. Our work complements with their work by using feedback-directed evolutionary algorithm that

scales well for large feature models. In [94] introduces five novel parallel algorithm for Multi-Objective Combinatorial Optimization (MOCO) to allow parallel processing. Our work complements with them by considering feedback-directed mechanism for MOCO problem using feedback-directed EAs.

Our work is also related to the feedback-directed methods in software engineering. Pacheco *et al.* [129] proposed RANDOOP, a feedback-directed mechanism for performing random test. It uses erroneous results of previous method invocation to generate a better random test. Clarke *et al.* [57] proposed CEGAR, which uses spurious counterexamples as a feedback to guide the refinement process. Our method is on feedback-directed methods in EAs for the optimal feature selection.

In addition to the SPL domain, multi-objective evolutionary optimization algorithms (MEOAs) have also been applied to various software engineering problems. In [99], Harman *et al.* proposed the term Search-Based Software Engineering (SBSE), and reported that the surveyed and proposed optimization techniques for SE problems by 2001 were all single-objective based. Seeing the potential of using multi-objective optimization, Harman [98] discussed about the possible usage of the meta-heuristic search techniques such as: simulated annealing and genetic algorithm. Harman considered it insensible combination of multiple metrics into an aggregate fitness in the way of assigning coefficients, and further suggested to use Pareto optimality rather than aggregate fitness.

5.4 Conclusion

In this work, we have presented a novel technique by introducing a feedback-directed mechanism into various EAs. Our approach is based on analyzing violated constraints, and uses the analyzed results as a feedback to guide the process of crossover and mutation operators. In addition, we also introduce a preprocessing technique to reduce the search

space, by filtering away the prunable features in all feasible feature sets. Our evaluation shows that both the preprocessing technique and the feedback-directed mechanism have improved over existing unguided EAs on the optimal feature selection. Without compromising on running time, the feedback-directed IBEA successfully found 72.33% and 75% more correct solutions for case studies in SPLOT and LVAT repositories, compared to the unguided IBEA. In addition, with “seeding method” proposed by [137] and feedback-directed IBEA, we have reduced the running time from about 3.5 – 4 *hours* to less than 40 *seconds* for finding 34 correct solutions.

Chapter 6

Verification of Functional and Non-functional Requirements of Web Service Composition

A real-world business process may contain a set of services. A Web service is a single autonomous software system with its own thread of control. A fundamental goal of Web services is to have a collection of network-resident software services, so that it can be accessed by standardized protocols and integrated into applications or composed to form complex services which are called *composite services*. A composite service is constructed from a set of *component services*. Component services have their interfaces and functionalities defined based on their internal structures. The *de facto* standard for Web service composition is Web Services Business Process Execution Language (WS-BPEL) [102]. WS-BPEL is an XML-based orchestration business process language. It provides basic activities such as service invocation, and compositional activities such as sequential and parallel composition to describe composition of Web services. BPEL is inevitably rich in concur-

rency and it is not a simple task for programmers to utilize concurrency as they have to deal with multi-threads and critical regions. It is reported that among the common bug types concurrency bugs are the most difficult to fix correctly, the statistic shows that 39% of concurrency bugs are fixed incorrectly [166]. Therefore, it is desirable to verify Web services with automated verification techniques, such as model checking [58].

There are two kinds of requirements of Web service composition, i.e., functional and non-functional requirements. Functional requirements focus on the functionalities of the Web service composition. Given a booking service, an example of functional requirement is that a flight ticket with price higher than \$2000 will never be purchased. The non-functional requirements are concerned with the Quality of Service (QoS). These requirements are often recorded in service-level agreements (SLAs), which is a contract specified between service providers and customers. Given a booking service, an example of non-functional requirements is that the service will respond to the user within 5 ms. Typical non-functional requirements include response time, availability, cost and so on. However, it is difficult for service designers to take the full consideration of both functional and non-functional requirements when writing BPEL programs.

Model checking is an automatic technique for verifying software systems [58], which helps find counterexamples based on the specification at the design time so that it could detect errors and increase the reliability of the system at the early stage. Currently, increasing number of complex service processes and concurrency are developed on Web service composition. Hence, model checking is a promising approach to solve this problem. Given functional and non-functional requirements, existing works [82, 89, 106, 125, 47, 96] only focus on verification of one aspect, and disregard the other, even though these two aspects are inseparable. Different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework.

In this chapter, we propose a method to verify BPEL programs against combined functional and non-functional requirements. A dedicated model checker is developed to support the verification. We make use of the labeled transition systems (LTSs) directly from the semantics of BPEL programs for functional verification. For non-functional properties, we propose different strategies to integrate different non-functional properties into the functional verification framework. We focus on three important non-functional properties in this chapter, i.e., availability, cost and response time. To verify availability and cost, we calculate them on-the-fly during the generation of LTS, and associate calculated values to each state in the LTS. Verification of response time requires an additional preprocessing stage, before the generation of LTS. In the preprocessing stage, response time tag is assigned to each activity that is participated in the service composition. With such integration, we are able to support combined functional and non-functional requirements.

The contributions of this chapter are summarized as follows.

1. We support integrated verification of functional and non-functional properties for Web service composition. To the best of our knowledge, we are the first work on such integration.
2. We capture the semantics of Web service composition using labeled transition systems (LTSs) and verify the Web service composition directly without building intermediate or abstract models, which makes our approach more suitable for general Web service composition verification.
3. Our approach has been implemented and evaluated on the three real-world case studies, and this demonstrates the effectiveness of our method.

6.1 Motivation Example

In our work, we assume that composite services are specified in the BPEL language. BPEL is the *de facto* standard for implementing composition of existing services by specifying an executable workflow using predefined activities. BPEL is an XML-based orchestration business process language for the specification of executable and abstract business processes. It supports control flow structures such as sequential and concurrency execution. In the following, we introduce the basic BPEL notations. `< receive >`, `< invoke >`, and `< reply >` are the basic communication activities which are defined to receive messages, execute component services and return messages respectively for communicating with component services. There are two kinds of `< invoke >` activities, i.e., synchronous and asynchronous invocation. Synchronous invocation activities are invoked and the process waits for the reply from the component service before moving on to the next activity. Asynchronous invocation activities are invoked and moving on to the next activity directly without waiting for the reply. The control flow of composite services is specified using the activities like `< sequence >`, `< while >`, `< if >` and `< flow >`. `< sequence >` is used to define the sequential ordering structure, `< while >` is used to define the loop structure, `< if >` is used to define the conditional choice structure, and `< flow >` is used to implement concurrency structure.

6.1.1 Computer Purchasing Services (CPS)

In this section, we introduce the computer purchasing service (CPS), which is designed to allow users to purchase a computer online using credit cards. The workflow of CPS is illustrated in Figure 6.1. CPS has four component Web services, namely Personal Billing Service (PBS), Corporate Billing Service (CBS), Manufacture Service (MS), Shipper Service (SS). CPS is initialized upon receiving the request from the customer (*fu*) with the infor-

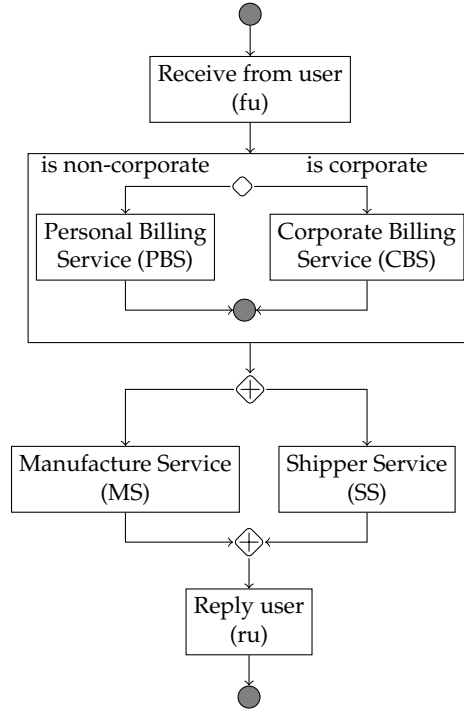


Figure 6.1: Computer Purchasing Service

mation of the customer and the computer that he wishes to purchase for. Subsequently, an $\langle \text{if} \rangle$ activity (denoted by \diamond) is used for checking whether the customer is a corporate customer or non-corporate customer. If it is a corporate customer, CBS is invoked synchronously to bill the corporate customer, otherwise, PBS is invoked synchronously to bill the non-corporate customer with credit card information. Upon receiving the reply, a $\langle \text{flow} \rangle$ activity (denoted by \oplus) is triggered and MS and SS are invoked concurrently. MS is invoked synchronously to notify manufacture department for manufacturing the purchased computers. SS is invoked synchronously to schedule shipment for the purchased computers. Upon receiving the reply message from SS and MS, reply user (ru) is called to return the result of the computer purchasing to the customer.

We assume a property that CPS must fulfill is that it must invoke reply user (ru) within 5 *ms*. Notice that this property combines the functional (must invoke reply user (ru)) and

non-functional (within 5 *ms*) requirements.

6.1.2 BPEL Notations

In order to present BPEL syntax compactly, we define a set of BPEL notations below:

- $rec(S)$ and $reply(S)$ are used to denote “receive from” and “reply to” a service S ;
- $sInv(S)$ (resp. $aInv(S)$) is used to denote synchronous (resp. asynchronous) invocation of a service S ;
- $P_1 || P_2$ is used to denote $\langle \text{flow} \rangle$ activity, i.e., the concurrent execution of BPEL activities P_1 and P_2 ;
- $P_1 \triangleleft b \triangleright P_2$ is used to denote $\langle \text{if} \rangle$ activity, where b is a guard condition. Activity P_1 is executed if b is evaluated to be true, otherwise activity P_2 will be executed;
- $P_1 \rightarrow P_2$ is used to denote $\langle \text{sequence} \rangle$ activity, where P_1 is executed followed by P_2 .

We denote activities that contain other activities as *composite activities*, there are $P_1 || P_2$, $P_1 \triangleleft b \triangleright P_2$ and $P_1 \rightarrow P_2$. For activities that do not contain any other activities, we denote them as *atomic activities*, there are $rec(S)$, $reply(S)$, $sInv(S)$ and $aInv(S)$.

6.2 QOS-AWARE COMPOSITIONAL MODEL

In this section, we define the QoS compositional model used in this chapter and briefly introduce the semantics of BPEL, captured by labeled transition systems (LTSs), and we introduce some definitions used in the semantic model in the following section.

QoS Attribute	PSB	CSB	MS	SS
Response Time(<i>ms</i>)	1	2	3	1
Availability(%)	90	80	80	80
Cost(\$)	3	2	2	2

Table 6.1: QoS Attribute Values

6.2.1 QoS Attributes

In this chapter, we deal with quantitative attributes that can be quantitatively measured using metrics. There are two classes of QoS Attributes, positive and negative attributes. Positive attributes (e.g., availability) have a good effect on the system, and therefore, they need to be maximized. Availability of the service is the probability of the service being available. Negative attributes (e.g., response time, cost) need to be minimized as they have the negative impact on the system. Response time of the service is defined as the delay between sending a request and receiving the response and cost of the service is defined as the money spent on the service. In this chapter, we assume the unit of response time, availability and cost to be millisecond (ms) , percentage (%) and dollar (\$). Table 6.1 shows the information of response time, availability and cost of each component service for CPS example as described in Section 6.1.1.

Given a component service s with n QoS attributes, we use a vector $Q_s = \langle q_1(s), \dots, q_n(s) \rangle$ to represent QoS attributes of the service s , where $q_i(s)$ represents the value of i th attribute of the component service s . Similarly, $Q'_{cs} = \langle q_1(cs)', \dots, q_n(cs)' \rangle$ is used to denote the QoS attributes of the composite service cs , where $q_i(cs)'$ represents the i th attribute of the composite service cs .

6.2.2 QoS for Composite Services

A composite service S is constructed using a finite number of component services to reach a business goal. Let $C = \langle s_1, s_2, \dots, s_n \rangle$ be the set of all component services that are used by S . The QoS of composite services is aggregated from the QoS of the component services, based on the service internal compositional structure, and the type of QoS attributes. Table 7.1 shows the aggregation functions for each compositional structure. We consider three types of QoS attributes: response time, availability and cost. For response time, in sequential composition, the response time of the composite service, is aggregated by summing up the response time of each component service. As for parallel composition, the response time of the composite service is the maximum response time among that of each participating component service. For loop composition, the response time of the composite service is obtained by summing up the response time of the participating component service for k times, where k is the number of maximum iteration of the loop. And for conditional composition, the response time of the composite service is the maximum response time of n participating component services since it is not known that which guard is satisfied at the design phase. For availability, in sequential composition, the availability of the composite service, is the product of that of all component services in the sequence because it means all component services are available during the sequential execution. It is similar to parallel and loop composition for aggregation of availability of the composite services. For conditional availability of the composite service, since one component service will be chosen at execution, therefore, we denote the availability as the minimum availability among all component services participated in the conditional composition. For cost, in sequential composition, the cost of the composite service is decided by the total cost of component services. For the conditional composition, the cost of the composite service is the maximum cost of n participating component services. Other common QoS attribute

QoS Attribute	Sequential	Parallel	Loop	Conditional
Response Time	$\sum_{i=1}^n q(s_i)$	$\max_{i=1}^n q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^n q(s_i)$
Availability	$\prod_{i=1}^n q(s_i)$	$\prod_{i=1}^n q(s_i)$	$(q(s_1))^k$	$\min_{i=1}^n q(s_i)$
Cost	$\sum_{i=1}^n q(s_i)$	$\sum_{i=1}^n q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^n q(s_i)$

Table 6.2: Aggregation Function

types can be aggregated in the similar way with these three attributes. For example, QoS attributes like reliability share the same aggregation function with availability.

6.2.3 Labeled Transition System

The QoS-aware composite model in this chapter is defined using labeled transition system (LTS). In the following we define various terminologies that will be used in this chapter.

Definition 5 (System State). *A system state s is a tuple (P, V, Q) , where P is the composite service process and V is a (partial) variable valuation that maps variables to their values, Q is a vector which represents QoS attributes of the composite service.*

Two states are equivalent iff they have the same process P , the same valuation V and the same QoS vectors Q . Given a system state $s = (V, P, Q)$, $Q = \langle r, a, c \rangle$ is a vector with three elements, where $r, a, c \in \mathbb{R}_{\geq 0}$, and $0 \leq a \leq 1$. r, a, c represents the response time, availability, and cost of the state s . The response time, availability, and cost are calculated from the execution that starts at initial state s_0 up to the state s . Henceforth, we use the notation $Q(\text{ResponseTime})$, $Q(\text{Availability})$ and $Q(\text{Cost})$ to denote the value of r, a , and c of QoS vector Q respectively.

Definition 6 (Composite Service Model). *A composite service model \mathcal{M} is a tuple (Var, P_0, V_0, F) ,*

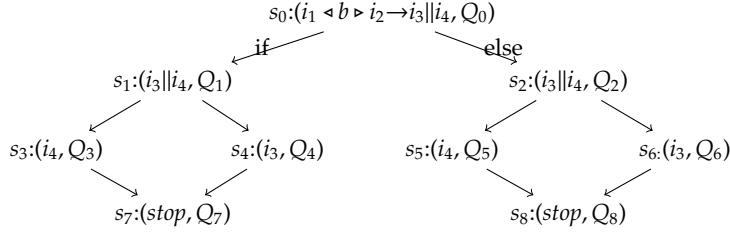
where Var is a finite set of variables, P_0 is the composite service process, and V_0 is an initial valuation that maps each variable to its initial value. F is a function which maps component services to their QoS attribute vectors.

Given a composite service (Var, P_0, V_0, F) , an example of valuation V is $\{var_1 \mapsto 1, var_2 \mapsto \perp\}$, where $var_1, var_2 \in Var$, and $var_2 \mapsto \perp$ is used to denote that var_2 is undefined.

Definition 7 (LTS). An LTS is a tuple $\mathcal{L} = (S, s_0, \Sigma, \delta)$, where

- S is a set of states,
- $s_0 \in S$ is the initial state,
- Σ is a set of actions,
- $\delta : S \times \Sigma \times S$ is a transition relation.

For convenience, we use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in \delta$ and we denote the LTS of a BPEL service \mathcal{M} as $L(\mathcal{M})$. Given a composite service model $\mathcal{M} = (Var, P_0, V_0, F)$, $L(\mathcal{M}) = (S, (P_0, V_0, Q_0), \Sigma, \delta)$. Q_0 is the QoS attribute vector of the initial state, where the availability is 1, cost and response time is equal to 0. Give a state $s \in S$, $Enable(s)$ is denoted as the set of states reachable from s by one transition; formally, $Enable(s) = \{s' | s' \in S \wedge a \in \Sigma \wedge s \xrightarrow{a} s' \in \delta\}$. An execution π of \mathcal{L} is a finite alternating sequence of states and actions $\langle s_0, a_1, s_1, \dots, s_{n-1}, a_n, s_n \rangle$, where $\{s_0, \dots, s_n\} \in S$ and $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$. We denote the execution π by $s_0 \xrightarrow{a_1} s_1 \xrightarrow{\dots} s_{n-1} \xrightarrow{a_n} s_n$. A state s is called reachable if there is an execution that ends in s and starts in an initial state. The LTS of CPS as discussed is shown in Figure 6.2, where we omit the *Receive from user*(fu), *Reply user*(ru), all actions $a \in \Sigma$, and component V in the state for the reason of brevity. From state s_0 , conditional activity $i_1 \triangleleft b \triangleright i_2$ is enabled. Given that $\{b \mapsto \perp\}$, either i_1 or i_2 might be executed, therefore states s_1 and s_2 are evolved from state s_0 . Noted that if guard b is defined, then only one branch is explored in the LTS. From



where i_1 is $sInv(PBS)$, i_2 is $sInv(CBS)$, i_3 is $sInv(MS)$ and i_4 is $sInv(SS)$

Figure 6.2: LTS of CPS

state s_1 , the flow activity $i_3 || i_4$ is enabled, and both activities i_3 and i_4 are allowed to execute. This leads to states s_3 and s_4 respectively. State s_3 evolves into state s_7 after activity i_4 is executed. *stop* activity in state s_7 is a special activity which does nothing. Other states in LTS could be reasoned similarly.

Assume a composite service model $\mathcal{M} = (Var, P_0, V_0, F)$ and the LTS of \mathcal{M} is $L(\mathcal{M}) = (S, s_0, \Sigma, \delta)$. Every action $a \in \Sigma$ is triggered by an atomic activity. The atomic activities used in this chapter are $rec(S)$, $reply(S)$, $sInv(S)$, and $aInv(S)$, where S is the component service that the atomic activities are communicated with. For activities $rec(S)$ and $sInv(S)$, they are required to wait for reply from component service S before continuing, therefore their availability, cost and response time are equivalent to the availability, cost and response time of component service S . For activities $reply(S)$ and $aInv(S)$, they are not required to wait reply from the component service S , therefore they are regarded as internal operations. We assume the availability, cost and response time for an internal operations as 100%, \$0 and 0 ms respectively (see Section 6.3.5 for discussion). Given two states $s = (P, V, Q)$, $s' = (P', V', Q')$, where $s, s' \in S$, $s \xrightarrow{a} s' \in \delta$, and $a \in \Sigma$, we use the function $AtomAct(a)$ to denote the atomic activity that triggers the action a . As an example, given $s = (sInv(S) \rightarrow rec(S), V, Q)$ and $s' = (rec(S), V, Q)$, the function $AtomAct(a)$ returns the activity $sInv(S)$. We define the function $ResponseTime(a)$, $Availability(a)$ and $Cost(a)$ to map the action a to the response time, availability, and cost of the activity returned by $AtomAct(a)$. Using the previous example,

$ResponseTime(a)$ is the response time of activity $sInv(S)$, which is essentially the response time of component service S .

6.3 Verification

This section is devoted to discuss how to verify combined functional and non-functional requirements based on the LTS semantics of Web service composition. Current works only verify one aspect of requirements, either functional or non-functional requirement, however, these two aspects are inseparable. Some properties such as CPS is required to reply the user within 2 seconds, involves both functional and non-functional requirements. Therefore, we propose an approach to combine functional and non-functional requirements. Our approach is based on the assumption that the LTS is finite and acyclic.

6.3.1 Verification of Functional Requirement

To verify functional requirements of a BPEL program, LTS of the BPEL program is built from composite service model. We support the verification of deadlock-freeness, reachability of a state. We also support LTL assertions with fairness assumptions, such as strong fairness and weak fairness. To verify the LTL formulae, we make use of automata-based on-the-fly verification algorithm [60], by firstly translating a formula to a Büchi automaton and then checking emptiness of the product of the system and the automaton. For fairness checking, we utilize the on-the-fly parallel model checking based on Tarjan strongly connected components (SCC) detection algorithms similar to [149].

6.3.2 Integration of Non-Functional Requirement

In this section, we present our approach in integrating the non-functional requirements into verification framework. Different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework. In the following, we adopt two different strategies in integrating the non-functional requirements. We first discuss our approach in integration of availability and cost, and following that, we discuss the integration of response time.

6.3.3 Integration of Availability and Cost

In this section, we present our approach to integrate the availability and cost to the verification framework. Given two states $s = (P, V, Q)$, $s' = (P', V', Q')$, where $s, s' \in S$, $s \xrightarrow{a} s' \in \delta$, and $a \in \Sigma$, the availability and cost of state s' is calculated using the following formulae:

$$\begin{cases} s'.Q(availability) = s.Q(availability) * Availability(a) \\ s'.Q(cost) = s.Q(cost) + Cost(a) \end{cases} \quad (6.1)$$

Example. We illustrate the integration using the LTS of CPS as shown in Figure 6.3. In state s_0 , it has the initial availability of 1 and initial cost of \$0. From state s_0 , it evolves into state s_1 after invocation of i_1 . Since i_1 has availability of 0.9 and cost of \$3 (refer to Table 6.1), therefore the resulting QoS vector of state s_1 is $\langle r_1, 1 * 0.9, 0 + 3 \rangle = \langle r_1, 0.9, 3 \rangle$. From state s_1 , it evolves into state s_3 after the invocation of i_3 , and since i_3 has availability of 0.8 and cost of \$2, the resulting QoS vector of state s_3 is $\langle r_3, 1 * 0.9 * 0.8, 0 + 3 + 2 \rangle = \langle r_3, 0.72, 5 \rangle$. Other states are calculated similarly. In general, given an execution $\pi = s_0 \xrightarrow{a_1} s_1 \rightarrow \dots \rightarrow s_{n-1} \xrightarrow{a_n} s_n$ in $L(\mathcal{M})$, where $\{s_0, \dots, s_n\} \in S$ and $s_i \xrightarrow{a_{i+1}} s_{i+1} \in \delta$, for all $0 \leq i < n$

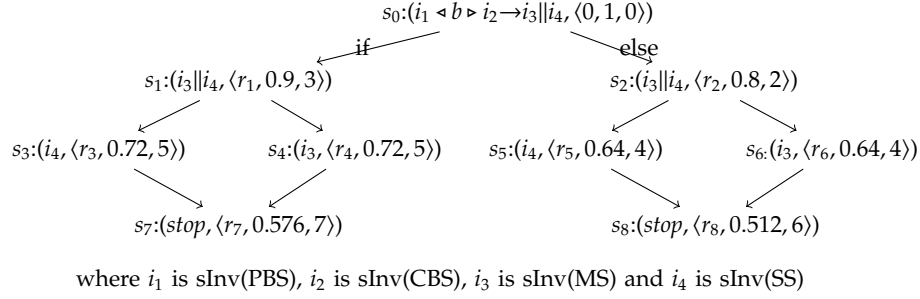


Figure 6.3: LTS of CPS with Availability and Cost

$$\begin{cases} s_{i+1}.Q(availability) = s_0.Q(availability) * \prod_{m=1}^i Availability(a_m) \\ s_{i+1}.Q(cost) = s_0.Q(cost) + \sum_{m=1}^i Cost(a_m) \end{cases} \quad (6.2)$$

with $s_0.Q = \langle 0, 1, 0 \rangle$.

6.3.4 Integration of Response Time

One might naively think that we can adopt the method of calculating the cost as the method for calculating the response time. However, this would result in incorrect result. Refer to Figure 6.3, the value of response times r_2 , r_5 , r_6 , and r_8 will be 2 ms, 5 ms, 3 ms, and 6 ms respectively by using the method of calculating the cost in Section 6.3.3. In such case the value of r_8 is incorrect. The reason is that it should be calculated as maximum of value of r_5 and r_6 , since parallelism allows both i_3 and i_4 to be executed simultaneously, and the total time for the response time is decided by the maximum response time of i_3 and i_4 . A challenge to evaluate the maximum time in state s_8 is that the information of parallelism in state s_2 ($i_3 || i_4$) is removed in state s_5 and state s_6 (only left with i_3 or i_4). In order to retain this information, we preprocess the BPEL service model \mathcal{M} to associate with a time tag which will be used to calculate the response time in the LTS generation stage.

Algorithm 17 presents the main algorithm for preprocessing. Given a BPEL process P_0 ,

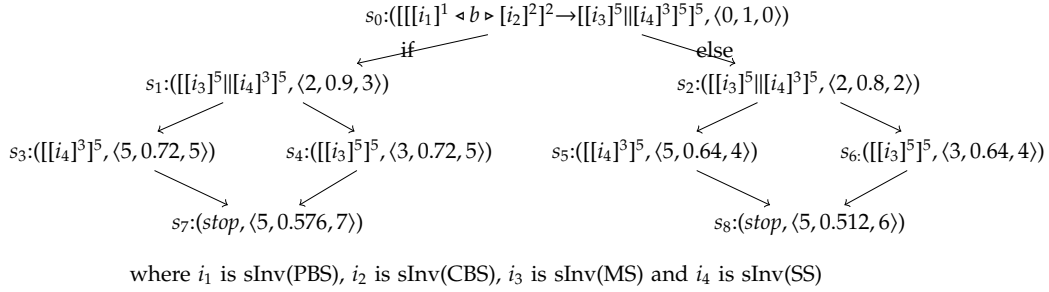


Figure 6.4: LTS of CPS with Response Time, Availability and Cost

$TagTime(P_0, x)$ returns the process P'_0 which is the process P_0 with its internal activities associated with time tags. Given each activity $Acv \in P_0$, a value $timetag \in \mathbb{R}_{\geq 0}$ is associated with Acv , denoted as $Acv.timetag$. $Acv.timetag$ represents the total time delay from the start of process P_0 , up to the completion of activity Acv . In the following, we describe the Algorithm 17. The function $TagTime(P_0, x)$ is used to calculate the total time delay from the start of process P_0 up to the completion of activity Acv . Variable $x \in \mathbb{R}_{\geq 0}$ is the total time delay from the start of process P_0 to the point just before the execution of activity Acv . Lines 1, 5, 9 and 11 are used to detect the structure of the activities. At line 1, if P is detected to be a sequential activity, activity A will be tagged with the delay x (line 2) as A is triggered once P is triggered. Subsequently, activity B will be tagged. Since activity B is executed after the completion of activity A , therefore the x is set to be the value of $A.timetag$ (line 3). Finally, the $timetag$ of P is the same as $timetag$ of B , since the completion of activity B implies the completion of execution of process P (line 4). At line 5, if P is detected to be a concurrent or conditional activity, activity A and activity B will be tagged with value x (lines 6 and 7), since A and B are triggered at the same time once P is triggered. At line 8, the $timetag$ of P is the maximum value of $timetag$ of A and B (refer to Section 6.2.2 for details). If P is detected to be a synchronous receive activity or invocation activity, the $timetag$ of P is set to the sum of x and $ResponseTime(P)$ (line 10).

Algorithm 10: Algorithm TagTime(P, x)

input : P, the BPEL process

input : x, the delay from the start to execution of process P

output: P', process P with time tag

```
1 if P is A→B then
2   TagTime(A, x);
3   TagTime(B, A.timetag);
4   P.timetag ← B.timetag ;
5 else if P is A||B or A ◁ b ▷ B then
6   TagTime(A, x);
7   TagTime(B, x);
8   P.timetag ← max(A.timetag, B.timetag) ;
9 else if P is rec(S) or sInv(S) then
10  P.timetag ← x + ResponseTime(P);
11 else if P is reply(S) or aInv(S) then
12  P.timetag ← x;
```

Example. In the following, we use an example to illustrate how to calculate the response time for each state in the LTS. Given initial service process $P_0 = sInv(PBS) \triangleleft b \triangleright sInv(CBS) \rightarrow (sInv(MS) || sInv(SS))$, we denote $P'_0 = TagTime(P_0, 0)$ and

$$P'_0 = [[[[sInv(PBS)]^1 \triangleleft b \triangleright [sInv(CBS)]^2]^2 \rightarrow [[sInv(MS)]^5 || [sInv(SS)]^3]^5]^5$$

where for each activity $A \in P$, $[A]^t$ is used to denote the activity A with $A.timetag = t$. Next, in the LTS generation stage, Algorithm 11 is used to calculate the response time for each state.

Given the process P of some state $s \in S$, $CalculateTime(P)$ in Algorithm 11 returns the total response time $t \in \mathbb{R}_{\geq 0}$ from the initial state s_0 to s' . The value t is assigned to $Q(responseTime)$ for state s' . Lines 1, 6, 11 are used to detect the structure of the activities. We introduce a special activity *skip* to denote the completion of execution of an atomic activity. *skip* is used for the purpose of calculating the response time, and it will be removed after the calculation. At line 1, if P is detected to be a sequential activity, the activity A is then

Algorithm 11: Algorithm CalculateTime(P)

input : P , BPEL process with time tagged

output: $t \in \mathbb{R}_{\geq 0}$, the time delay from the start of initial process P_0 to the completion of P

```
1 if  $P$  is  $A \rightarrow B$  then
2   if  $A$  is skip then
3     return  $A.time_{tag}$ ;
4   else
5     return  $CalculateTime(A)$ ;
6 else if  $P$  is  $A \parallel B$  or  $A \triangleleft b \triangleright B$  then
7   if  $A$  is skip and  $B$  is skip then
8     return  $P.time_{tag}$ ;
9   else
10    return  $CalculateTime(PreviousActive(P))$ ;
11 else if  $P$  is skip then
12   return  $P.time_{tag}$ ;
```

checked whether it is a *skip* activity. If it is (line 2), which implies that activity A has finished execution, $A.time_{tag}$ is returned (line 3). Otherwise, $CalculateTime(A)$ is invoked in order to determine the response time (line 5). At line 6, if P is detected to be a concurrent activity or conditional activity, A and B will be determined whether both are *skip* activities. If it is (line 7), which implies that P has finished execution, $P.time_{tag}$ is returned (line 8). Otherwise, $CalculateTime(PreviousActive(P))$ is invoked in order to obtain the response time (line 10) where $PreviousActive(P)$ is used to denote previous execution activity. For example, given $s = (i_1 \parallel i_2, V, Q)$, $s' = (skip \parallel i_2, V', Q')$, and $s \xrightarrow{a} s' \in \delta$, $PreviousActive(skip \parallel i_2)$ will return $AtomAct(a) = i_1$. At line 11, P is determined to be a *skip* activity implies that P has finished execution, therefore, $P.time_{tag}$ is returned (line 12).

Example. In Figure 6.4, given the initial state s_0 , there are two branches due to the conditional process. If $sInv(PBS)$ is executed, it will evolved into state s_1 with process P'_1 where

$$P'_1 = [[[skip]^1]^2 \rightarrow [[sInv(MS)]^5 \parallel [sInv(SS)]^3]^5]^5$$

By running the Algorithm 11 for PBS to get the response time of PBS, it will return the value 2, therefore state s_1 has the response time of 2 *ms*. After the calculating the response time, the *skip* are removed from P'_1 , which result in process $P_1 = [[sInv(MS)]^5][[sInv(SS)]^3]^5$ as shown in Figure 6.4. The calculation of other states are similar.

6.3.5 Discussion

If a system is verified that it does not satisfy the requirement that the response time is less than 5 *ms* in a state s , it does not necessarily mean that such constraint will be violated in the state s during the execution. The response time is served as a estimated reference value. Furthermore, we do not take the response time, cost, and availability of internal operations into account. In reality, such information can be estimated using runtime monitoring method [123]. Our method can easily extended with such information if they are available.

6.4 Experiment

We evaluate our approach using three case studies. Each case study is a composite service represented as a BPEL process. The experiment data was obtained on a system using Intel Core I7 3520M CPU with 8GB RAM. The experimental results are summarized in Table 6.3.

6.4.1 Computer Purchasing Service (CPS)

As described in Section 6.1.1, CPS is used for allowing users to purchase a computer online using credit cards. The workflow of CPS is illustrated in Figure 6.1. The property *Reach* ($replyUser \wedge (responseTime > 5)$) is to verify whether the activity *reply user* (ru) can be reached with response time more than 5 *ms*. The result is invalid as shown in Table 6.3, which implies that if the *reply user* (ru) is reached, it will be always be less than 5 *ms*, which is

Services	Property	Result	#S	#T	Time(s)
CPS	$(\text{replyUser} \wedge (\text{responseTime} > 5))$	invalid	21	29	0.0087
	$\square \text{responseTime} \leq 5$	valid	26	36	0.0089
	$\square \text{availability} > 0.6$	valid	26	36	0.0083
LS	$\mathbf{R} (\text{replyUser} \wedge (\text{responseTime} > 6))$	invalid	106	241	0.0584
	$\square \text{responseTime} \leq 6$	valid	242	572	0.1866
TAS	$\mathbf{R} (\text{replyUser} \wedge (\text{responseTime} > 3))$	invalid	128	287	0.0631
	$\square \text{responseTime} \leq 3$	valid	264	622	0.0642
	$\mathbf{R} (\text{replyUser} \wedge (\text{availability} \leq 0.3))$	invalid	128	287	0.0437

Table 6.3: Experiment Results

the intended outcome we need. Properties $\square \text{responseTime} \leq 5$ and $\square \text{availability} > 0.6$ are LTL formulas, which are invariant properties denoted that the CPS's response time must always be less than two milliseconds and the CPS's availability is always larger than 50%. These two properties are both verified to be valid in the CPS system. The number of visited states, total transitions and time used for verification are listed in Table 6.3, where #S represents #State, #T represents #Transition and **R** represents reachability.

6.4.2 Loan Service (LS)

The goal of a loan services (LS) is to provide users for applying loans. The loan approval system has several component systems, Loan Record Service (RS), Loan Approval Service (LAS), Customer Details Service (CDS), Customer Loan History Service (CLHS), Customer Credit Card History Service (CCHS), Customer Employment Information Service (CES) and Customer Property Information Service (CPIS). Upon receiving the request from a customer, CDS will be invoked synchronously. If the requested load amount is less than \$10000, CES is invoked and then RS is invoked to record the customer's loan information. After that, loan approval message will be replied to the customer. Otherwise, if the requested amount

is not less than \$10000, CLHS, CCHS, CES and CPIS are invoked concurrently to obtain more detailed information about the customer. Upon receiving all replies, LAS is invoked to determine whether to approve the loan request of the customer or not. If the request is approved, RS is invoked synchronously and then loan approval message will be replied to the customer, otherwise, loan failure message will be replied to the customer. Two properties are verified for LS as listed in Table 6.3, we omit the discussion of the properties as they are similar to properties of CPS.

6.4.3 Travel Agency Service (TAS)

Travel Agency service (TAS) provides a service that helps users to arrange the flight, hotel, transport, etc., for a trip. Once the request is received from the user, Hotel Booking Service (HBS), Flight Booking Service (FBS), Local Transport Service (LoTS) and Local Agent Service (LAS) are triggered to search for available hotel, flight, local transportation and local travel agent concurrently that fulfill the user's requirements. If all four services have returned non-empty results, Record Booking Information Service (RBS) and Notify Agent Service (NAS) are invoked concurrently to store detailed booking information into the system and notify the agent about the customer's details. Finally, TAS replies the detailed booking information to the user. Otherwise, TAS replies booking failure result to the user. Three properties are verified for TAS as listed in Table 6.3. Properties *Reach (reply User \wedge (responseTime>3))* and $\square \text{ responseTime} \leq 3$ are similar to properties verified in CPS, therefore we omit discussion of these two properties here. Property *Reach (replyUser \wedge (availability \leq 0.3))* is to verify whether *reply user (ru)* can be reached with the availability less than 0.3. The result is invalid as shown in Table 6.3, which implies that if the *reply user (ru)* is reached, the availability is always greater than 0.3, which is the intended result we need.

The experiment shows that our approach can be used to verify the combined functional

and non-functional property for real-world BPEL program efficiently.

6.5 Related Work

In this section, previous works will be discussed and compared with verification of web service composition, constraint synthesis of web service composition.

6.5.1 Verification of Web Service Composition

A number of approaches have been proposed to deal with requirements of web service composition. These work can be divided into two major directions. One direction is to transform WS-BPEL processes into intermediate formal models specified in some formal languages and then verify the functional behaviors of the service composition based on the formal models.

Xiang Fu et al. [86, 84, 88, 48] propose a two steps approach. First, each BPEL process is translated to a guarded automaton. Subsequently, these guarded automata are mapped to Promela. Interactions of composite web service are modeled as conversations, the global sequence of messages exchanged by the web services. Data used in the conversations are mainly in XML format. To capture the data semantics, each transition of a guarded automaton is equipped with a guard that is expressed using an XPath [12] expression. The use of XPath expressions as guards allows them to express the manipulation of XML message contents. As communication among web services is asynchronous, each peer is equipped with a FIFO queue to store incoming messages. It is known that checking the temporal properties of message sequences for unbounded queues is undecidable [85]. To mitigate this problem, an abstraction technique called synchronizability [86] is proposed. A composite web service is synchronizable, if the conversation set remains same when asyn-

chronous composition is replaced by synchronous communication. Complete verification is possible if the composite web services are synchronizable. If the composite web service cannot be shown to be synchronizable, then only partial verification can be achieved by fixing the length of input queues. Synchronizability of the web services is checked at the level of guarded automata before translation into Promela. A tool, Web Service Analysis Tool (WSAT), has been developed and the details can be found in [88, 47, 44].

Foster et al. [78] proposed a Finite State Processes (FSP) approach. First, they transform the BPEL process into FSP. Subsequently, the model checker for FSP, Labelled Transition System Analyzer (LTSA), is used for the verification for the deadlock freeness and temporal logic properties. If a counterexample is found, it is shown in a message sequence chart (MSC) for the purpose of intuitive presentation. In [79, 81], they further consider the conformance checking of BPEL and other specification languages such as WS-CDL and MSC. The verification of BPEL properties, as well as conformance checking could be done in the WS-Engineer [80], which has been implemented as an Eclipse plug-in.

Stahl gives the complete transformation from BPEL4WS 1.1 to Petri nets in [145]. Lohmann extends the work of Stahl, presenting a feature-complete Petri net semantics for WS-BPEL 2.0 [112]. With the semantics, Lohmann presents several analysis: detection of unreachable activities, detection of multiple simultaneously enabled activities that may consume the same type of message, detection of cyclic links, and checking for 56 of the 94 WS-BPEL static analysis goals. Martens [116, 118, 117, 115] introduces three criteria (i.e., usability, compatibility and consistency between executable and abstract processes) for business processes and their compositions. A BPEL process is called usable (or controllable) if there exists an environment with which the process can interact with such that the process terminates properly. Two BPEL processes are called compatible if their composition is usable. A BPEL is said to simulate another BPEL process if each environment that makes the latter usable makes the former usable as well. Two BPEL processes are called equivalent (or

consistent) if the one simulates the other and vice versa. Martens also presents algorithms to check if BPEL processes satisfy these criteria. These algorithms have been implemented in the tool WOMBAT [119].

Fisteus et al. [36] proposed a tool VERBUS. VERBUS is designed in a modular way. It proposed an architecture with three layers: the design layer, the formal layer and the verification layer. The design Layer could potentially accept multiple input languages (currently it only supports BPEL4WS 1.1), and translates them to a common formal model in the formal layer. With this formal model, it could make use of different model checkers in verification layer for verification purpose. Model checkers that are supported currently are NuSMV, SMV and Spin. Ferrara [76] proposed an approach that translates BPEL4WS to LOTOS. The specification in LOTOS allowed temporal logic model checking, but also a simulation and bisimulation analysis. A two-way mapping between between LOTOS and BPEL4WS is supported, so that the counterexamples that encountered could be expressed in BPEL. Furthermore, the counterexamples could be mapped back to BPEL4WS for displaying to the user. Koshkina [107, 108] proposed a small language BPE-calculus to capture a subset of BPEL features. An existing verification tool called the Concurrency Workbench (CWB) is customized to verify BPE-calculus. The tool provides deadlock-freeness checking, temporal checking, event preorder checking, and behavior equivalence checking. Salaün et al. [135] advocates the use of strength of different process algebra to collaborately model check the web service compositions. In particular, it gives the mapping from BPEL4WS to CCS, and CWB tool, which supports verification of CCS, could be used to model check it subsequently.

Hallé *et al* [96] propose an approach to verify whether Web applications are implemented following the interface contract that specifies their expected behaviour or not. However, our work is focused on the verification of combined functional and non-functional requirements.

Another direction is focused on the non-functional aspect of BPEL processes. In [106], Koizumi and Koyama propose a performance model to estimate the processing execution time by integrating a Timed Petri Net model and statistical models. However, it only focuses on one type of non-functional requirements and does not consider the functional behaviors. In [89], Fung et al. propose a message tracking model to support QoS end-to-end management of BPEL processes. This work is based on the run-time data, which needs the deployment of the services, in addition, it does not consider the functional requirements of BPEL processes. Our approach verifies both functional and non-functional requirements at design time, which can detect errors at the early stage. In [163], Xiao et al. propose a framework to use the simulation technique to verify the non-functional requirements before the service deployment, which is similar to our work. While their work only focus on non-functional aspect, our work supports verification of combined functional and non-functional properties. In [50], Cardoso et al. and Jaeger et al. propose measurement approaches based on the workflow structures of web service processes, such as parallel, sequential and so on.

6.5.2 Constraint Synthesis of Web Service Composition

Constraint synthesis of Web Service Composition shares common techniques with work for constraint synthesis for scheduling problems. The use of models such as Parametric Timed Automata (PTA) [25] and Parametric Time Petri Nets (TPNs) [159] for solving such problems has received recent attention. In particular, in [56, 56] parametric constraints are inferred, guaranteeing the feasibility of a schedule using PTAs with stopwatches. In [29], we extended the inverse method to the synthesis of parameters in a parametric, timed extension of CSP. Although PTAs or TPNs might have been used to encode (part of) BPEL language, our work is specifically adapted and optimized for synthesizing local timing constraint in the area of service composition. The quantitative measure of the robustness

of real-time systems has been tackled in different papers (see [114] for a survey). However, most approaches consider a single dimension : transitions can usually be taken at most ϵ (before or after) units of time from their original firing time. This can be seen as a ball in $|U|$ dimensions of radius ϵ . In contrast, our approach quantifies robustness for all parameter dimensions, in the form of a polyhedron in $|U|$ dimensions. Our method is related to using LTS for analysis purpose in Web services. In [130], the author proposes an approach to obtain behavioral interfaces in the form of LTS of external services by decomposing the global interface specification. It also has been used in the model checking the safety and liveness properties of BPEL services. For example, Foster et al. [78] transform BPEL process into FSP, subsequently using a tool named as WS-Engineer for checking safety and liveness properties. Simmonds et al. [143] proposes a user-guided recovery framework for Web services based on LTS. Our work uses LTS in synthesizing local time requirement dynamically.

Our method is related to the finding of a suitable quality of service (QoS) for the system [169]. The authors of [169] propose two models for the QoS-based service composition problem [34] model the service composition problem as a mixed integer linear problem where constraints of global and local component serviced can be specified. The difference with our work is that, in their work, the local constraint has been specified, whereas for ours, the local constraints is to be synthesized. An approach of decomposing the global QoS to local QoS has been proposed in [20]. It uses the mixed integer programming (MIP) to find optimal decomposition of QoS constraint.

6.6 Chapter Summary

In this chapter, we have illustrated our approach to verify combined functional and non-functional requirements (i.e., availability, response time and cost) for Web service com-

position. Furthermore, our experiments show that our approach can work on real-world BPEL programs efficiently. In addition, we have implemented our approach into the tool VeriWs [53], which will be introduced in Chapter 7.

Chapter 7

Tool Implementation: VeriWS

Web service technologies enable dynamic inter-operability of heterogeneous and distributed Web-based platforms. The *de facto* standard for Web service composition is Web Services Business Process Execution Language (WS-BPEL) [102], an XML-based orchestration business process language for describing the behavior of a business process based on its interactions with its component services. It supports various compositional structures such as sequence, parallel composition, conditional choice, etc., to facilitate the composition of Web services. This chapter we use the approach proposed in the previous chapter (Chapter 6) to implement our tool, which will be presented in this chapter.

There are two crucial classes of requirements for Web service composition, i.e., functional and non-functional requirements. Functional requirements are related to the conformance of Web service composition to the requirements on its functionality, whereas non-functional requirements are related to the quality of service (QoS), e.g., response time, availability, and cost. Non-functional requirements can determine the success or failure on Web service composition, as the Web service composition that is functionality correct but with poor performance is not likely to be adopted by the users. To guarantee the performance of Web

service composition, the non-functional requirements are often noted down in service-level agreements (SLAs), which are a contractual basis between service consumers and service providers on the expected QoS level. Given a computer purchasing service (CPS), e.g., Dell.com, an example of functional requirements is that “the CPS always replies to users with the purchasing status”, whereas an example of non-functional requirements is that “the CPS always responds within 3 seconds”.

Concurrency has been frequently used in Web service composition. Nevertheless, concurrency often leads to subtle bugs as programmers have to deal with issues like multi-threads and critical regions. Yin *et al.* [166] reports 39% of concurrency bugs are not fixed correctly and concurrency bugs are the most difficult to fix among common bug types. Thus, it is desirable to apply automatic verification techniques on WS-BPEL, e.g., model checking [58]. Existing tools have provided verification for either functional requirements [82], [88], [36], [119], [150] or non-functional requirements [171], however, they could not support for combined functional and non-functional requirements. An example of combined functional and non-functional requirements is “CPS will always reply to the user, and when CPS replies to the user, the delay of CPS will not be larger than 3 seconds (from the points where it receives the request)”.

Although combined functional and non-functional requirements are important for Web service composition, currently there is no integrated tool support for these two classes of requirements. To facilitate the checking and improvement of functional and non-functional aspects of Web service composition, we have developed a toolkit called VeriWS. VeriWS is a tool designed to verify Web service composition for combined functional and non-functional requirements, based on the QoS of participated component services. A counterexample will be provided when the violation of a requirement is detected. It also integrates with a simulator component to provide the simulation on behaviors of the composite service, as well as, to replay the counterexample that is reported.

In the following, we present the main features of VeriWS.

- It supports verification on different kinds of combined functional and non-functional properties of Web service composition, i.e., linear temporal logic (LTL) properties, reachability properties, and deadlock-freeness properties.
- It supports the simulation of Web service composition models and provides the counterexample in WS-BPEL, so that developers can easily locate the origin of the bug and subsequently fix it.
- It is easy to use for Web service modeling, testing and verification.

Our initial experiment was illustrated in [54], it has demonstrated the effectiveness of our method. To our knowledge, VeriWS is the first tool to provide verification on combined functional and non-functional requirements of Web service composition.

7.1 VeriWS

7.1.1 Architecture and Implementation

VeriWS is a self-contained toolkit that provides the state-of-the-art verifier for combined functional and non-functional requirements for Web service composition specific to WS-BPEL. Given WS-BPEL programs with QoS values for each component service, VeriWS enables integrated verification of both functional and non-functional requirements. Web service composition is verified directly based on its operational semantics. We adopt the formal operational semantics of WS-BPEL described in [78]. With the operational semantics, a WS-BPEL program can be treated as a *transition system*, which is subject to model checking. When the verification is violated, a counterexample in WS-BPEL will be provided to make developers easier to find and correct problems.

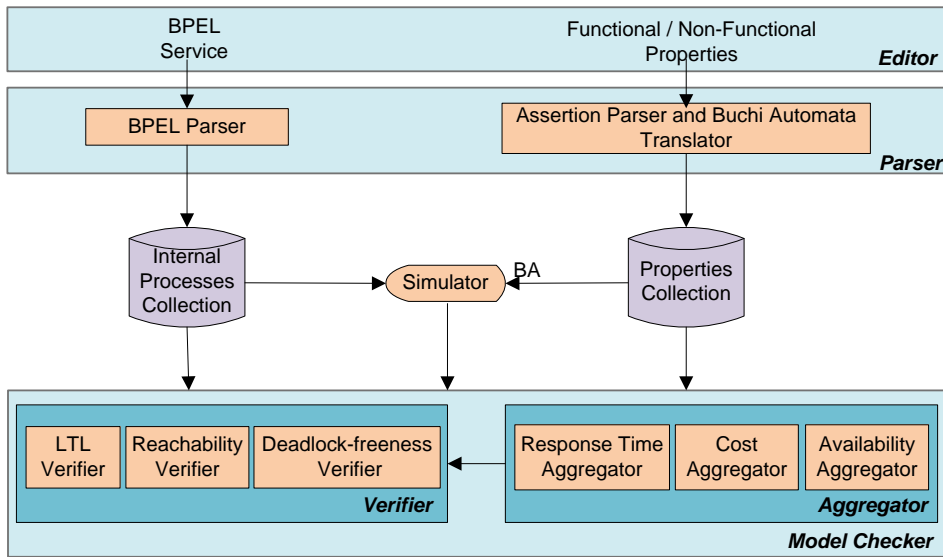


Figure 7.1: VeriWS Architecture

VeriWS is implemented in C# and uses a modular architecture. It provides powerful editor, simulator, and verifier. Figure 7.1 shows the architecture of VeriWS. To verify the composite service using WS-BPEL, a user first inputs the WS-BPEL service description and combined requirements to be verified using the editor. The editor is implemented using the text editor component of Sharp Develop framework¹, which supports multi-documents environment and is customizable in terms of syntax highlighting, code folding, etc.

Subsequently, the WS-BPEL service description and the combined requirements are parsed into processes collection and properties collection respectively. The verifier is then used to check the WS-BPEL service against the combined requirements. The verifier checks the combined requirements based on the aggregated QoS values obtained from the aggregators. There are three kinds of verifiers, i.e., LTL verifier, reachability verifier and deadlock-freeness verifier, which are designed to check LTL properties, reachability and deadlock-freeness properties respectively. VeriWS offers extensible software architecture, such that new verifiers and aggregators could be plugged in easily.

¹<http://www.icsharpcode.net/OpenSource/SD/Default.aspx>

QoS Attribute	Sequential	Parallel	Loop	Conditional
Response Time	$\sum_{i=1}^n q(s_i)$	$\max_{i=1}^n q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^n q(s_i)$

Table 7.1: Aggregation Function

The simulator can be used to visualize the behavior of a WS-BPEL service and the combined requirements. The simulator can also be used to replay the counterexample returned by the verifier. We illustrate the details of verifier, aggregator and simulator in the following.

7.1.2 Aggregator

Different aggregators are used to aggregate different QoS based on the their aggregation functions. Table 7.1 shows the aggregation functions that *response time aggregator* used for different compositional structures. *Response time aggregator* sums up the response time of component services in sequential structure, i.e., the response time of the composite service is calculated by summing up the response time of the n participated component services. The response time of the composite service composed by parallel structure is decided by the maximum response time among the n participated component services. As for loop structure, the response time of the composite service is obtained by summing up the response time of the participating component service for k times, where k is the number of maximum iteration of the loop. While for the conditional composition, the response time of the composite service is the maximum response time of n participating component services since at the design phase it is unknown which guard will be satisfied. Cost and availability aggregators work similarly, where their aggregation functions could be found in [54].

7.1.3 Verifier

The verifier model checks combined functional and non-functional requirements. If a counterexample is found, it can be replayed using the simulator. Several verifiers are implemented to cater for different kinds of combined requirements. There are three categories of properties that are currently supported by the verifier: deadlock-freeness, reachability, and LTL. The verifier integrates the aggregated QoS values from the aggregators, into the transition system that represents the WS-BPEL process using approaches in [54], and offers verification for the following properties:

LTL Property. An LTL property checks whether the property specified in LTL holds. To verify the LTL formulae, we adopt the automata-based on-the-fly verification algorithm [149, 70], i.e., by firstly translating a formula to a Büchi automaton (BA) and then checking emptiness of the product of the system and the automaton.

Reachability Property. A reachability property asks whether there exists a state that fulfills a given property. The properties are specified using the constraint on a set of verification variables. The verification variables are manipulated by the WS-BPEL service using the extended WS-BPEL attributes [128]. To verify reachability, Depth First Search (DFS) or Breath First Search (BFS) is applied on the transition system of the WS-BPEL process to search for a state that fulfilled the given property.

Deadlock-freeness. Checking deadlock-freeness is to check whether a WS-BPEL service contains a deadlock. The WS-BPEL service starts with receiving the message from the user, and ends with reply the user with the desired result. A state is deadlocked if it does not have any outgoing transitions, and the user has not yet been replied. To verify deadlock-freeness, standard graph traversal algorithm (e.g., DFS or BFS) is applied on the transition system of the WS-BPEL process to search for a deadlock state.

Tool	Requirement	Input	Intermediate
WSEngineer	Functional	BPEL	FSP
WSAT	Functional	BPEL	GFSA
VERBUS	Functional	BPEL	Promela
WOMBAT	Functional	BPEL	Petri nets
AgFlow	Non-functional	Statecharts	–
VeriWS	Combined	BPEL	–

Table 7.2: Web Service Verification Tools

7.1.4 Simulator

The simulator could be used to visualize the behaviors of a WS-BPEL service in the form of a transition system. The simulator provides various simulation functions for users, e.g., complete generation of the transition system – where the user could generate entire state space of the WS-BPEL program; interactive exploration of the transition system – where the user could view the subset of the transition system by exploring on the actions of their interest; random simulation - where an example trace is automatically generated for the user. This allows users to have an in-depth understanding on the behavior of the WS-BPEL service through the simulation interface. The simulator is also used to visualize BA generated from the negation of a LTL property. In addition, the simulator could also allow the user to replay the counterexample returned by the verifier, when a property is violated, in order to aid the user on finding out the origin of the problem.

7.1.5 Comparison with Existing Tools

Table 7.2 shows the comparison of VeriWS with existing tools. Existing tools can either verify only functional requirements or non-functional requirements as shown in the *Requirement* column. Existing functional verification tools (WSEngineer [82], WSAT [88], VERBUS [36], WOMBAT [119]) takes WS-BPEL as input, and translate WS-BPEL into an intermediate formal language (e.g., FSP, Petri nets) and use verification techniques and tools

for the intermediate formal language (e.g., LTSA² tool is used for FSP) for verification. Their counterexamples are in their respective intermediate formal language (e.g., counterexample of WSEngineer is in FSP). Existing non-functional verification tool (AgFlow [171]) requires the user to provide corresponding statecharts [97] as input to provide the non-functional analysis for the composite service.

In [156, 151], we have developed the tool on verification of computation orchestration language, nevertheless the tool is only focused on functional requirement of Orc language [105]. In [153, 111], we have developed tools in analyzing the time requirement, which are only focused on the non-functional requirement.

Compared to existing tools, VeriWS is distinguished by several features. First, VeriWS supports efficient combined functional and non-functional verification which could not be achieved by any existing tools. In addition, for non-functional verification, AgFlow only provides for the non-functional analysis for the composite service as a whole, e.g., "the CPS will always be response within 3 seconds". In contrast, VeriWS could support more "fine-grained" non-functional requirement such as "when invoking the shipping service, CPS will not be delayed for more than 3 seconds". Second, VeriWS does not translate WS-BPEL to an intermediate formal language; therefore it could provide the counterexample in WS-BPEL language. Another advantage is that, this also provides a more natural handling of data semantics in XML, where formalism like XPath³ is normally used to retrieve particular data elements in an XML document. WSAT is the only existing tool that supports on the XML data manipulation, and to support a single line of XPath operation, it requires to translate to 56 lines of Promela codes (excluding the comments) [87]. The translated code is hardly comprehensible. While in our approach, we could directly manipulate the XML data based on the semantics of XPath operation. This will in turn provide the user with

²www.doc.ic.ac.uk/ltsa/

³www.w3.org/TR/xpath/

```

<process xmlns:bpel="http://VeriWS/" ... >
...
<sequence>
  <receive ... />
  <if>
    <condition>$CustomerType = 'Corporate'</condition>
    <invoke partnerLink="CBS" ... />
  <else>
    <invoke partnerLink="PBS" ... />
  </else>
</if>
<flow>
  <invoke partnerLink="MS" ... />
  <invoke partnerLink="SS" ... />
</flow>
<reply ... />
</sequence>
</process>

```

Figure 7.2: WS-BPEL Description for CPS

a more pleasant experience to understand the behaviors of WS-BPEL services using the simulation tool.

7.2 Demonstration

The section is to complement with the video demonstration to illustrate the models and requirements to be verified. We use the Computer Purchasing Service (CPS), a service that allows users to purchase a computer online using credit cards.

7.2.1 Computer Purchasing Service (CPS)

The WS-BPEL program of CPS is described in Figure 7.2. In the following we illustrate the workflow of CPS. Upon receiving the request from the customer with his personal information and the computer he wishes to buy, if the type of the customer is corporate, Corporate Billing Service (CBS) is invoked synchronously (i.e., waiting for the reply from CBS before moving on) to bill the customer, otherwise, Personal Billing Service (PBS) is invoked synchronously to bill the customer. Manufacture Service (MS) and Shipping

Service (SS) are triggered concurrently once receiving the billing confirmation message. MS is invoked synchronously to inform the manufacture department with the purchased computer. SS is invoked synchronously to arrange the shipment for the customer. Finally, the purchasing result will be replied to the customer.

7.2.2 Requirements for Verification

We provide verification on five requirements, which are listed as follows:

1. $CPS \models \text{deadlockfree}$
2. $CPS \models \Box (\text{ReplyUser} \implies \text{ResponseTime} \leq 6)$
3. $CPS \models \Box \text{Availability} \geq 0.95$
4. $CPS \models \Box \text{Cost} \leq 5$
5. $CPS \models \text{reach} (\text{invokeCBS} \wedge \text{Cost} < 1)$

The first property is the deadlock-freeness property which is used to check whether CPS is deadlock-free. The second property (i.e., $\Box (\text{ReplyUser} \implies \text{ResponseTime} \leq 6)$) is an LTL property, which is to check whether the CPS always replies to users within 6 seconds. The third property (i.e., $\Box \text{Availability} \geq 0.95$) is an LTL property which is used to check whether the availability of CPS is always greater or equals to 0.95. The fourth property (i.e., $\Box \text{Cost} \leq 5$) is an LTL property which is used to check whether the cost incurred by CPS is always less than or equal to 5 dollars. The fifth property is a reachability property ($\text{reach} (\text{invokeCBS} \wedge \text{Cost} < 1)$) is a reachability property, which is used to find out whether there is a possibility that the accumulated cost is less than 1 dollar at the time when CBS is invoked. Both the second and fifth properties check the combined functional and non-functional requirements.

7.3 Chapter Summary

For Web service composition, both functional and non-functional requirements are important. Therefore, it is crucial to verify functional and non-functional requirements of composite services at design time so that it could detect the problem before deployment. With VeriWS, we provide a tool to check the satisfiability of combined functional and non-functional requirements of composite services directly based on their semantics.

At the runtime, component services could behave differently after being modified by service providers, or could fail due to various reasons such as network problems, software bugs, hardware failure, etc. Therefore, we propose an automated approach to calculate the recovery plan once detecting the failure in Chapter 8.

Chapter 8

Automated Runtime Recovery for QoS-based Service Composition

Since SOA allows functionalities of the composite services to be distributed to third party service providers; therefore component services are allowed to evolve freely, independently of each other. Component services could behave differently after being modified by service providers, or could fail due to various reasons such as network problems, software bugs, hardware failure, etc. In addition, a composite service (expressed, e.g., in BPEL) uses late binding mechanism, where abstract services are used during design time, and the concrete services would only be decided during runtime. As a result, design-time validation of composite services, such as through testing or static verification, is insufficient. Therefore, runtime monitoring of the functional properties, and being able to recover from properties violations, are essential for the dependability of a composite service.

Composite service languages, such as BPEL, are equipped with constructs to support the *compensation* mechanism. The compensation mechanism is an application-specific way to reverse completed activities. For example, the compensation of making a hotel reservation

would be to cancel the reservation. One of the important issues of the current compensation mechanism is that it is uncertain whether the compensation will lead to a system state that could satisfy the functional properties of the composite service.

Existing works [142, 143] address this problem by devising a recovery plan that allows the system to recover from properties violations, based on exploring the state space of the composite service using planning techniques based on SAT-solvers. This approach suffers from several disadvantages. First, the full state space needs to be generated for recovery plans exploration; therefore it might encounter the state explosion problem, especially when dealing with large-scale service composition (see, e.g., [73]). Second, the QoS aspects (e.g., dependability and response time) of the recovery plan are not taken into account explicitly in this approach. An important aspect of a recovery plan is the QoS. A recovery plan with poor QoS is not only ineffective, but also it might result in undesired side effects such as compensation loops, i.e., it leads to failed services and compensate repeatedly. Because a failed service has low dependability, the recovery plan that involves the invocation of the failed services will be filtered away in a selection procedure that is QoS-aware.

In this chapter, we address this issue by proposing a technique based on genetic algorithms (GA) for searching for a recovery plan. GA are computational methods inspired by the biologic evolution, which have been used to solve a variety of problems (see, e.g., [100]). Traditional GA use fix-length encodings, called chromosomes. However, using chromosomes to encode the recovery plan poses a challenge, as the length of chromosome depends on the size of the state space, which is unknown beforehand. Therefore, an estimation on the chromosome length is necessary. Exact calculation of the length of chromosome by exhaustively exploring all possible states is not feasible, as it obviously leads to the state space explosion. Furthermore, over-approximating the length of chromosome might render GA ineffective, whereas under-approximation might result in the incomplete encoding of recovery plan. In this chapter, we propose *rGA* (recovery plan GA), to find a near-optimal

recovery plan in a large state space. *rGA* addresses the aforementioned problems by adaptively adjusting the length of chromosomes with respect to the size of the state space during the recovery plan searching. Furthermore, *rGA* does not require generating the full state space – it only generates the partial state-space on-the-fly during the exploration. Our contributions are summarized as follows.

1. Novel representation and operations – We propose *rGA*, a novel GA making use of dynamic-length chromosomes to represent the recovery plans, and manipulating them using genetic operators for evolving new recovery plans.
2. On-the-fly state-space exploration – *rGA* does not require the generation of full state space beforehand. State space is generated on-the-fly during recovery plan exploration. Since *rGA* performs guided exploration on the most promising region of the search space for the recovery plans, only partial state space is explored in the end. This improves time and space efficiency.
3. Practical recovery plan generation – *rGA* adopts an enhanced initial population policy, and selects a recovery plan with near-optimal QoS; this enables the effective restoration of correctness for the composite service. Furthermore, *rGA* utilizes runtime information (such as variable value before failure) and the structure of composite service, resulting in a more realistic recovery plan with higher chance of success.

We have evaluated *rGA* with real-world case studies, which demonstrate the effectiveness over existing approaches.

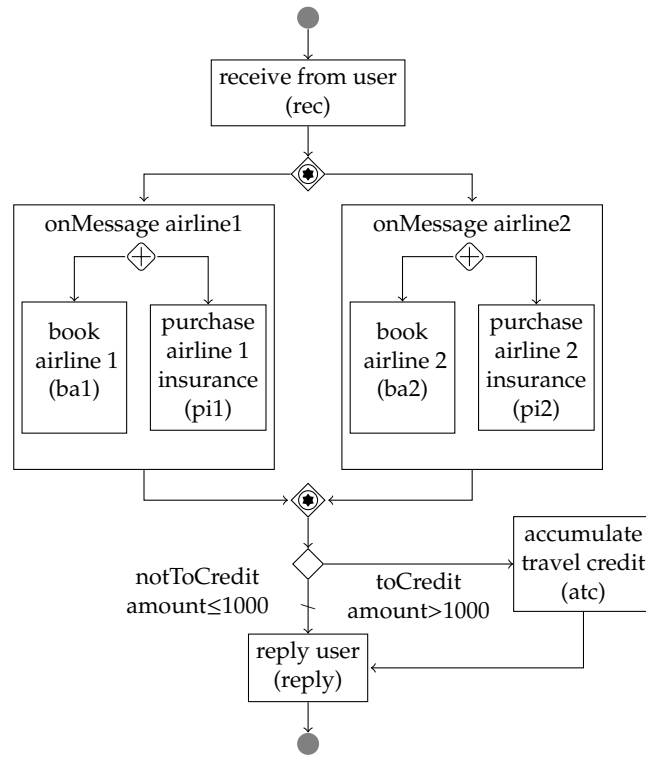
Chapter Outline. Section 8.1 presents a running example. Section 8.2 introduces the QoS compositional model and the necessary terminology. Section 8.3 presents *rGA*. Section 8.4 provides the evaluation of our approach. Section 8.5 reviews the related work. Finally, Section 8.6 concludes this chapter.

8.1 Motivating Example

BPEL [26] is a *de-facto* industry standard for implementing composition of existing Web services by specifying an executable workflow using predefined activities. In this chapter, we assume that composite services are specified using the BPEL language. Basic BPEL activities that communicate with component Web services include `< receive >`, `< invoke >`, and `< reply >`, which are used to receive messages, execute component Web services and return values respectively. In addition, a `< pick >` activity is used to wait for the occurrence of exactly one message from a set of messages.

The control flow of services is defined using activities such as `< sequence >`, `< while >`, and `< if >`, to provide sequential ordering, loop, and conditional structure respectively. BPEL also supports parallel execution of activities by using the `< flow >` activity. A `< scope >` activity is used to contain other activities, and it can be associated with a compensation handler, which specifies activities for compensating the effects of executing the `< scope >` activity. In this chapter, the `< while >` loops are assumed to be bounded and the loop bound could be estimated using methods like [74].

We consider here a toy example of a Travel Booking Service (TBS), where the goal is to help users to book for the transportation for their travel choice. The workflow of this example is illustrated in Figure 8.1a. Upon receiving the service request from the user (`rec`), a `< pick >` activity (denoted by \blacklozenge) is enabled to wait for exactly one message from two possible messages (`airline1`, `airline2`) provided by the user. If `airline1` message is received, a `< flow >` activity (denoted by \oplus) is invoked: two activities `ba1` and `pi1` are invoked concurrently to book airline 1 and purchase airline 1's insurance respectively. Similar workflow applies when `airline2` message is received. Subsequently, an `< if >` activity (denoted by \diamond) is used to check whether the purchased amount is larger than 1000. If yes, the `accumulate travel credit (atc)` service is invoked to accumulate the travel credit



(a) Workflow for TBS

```

<pick ext:isControllable=true ... >
...
<invoke operation="ba2"... >
  <compensationHandler>
    <invoke operation="cA2" ... ./>
  </compensationHandler>
</invoke>
...
</pick>
  
```

(b) Compensation for book car service

Figure 8.1: Transport Booking Service (TBS)

that could be used in the next purchase of the user. In either case, `reply user (reply())` is called to return the result of purchases to the user.

Now, let us consider a scenario where the book airline 2 (ba2) service is unreachable. Classic recovery strategies may retry it or switch it to an alternating service [38]. We denote such recovery strategy a *point recovery strategy*, as it involves retrying or switching of a particular service. There are cases where such a strategy does not work. For example,

ba2 service could be down, therefore retrying would not work. In addition, there might not exist an alternating service that could be switched directly. In such a case, another important strategy, which we denote as *workflow recovery strategy*, could be used. A flow recovery strategy involves modifying the workflow by backtracking to a previous state, and finding an alternative path for execution. To implement the flow recovery strategy, one needs to devise a *recovery plan* specifying how the compensation should be done, and which alternative path to choose. A good recovery plan also needs to be *QoS-aware*. We give below some of the QoS factors that need to be considered.

- 1) Cost: What is the cost for compensation, and what is the possible future costs that would be likely to incur in the recovery plan?
- 2) Dependability: What is the chance of success of the recovery plan?
- 3) Response time: What is the expected response time of the recovery plan?

These issues will be addressed in the next sections.

8.2 QoS-aware Compositional Model

In this section, we define the QoS-aware compositional model used in this work. We first give the formal definition of a composite service.

Definition 8 (Composite Service). A composite service \mathcal{M} is a tuple (Var, V_0, P_0) , where Var is a finite set of variables, V_0 is an initial valuation that maps each variable to its initial value, and P_0 is the composite service process.

The semantics of composite service is captured using labeled transition systems (LTSs), as discussed in the following.

8.2.1 Labeled Transition System

Definition 9 (Labeled Transition System (LTS)). *An LTS is a tuple $\mathcal{L} = (S, s_0, \Sigma, \delta)$, where S is a set of states, $s_0 \in S$ is the initial state, Σ is the universal set of actions, and $\delta : S \times \Sigma \times S$ is a transition relation.*

In this chapter, a *state* s is of the form (V, P) , where valuation V is a partial function that maps a variable to its value (in its domain), and process P is a composite service process. Given a composite service (Var, V_0, P_0) , a sample valuation V is $(\{var_1 \mapsto 1, var_2 \mapsto \perp\}, P_0)$, where $var_1, var_2 \in Var$. $var_2 \mapsto \perp$ denotes that var_2 is undefined.

In this chapter, we assume that an error action Err (resp. an error state s_{Err}) always exists in Σ (resp. S) of any LTS. The error action Err is used to model the error condition (e.g., component service unreachable, functional correctness property violated). The error state s_{Err} is reachable from any state of S via action Err , i.e., $\forall s \in S \setminus \{s_{Err}\}, (s, Err, s_{Err}) \in \delta$.

Given an LTS $\mathcal{L} = (S, s_0, \Sigma, \delta)$, we use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in \delta$. Given a state $s \in S$, we denote by $Enable(s)$ the set of states reachable from s by one transition; formally, $Enable(s) = \{s' | s' \in S \wedge a \in \Sigma \wedge s \xrightarrow{a} s' \in \delta\}$. An *execution* π of \mathcal{L} is a finite alternating sequence of states and actions $\langle s_0, a_1, s_1, \dots, s_{n-1}, a_n, s_n \rangle$, where $\{s_0, \dots, s_n\} \in S$ and $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$. We denote by $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_{n-1} \xrightarrow{a_n} s_n$ the execution π . The *prefix* of execution π is a fragment of π that starts from state s_0 and ends with a state s_i where $i \leq n$. A *complete execution* is an execution starting in the initial state and ending in a terminal state. A state $s \in S$ is *terminal* if there does not exist a state $s' \in S$ and an action $a \in \Sigma$ such that $s \xrightarrow{a} s' \in \delta$; otherwise, s is *non-terminal*. In addition, we denote the LTS of a BPEL service \mathcal{M} by $L(\mathcal{M})$.

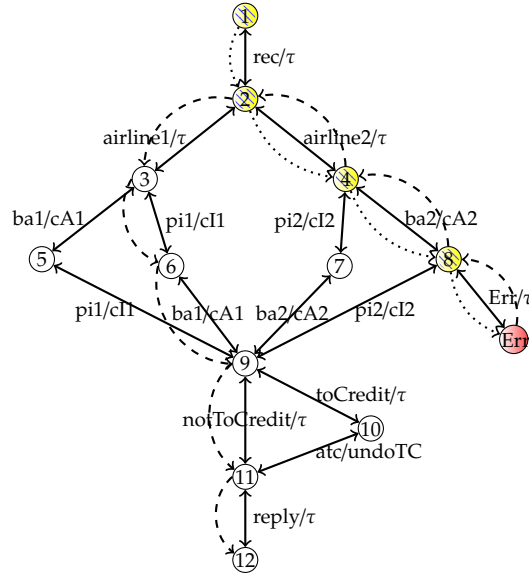


Figure 8.2: LTS of TBS example

8.2.2 Example: Transport Booking Service

The LTS $L(TBS)$ of the TBS example is shown in Figure 8.2. The dashed and dotted arrows are not part of the semantics and they will be explained later on. The formal semantics of BPEL activities in this chapter is based on [78]. For example, consider the conditional activity, $A_{atc} \triangleleft b \triangleright A_{reply}$, that is enabled at state s_9 , where the activity A_{atc} is executed when the guard $b = (\text{amount} > 1000)$ is evaluated to be true, otherwise the activity A_{reply} is executed. From state $s_9 = (\{\text{amount} \mapsto \perp\}, A_{atc} \triangleleft b \triangleright A_{reply})$, it has two possible enabled states, which are $s_{10} = (\{\text{amount} \mapsto \perp\}, A_{atc})$, and $s_{11} = (\{\text{amount} \mapsto \perp\}, A_{reply})$ respectively. This is denoted in the LTS as a state s_9 with two outgoing transitions to states s_{10} and s_{11} . Noted that if b is defined, b is either false or true; therefore only one branch is explored in the LTS. Similarly, pick activities ($\langle \text{pick} \rangle$) and parallel activities ($\langle \text{flow} \rangle$) are specified using two outgoing arrows to denote all possible execution orders of their child activities. For the sake of readability, the error transitions Err from all states (except state s_8) to the error state s_{Err} are not shown in the LTS.

A state is a *migration state* if the state provides alternative choices of execution, i.e., it mi-

grates from the current execution to another one. Migration states include the states where the `< flow >` activity, `< pick >` activity or non-idempotent service invocation is enabled. A service invocation is *idempotent* if any invocation with the same input parameters give the same result. In Figure 8.2, valid migration states from the state s_8 are states s_4 , s_2 , and s_1 , shown in hatched yellow circles.

8.2.3 Backward Actions

BPEL supports compensation mechanism [26] as an application-specific way to reverse the activity that has already been completed. The limitation of the default compensation mechanism is that it is difficult to determine the system state after compensation, and therefore it is hard to decide whether it would end up in a system state where the functional properties could be satisfied.

To address this problem, we make an observation that every action of BPEL can make up to two kinds of changes – internal and external changes. Internal changes modify the valuation V of current system state to a different valuation V' , while external changes modify the state of component services. External changes could only be made by communication activities, e.g., `< receive >`, `< invoke >`, and `< reply >`, since communication activities are the only activities communicating with component services.

To undo internal changes, the valuation prior to executing for an action is stored as a snapshot valuation; therefore during recovery process, internal changes can be undone by reversing the current valuation V' to the snapshot valuation V automatically. To allow undoing of the external changes, users are required to specify a compensation handler for each communication activity a . For example, in Figure 8.1b, a compensation handler is specified for `ba2` operation, which compensates the external changes made by `ba2` by invoking the `ca2` operation to cancel the flight that has been booked. As a consequence, for

every action a , we have a corresponding *backward action*, a_{bak} , which “goes back” to the state prior to execute action a by reversing the internal changes using the snapshot valuation and external changes with the help of compensation handler. An example of backward action is $s_8 \xrightarrow{cA2} s_4$ in $L(TBS)$, where the composite service compensates from state $s_8 = (P, V)$ to $s_4 = (P', V')$, by undoing the valuations from V' to snapshot valuation V and at the same time canceling the flight that has been booked. We use τ to denote a backward action that does nothing to compensate. A non-backward action is a *forward action*; an example is $s_4 \xrightarrow{ba2} s_8$. Given a pair of forward action a_f and backward action a_b , where $s \xrightarrow{a_f} s'$ and $s' \xrightarrow{a_b} s$, we combine them as single notation $s \xleftrightarrow{a_f/a_b} s'$, e.g., $s_4 \xleftrightarrow{ba2/cA2} s_8$. We use Σ_F and Σ_B to denote the set of all possible forward actions and backward actions respectively.

8.2.4 Monitoring Automata

In this section, we introduce how the functional properties are represented and verified. The functional properties are represented using deterministic finite automata (DFA), called here *monitoring automata*. Formally:

Definition 10. A monitoring automaton \mathcal{A} is $(Q, Q_0, \Sigma, \delta, F)$, where Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, Σ is the universal set of actions, $\delta : Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is a set of accepting states.

We use Σ^* to denote a set of finite sequences of actions. Given a monitoring automaton \mathcal{A} , a sequence of actions $a_1 a_2 \dots a_n \in \Sigma^*$ is *accepted* by \mathcal{A} if there exists a path in \mathcal{A} of the form $q_0 \xrightarrow{a_1} q_1 \xrightarrow{\dots} q_{n-1} \xrightarrow{a_n} q_n$, where $q_0 \in Q_0$, $q_n \in F$, $a_i \in \Sigma$ and $\forall 1 \leq i \leq n, (q_{i-1}, a_i, q_i) \in \delta$. An execution $\pi = s \xrightarrow{a_1} s_1 \xrightarrow{\dots} s_{n-1} \xrightarrow{a_n} s_n$ is *accepted* by \mathcal{A} if the sequence of actions $a_1 a_2 \dots a_n$ is accepted by \mathcal{A} ; otherwise it is *rejected* by \mathcal{A} . We denote the set of accepted sequences of actions as $L(\mathcal{A})$.

Given a functional property P_s , $\mathcal{A}(P_s)$ denotes its monitoring automaton. An execution is

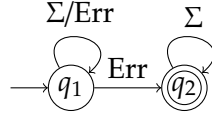


Figure 8.3: Monitoring Automata

accepted by $\mathcal{A}(P_s)$ if it violates the property P_s (otherwise it conforms to P_s). For example given a functional property P_1 “Unreachability of component service can never happen in TBS”, where error action Err is triggered when the component service is unreachable. The monitoring automata for functional property P_1 is shown in Figure 8.3. Given a set of properties, we define the monitoring automata of a composite service CS as $\mathcal{M}_{CS} = \langle \mathcal{A}(P_1), \dots, \mathcal{A}(P_N) \rangle$, where P_i is a functional property (for $1 \leq i \leq N$). Given an execution π in $L(CS)$, π satisfies \mathcal{M}_{CS} , denoted as $\pi \models \mathcal{M}_{CS}$, if π is rejected by all automata $\mathcal{A} \in \mathcal{M}_{CS}$. Otherwise, π violates \mathcal{M}_{CS} , denoted by $\pi \not\models \mathcal{M}_{CS}$.

8.2.5 Recovery Plan

Consider again the LTS $L(\text{TBS})$ in Figure 8.2. An execution starts from state s_1 to state s_8 as shown using dotted arrow ($\cdots \rightarrow$). At state s_4 , the ba2 service is invoked and subsequently evolves into state s_8 . Since the ba2 service is unreachable and timeout by the BPEL runtime engine, this could lead the system to the error state s_{Err} . However, the service monitor discovers the anomalies, and interferes the current process. To recover from the error, a recovery plan is calculated. A recovery plan is a guideline of execution that is used to compensate the current error (using backward actions) to a migration state, and choose an alternative path that could lead to the terminal state (using forward actions). In TBS, a possible recovery plan r is to compensate from state s_{Err} to migration state s_2 using backward actions, and go forward from state s_2 to state s_{12} . The recovery plan of TBS, denoted by r_{TBS} , is shown using dashed arrow ($-- \rightarrow$).

Definition 11. A recovery plan r is an execution $s_{\text{Err}} \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} s_m \xrightarrow{a_{m+1}} s_{m+1} \xrightarrow{a_{m+2}} \dots \xrightarrow{a_n} s_n$ where s_n is a terminal state, s_m is a migration state with $0 \leq m \leq n$, $\forall j \leq m, a_j \in \Sigma_B$, and $\forall k > m$,

$$a_k \in \Sigma_F.$$

A *prefix* of the recovery plan r is a fragment of execution of r that starts with s_{Err} and ends with s_i where $i \leq n$. Sometimes, we also use the term *partial recovery plan* to denote a prefix of a recovery plan. A *suffix* of a recovery plan r is the fragment execution of r that starts with any state in the execution, and ends with terminal state s_n .

Controllability of a recovery plan. Consider the recovery plan r_{TBS} for TBS. At migration state s_2 , according to the recovery plan r_{TBS} , it needs to proceed to state s_3 . However, the semantics of the `< pick >` activity chooses which branch to execute depending on the messages (airline1 or airline2) that are received from the user. In such case, it is a violation of semantics if we follow the recovery plan. Therefore, we extend the `< pick >` activity with an attribute `isControllable` by using BPEL extension attribute [26] feature, so that users are allowed to specify which activities are controllable by the recovery module. In our example, the `< pick >` activity that is activated at state s_2 is specified to be controllable, by setting the `isControllable` attribute to true (see Figure 8.1b). Since the `< pick >` activity is specified as controllable, the activity would follow the recovery plan. Besides the `< pick >` activity, the user also needs to specify the controllability of the `< flow >` and `< if >` activities. If the `< flow >` activity is set to be controllable, then the runtime engine would disregard the concurrent semantics of the `< flow >`, and follow the recovery plan using sequential semantics. If the `< if >` is set to be controllable, then the runtime engine would disregard the valuation of guard condition and execute the branches that are chosen by the recovery plan. Suppose that the `isControllable` of the `< flow >` activity that is enabled at state s_3 and the `< if >` activity that is enabled at state s_9 is set to be true and false respectively. In this case, the recovery process would proceed until state s_9 . At state s_9 , since the `< if >` activity is uncontrollable, the recovery process ends, and normal execution proceeds. During normal execution, the `< if >` activity will decide to enter state s_{10} or s_{11} depending on the value of amount.

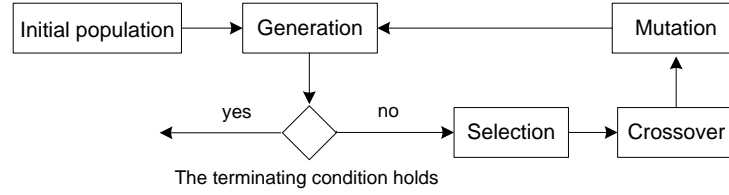


Figure 8.4: Typical Flow of Genetic Algorithms

We call the maximal controllable portion of an execution its *controllable prefix*. For the case of TBS, the controllable prefix is from state s_{Err} to state s_9 . We say that state s_9 is an *uncontrollable state*, which is a state that puts an end to the controllable prefix. Similarly, we denote the portion of an execution starting from uncontrollable state as its *uncontrollable suffix*. Although the uncontrollable suffix (i.e., from state s_9 to state s_{12}) is not executed as part of the recovery process, but it provides an insight on the executions that starts from uncontrollable state. During the calculation of recovery plan, the uncontrollable suffix could help to find a recovery plan that ends up in an uncontrollable state that has a better executions starting from it. Therefore, the composite service has higher chance to conform with both functional and non-functional requirement when recovery process ends and normal execution starts.

8.3 Service Recovery as a GA Problem

Our work is based on genetic algorithms (GA) for calculation of the recovery plan.

8.3.1 Preliminaries of Genetic Algorithms

GA [146] are stochastic search methods based on principles of biological evolution, inspired by the "survival of the fittest" principle of the Darwinian theory of natural evolution. GA encode a potential solution to a specific problem using a simple chromosome-like data

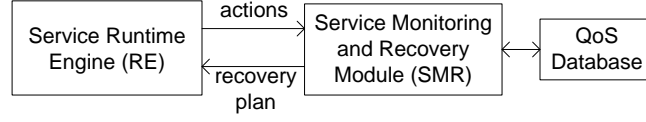


Figure 8.5: Service Monitoring and Recovery Framework

structure, and apply genetic operators to these structures in such a way to preserve critical information. GA are typically suited for optimization problems where the problem space is large and complex.

Figure 8.4 introduces a typical workflow of GA. A GA begins with an (typically random) initial generation of chromosomes, which we call it *initial population*. Genetic operators, such as *selection*, *crossover*, and *mutation*, are applied on a generation, to evolve the next generation of chromosomes. Genetic operators operate based on the *fitness* of chromosomes – the highly-fit chromosomes have higher chance to be evolved into the next generation. The fitness of chromosomes is typically quantified by the *fitness value* of the chromosome. The evolution continues until the terminating condition. An example of the terminating condition could be that the number of generations exceeds a predefined upper bound $n \in \mathbb{Z}_{>0}$.

We name our GA-based approach, *rGA*. To support on-the-fly partial exploration of state-space in *rGA*, the recovery plan is encoded in dynamic-length chromosome, in contrast to the typical fix-length chromosome. The details of encoding will be provided in Section 8.3.3. Subsequently, we introduce the genetics operators that manipulate the chromosomes in Section 8.3.4, and demonstrate how the fitness value of a chromosome is calculated in Section 8.3.5. To allow fast convergence of *rGA*, we propose an enhanced initial population policy, as explained in Section 8.3.9. In the following, we discuss the architecture of the service monitoring and recovery framework that is used in this chapter.

8.3.2 Architecture

The architecture of our work is shown in Figure 8.5. The service runtime engine (RE) is an environment used to execute the BPEL composite services; here, we are using ApacheODE [3], an open-source runtime engine for BPEL composite services. The Service Monitoring and Recovery Module (SMR) contains the monitoring automata, \mathcal{M}_{CS} , of the composite service CS that is executing in the RE. During the execution of CS , the SMR intercepts the actions from the RE. The intercepted actions are used to update the states of all monitoring automata $m_i \in \mathcal{M}_{CS}$ that are stored in the SMR, and these actions will also be recorded as part of the execution π_{CS} for the composite service CS . In addition, after the RE communicating with a component service S , the SMR will update the QoS database with the latest QoS information (e.g., response time and availability) of component service S .

By checking the status of each monitoring automata $m_i \in \mathcal{M}_{CS}$, the SMR could detect whether the functional properties of CS are violated. If so, service recovery is initiated to calculate the recovery plan. The recovery plan will be calculated based on the execution π_{CS} , and estimated QoS attributes from QoS database. Subsequently, the recovery plan would be returned to RE and RE would resume with the recovery according to the recovery plan. The details of the calculation of recovery plan will be introduced in the rest of this section.

8.3.3 Genetic Encoding of a Recovery Plan

We now introduce the representation of recovery plans as chromosomes. The technical challenge when developing the representation is that classic GA use fixed-size chromosomes, while the recovery plan lying within an LTS has an unknown number of states and transitions. Providing a unique representation of a recovery plan requires an exhaustive exploration of the LTS in order to know the chromosome length required to encode the

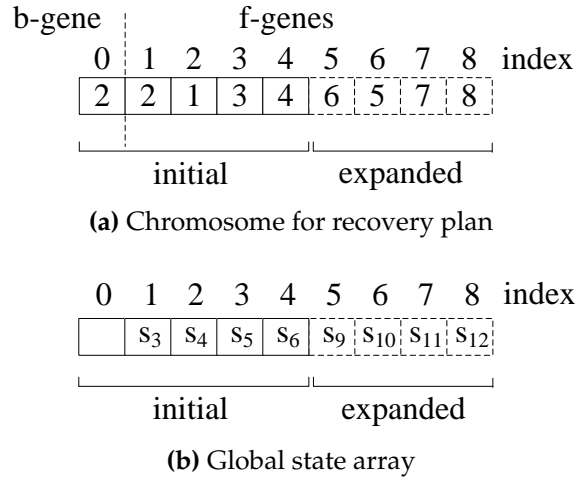


Figure 8.6: Genetic Encoding of Recovery Plan

recovery plan; this might encounter the infamous state-explosion problem. In order to address this problem, we propose *dynamic-length chromosomes* to encode the recovery plan, where the length of chromosomes is adjusted adaptively during the (partial) exploration of the LTS for the optimal recovery plan.

We adopt here array-based chromosomes. The chromosome in Figure 8.6a represents the recovery plan r_{TBS} . Given a chromosome of length n , array indices are numbered from 0 to $n-1$. A gene is an element of the array, and the value of a gene ranges over non-negative integer number. Given a recovery plan, there are two parts: backward execution and forward execution. The backward execution contains only backward actions, followed by the forward execution that contains only forward actions. Similarly, the genes are divided into two parts: a b-gene (backward-gene) and a set of f-genes (forward-genes), to represent the backward execution and forward execution respectively. The b-gene is located at index 0 of the chromosome, and f-genes are located from indices 1 to $n-1$. We demonstrate genetic encoding of recovery plan using the example shown in Figure 8.2. The value of b-gene shows the number of backward actions are used to compensate. Assume the value of the b-gene is 3: compensating three steps from error state s_{Err} would reach the migration state s_2 .

After compensation, we consider forward actions. The forward execution is encoded differently from the backward execution. To encode the forward execution, we need to make use of a global state array (see Figure 8.6b) which is shared by all chromosomes. Intuitively, the values in f-genes give the *priority values* of the states in state-array. We introduce f-genes and state-array using the recovery plan r_{TBS} that has been compensated to migration state s_2 according to the value of b-gene. Initially, f-genes and the state array are empty. From migration state s_2 , two states s_3 and s_4 are enabled, calculated by $EnableStates(s_2)$. Since these two states do not exist in the state array, they are added to the array at indices 1 and 2 (index 0 is always left empty because of b-gene). At the same time, assume two values, 2 and 1 are added to f-genes. Details on how these values are decided will be given in Section 8.3.9. Values in f-genes represent the *priority values* of the states in state-array at the same index – states s_3 and s_4 have priority values of 2 and 1 respectively. Since state s_3 has higher priority than state s_4 ($2 > 1$), state s_3 is chosen to be the next state in the recovery plan. This process goes until state s_6 . At this point, both the state array and chromosome are *dynamically expanded* to contain additional states and their priority values.

8.3.4 Genetic Operators

GA make use of crossover and mutation to create new chromosomes for the next generation of a population. We introduce these two operators, adapted from [90].

Crossover. We make use in rGA of a position-based crossover operator. Two new chromosomes are produced from each crossover operation. The algorithm of crossover operator is shown in Algorithm 12. At line 2, $rand(0, 1)$ randomly chooses a real number between 0 and 1. If the number is less than P_{cross} then it performs the positional crossover $pCrossOver$ (line 2) on chromosomes P_1 and P_2 .

We illustrate the positional crossover $pCrossOver$, using an example shown in Figure 8.7a.

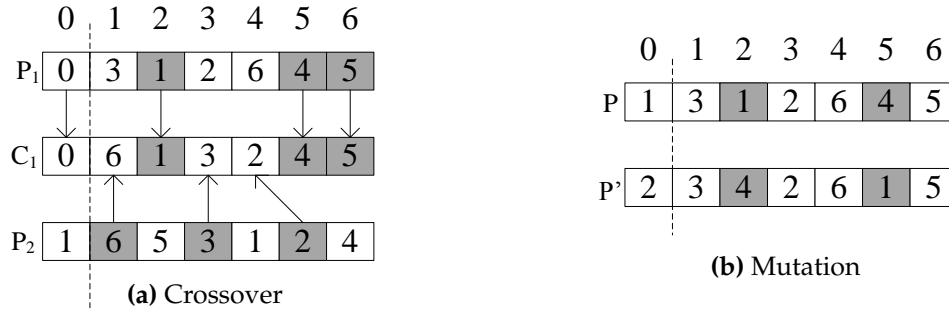


Figure 8.7: Genetic Operations

In Figure 8.7a, a new chromosome C_1 is produced by applying positional crossover $pCrossOver$ to chromosomes P_1 and P_2 . The b-gene of C_1 is created by choosing the b-gene from P_1 or P_2 randomly. The f-genes of C_1 are produced by taking some f-genes from P_1 at random positions; in the example, we take f-genes at positions 2, 5, and 6 from P_1 . Subsequently, the empty positions of C_1 , viz., positions 1, 3, and 4, are filled up by performing left-to-right scan on P_2 , and the unused numbers will be used to fill in the empty positions. The production of another new chromosome C_2 (not shown in the graph) is symmetric to the production of C_1 . For the b-gene, the crossover operator chooses from P_2 , as P_1 has been chosen by C_1 . Subsequently, it takes the f-genes positions 2, 5, and 6 from P_2 , and fills in the empty position by performing left-to-right scan on P_1 . The resulting chromosome of C_2 is $\langle 1, 3, 5, 1, 6, 2, 4 \rangle$.

If the number is greater than P_{cross} , it simply returns the chromosomes P_1 , and P_2 at line 3.

Mutation. The swap-based mutation operator is used for the mutation operation. The algorithm of mutation operator is given in Algorithm 13. At line 4, *getBackwardSteps()* returns the set of numbers for b-gene that could lead to a migration state, and the *rand* function chooses one of them randomly. At line 7, the value of the gene at position i is randomly swapped with a gene from position 1 to $n - 1$.

Figure 8.7b shows how a new chromosome P' is produced by applying the mutation operator to chromosome P . The b-gene is mutated by randomly picking a number that

Algorithm 12: Crossover

input : Chromosomes P_1, P_2 **output**: Chromosomes C_1, C_2

```
1  $C_1 \leftarrow P_1$  ;  $C_2 \leftarrow P_2$ ;  
2 if  $\text{rand}(0, 1) \leq P_{\text{cross}}$  then  $\langle C_1, C_2 \rangle \leftarrow p\text{CrossOver}(C_1, C_2)$ ;  
3 return  $\langle C_1, C_2 \rangle$ ;
```

Algorithm 13: Mutation

input : Chromosome P **output**: Chromosome C

```
1  $C \leftarrow P$ ;  
2  $n \leftarrow |P|$ ;  
3 if  $\text{rand}(0, 1) \leq P_{\text{mut}}$  then  
4    $C[0] \leftarrow \text{rand}(\text{getBackwardSteps}());$  // for b-gene  
5 for  $i = 1$  to  $n - 1$  do  
6   if  $\text{rand}(0, 1) \leq P_{\text{mut}}$  then  
7      $\text{swap}(C[i], C[\text{randInt}(1, n - 1)])$ ; // for f-genes  
8 return  $C$ ;
```

could compensate to a migration state. For f-genes, two genes are chosen randomly and their values are swapped.

8.3.5 Calculating the Fitness Value

8.3.6 QoS Optimality

In this chapter, we focus on quantitative QoS attributes that can be quantitatively measured using metrics. There are two classes of attributes, namely positive ones (e.g., availability) and negative ones (e.g., response time). Positive attributes have a positive effect on the QoS, and therefore need to be maximized. Conversely, negative attributes need to be minimized. For simplicity, we only consider negative attributes in this chapter, since positive attributes can be transformed into negative attributes by multiplying their value with -1 . Given n

QoS attributes of a service s , we use an attribute vector $Q_s = \langle q_1(s), \dots, q_r(s) \rangle$ to represent it, where $q_i(s)$ is the i th QoS attributes of Q_s .

A composite service S makes use of a finite number of component services to accomplish a task. Let $C = \{s_1, \dots, s_n\}$ be the set of all component services that are used by S . The composite service communicates with component services using the communication activities, which includes $\langle \text{invoke} \rangle$ and $\langle \text{onMessage} \rangle$ activities. Given an action a belonging to the communication activity, $S(a)$ denotes the component service the communication activities communicates with.

Given an execution $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$, we use a vector $Q'_\pi = \langle q'_1(\pi), \dots, q'_n(\pi) \rangle$ to represent its aggregated QoS attributes, where $q'_k(\pi)$ is the k th aggregated QoS attributes. $q'_k(\pi)$ is calculated as follows:

$$q'_k(e) = F_k \left(q_k(S(a)) \right), \quad (8.1)$$

with $R = \{a_i \mid i \in \{1, \dots, n\} \wedge a_i \text{ is an action belonging to synchronous communication activity}\}$ and F_k is the QoS aggregation function for attribute k , defined below:

Response time	Availability	Throughput
$\sum_{i=1}^n q(s_i)$	$\prod_{i=1}^n q(s_i)$	$\min_{i=1}^n q(s_i)$

Other QoS attributes share the similar aggregation functions, e.g, the cost attribute has the same aggregation function as the response time attribute. Component services have multi-dimensional attributes, and we need a methodology to facilitate their comparison in term of their QoS. In this work, we use a simple additive weighting (SAW) technique [167] to obtain a score for multi-dimensional attributes. This simple additive weighting technique uses two stages for producing the score. The *normalization* stage normalizes the QoS attribute values so that they are independent of their units and range to allow comparison. The *weighting* stage allows users to specify their preferences on different QoS attributes. Normalization of aggregated QoS of an execution π is done by comparing with the maximum and minimum

Algorithm 14: Fitness

input : Population *popul*, chromosome *csome*
output: Fitness value of *csome*

```
1 Exec  $\leftarrow \emptyset$ ; state  $\leftarrow$  failure state;  
  // for b-gene  
2 for int i = 1 to ind[0] do  
3   | Exec  $\leftarrow$  Exec  $\cap$  state; state  $\leftarrow$  compensate(state);  
4 Exec  $\leftarrow$  Exec  $\cap$  state;  
  // for f-genes  
5 states  $\leftarrow$  EnableStates(state);  
6 while true do  
7   | foreach s  $\in$  states do  
8     | if s.id =  $\emptyset$  then  
9       | currId  $\leftarrow$  currId + 1;  
10      | if currId  $\geq$  chromo_size then  
11        | eArr  $\leftarrow$  createNewGenes(csome, stateArr);  
12        | foreach csome  $\in$  popul do  
13          | | csome  $\leftarrow$  csome  $\cap$  eArr;  
14        | s.id  $\leftarrow$  currId;  
15      | state  $\leftarrow$  arg maxs  $\in$  states (csome[s.id]);  
16      | Exec  $\leftarrow$  Exec  $\cap$  state; states  $\leftarrow$  EnableStates(state);  
17 feasible  $\leftarrow$  verify(M, Exec); ncState  $\leftarrow$  getNcState(Exec);  
18 ncState.execs.add(getNcExec(Exec));  
19 if feasible then  
20   | return 0.5 + 0.5 * (G(getCExec(Exec)) + G(ncState));  
21 else  
22   | return 0.5 * (G(getCExec(Exec)) + G(ncState));
```

aggregated QoS. The maximum (resp. minimum) aggregated QoS can be obtained by aggregating maximum (resp. minimum) QoS attribute values. Formally: $Q_{min}(k) = F_{i=1}^m K_{min}$ and $Q_{max}(k) = F_{i=1}^m K_{max}$ with $K_{min} = \min_{s \in CS} q_k(s)$ and $K_{max} = \max_{s \in CS} q_k(s)$, where $Q_{min}(k)$ and $Q_{max}(k)$ are the minimum and maximum aggregated values for k th QoS attribute of execution π , m is the number of states in the longest execution of a composite service, and CS is the set of all component services that are used by the composite service. m can be easily obtained with static analysis on the composite service.

Suppose each service has r QoS attributes; the QoS optimality of the execution π , $Q(\pi)$, is

calculated as follows using SAW:

$$Q(\pi) = \begin{cases} \sum_{k=1}^r \frac{Q_{max}(k) - q'_k(\pi)}{Q_{max}(k) - Q_{min}(k)} \cdot w_k & \text{if } Q_{max}(k) \neq Q_{min}(k) \\ 1 & \text{otherwise} \end{cases} \quad (8.2)$$

where $w_k \in \mathbb{R}^+$ is the weight of q_k and $\sum_{k=1}^r w_k = 1$.

Given an uncontrollable state s , the *QoS optimality of the state s* , $Q(s)$, is the average value of the QoS optimality of execution that starts from the state s , i.e.:

$$Q(s) = \frac{1}{|E|} \sum_{e \in E} Q(e) \quad (8.3)$$

where E is the set of execution that starts from state s and ends in a terminal state. Given E_r a controllable prefix of recovery plan r , and S_r an uncontrollable state of r , the QoS optimality of r , $Q(r)$, is:

$$Q(r) = Q(E_r) + Q(S_r) \quad (8.4)$$

where $Q(E_r)$ and $Q(S_r)$ are calculated using Equation 8.2 and Equation 8.3 respectively.

8.3.7 Global Optimality

The global optimality of a recovery plan concerns both the QoS optimality and whether the recover plan satisfies the functional requirements. The global optimality $G(r)$ of a recovery plan r is:

$$G(r) = \begin{cases} 0.5 + 0.5 \cdot Q(r) & \text{if } r \models \mathcal{M} \\ 0.5 \cdot Q(r) & \text{otherwise.} \end{cases} \quad (8.5)$$

The global optimality for a recovery plan such that $r \models \mathcal{M}$ (resp. $r \not\models \mathcal{M}$) has its value ranging from 0.5 to 1 (resp. 0 to 0.5). Therefore, it can be guaranteed that a recovery

plan satisfying the functional requirements has a higher global optimality value than any recovery plan violating the functional requirements.

Definition 12. *Given a composite service CS, and the set of all feasible recovery plans R_f , the optimal recovery plan r_m is the feasible recovery plan with the maximal optimal value, i.e., $r_m = \arg \max_{r \in R_f} (G(r))$.*

In the following, we present a heuristic method *rGA*, used to find the optimal recovery plan.

8.3.8 Fitness Function

Given a chromosome, we need a metric to decide its worthiness as a candidate solution. The fitness function, denoted as *Fitness*, is used to provide the evaluation, and returns a value called fitness value that represents the worthiness of the candidate solution. The fitness value is typically used by the selector to decide which pair of chromosome instances will be chosen for mating. Highly fit chromosomes relative to the whole population will have higher chance of being selected for mating, whereas less fit chromosomes have a correspondingly low probability of being selected. Some chromosomes generated by the crossover and mutation operations might be infeasible, i.e., they do not satisfy the functional properties of the composite service. We do not simply discard infeasible chromosomes as they might provide candidates that are essential for the optimal solution. Therefore, the strategy is to allow infeasible chromosomes to stay in the population, but with a lower fitness value compared to any feasible chromosome. The fitness value of the chromosome C_r is the global optimality of the recovery plan r that it represents, i.e., $Fitness(C_r) = G(r)$.

The fitness function is computed using Algorithm 14. Lines 7-14 are the procedure discussed in Section 8.3.3 for the purpose of associating states in LTS with f-genes on the chromosome. At line 8, *s.id* is the index of f-gene on the chromosome that state *s* has

been associated with. At line 11, if the current size of the chromosome is insufficient for encoding the recovery plan, an extension array $eArr$ is created, populated with unused and unique priority values. The new states that are encountered will be added to the global state array $stateArr$ at the same time. Subsequently, all chromosomes in the population are extended with $eArr$ (line 13). At line 15, the enabled state with maximal priority value is chosen as the next state. At line 17, *verify* checks whether the execution could satisfy \mathcal{M} using approach discussed in Section 8.2.4; then, *getNcState* gets the uncontrollable state of the execution. At line 18, *getNcExec* gets the part of execution $Exec$ that starts from the uncontrollable state, and is added to $ncState.execs$, which is a set of uncontrollable suffixes that are associated with the $ncState$. At lines 20 and 22, *getCExec* gets the controllable prefix of the execution. The calculation of fitness value in lines 19 to 22 is according to Equation 8.5. For the calculation of $Q(s)$ for uncontrollable state s using Equation 8.3, since we may not have exact set E due to the partial exploration of the state space, the set E is approximated using the set of execution starts from s that we have explored so far, which is the set $s.execs$ (calculated in line 18).

8.3.9 Enhanced Initial Population Policy

We propose an Enhanced Initial Population Policy (EIPP) to overcome shortcomings resulting from randomness of genetic algorithm, such as slow convergence and great variance among the running results. The idea behind is that, by adding the chromosomes likely to contribute to high fitness values to the initial population, we have a higher chance to converge faster to an optimal value.

We introduce the EIPP using the recovery plan r_{TBS} . Suppose we now have value 3 in the b-gene, and the recovery plan would go from state s_{Err} to state s_2 . At state s_2 , there are two enabled states – states s_3 and s_4 . Assume states s_3 and s_4 do not have their priority values assigned yet. EIPP decides the priority values based on the global optimality values

Algorithm 15: Initial Population

input : n (population size), l (chromosome size)

output: An initial population P

```
1  $P \leftarrow \langle c_1, c_2, \dots, c_n \rangle$ ;  $stateArr \leftarrow \langle \emptyset_1, \emptyset_2, \dots, \emptyset_l \rangle$ ;
2 foreach  $c_i \in P$  do
3    $c_i \leftarrow \langle 0_1, 0_2, \dots, 0_l \rangle$ ;  $len \leftarrow |stateArr|$ ;
4    $c_i[0] \leftarrow rand(1, len)$ ;
5    $c_i[1 \dots len] = shuffle(\{1, \dots, len\})$ ;
6    $S \leftarrow EnableStates(R(c))$ ;
7   if  $rand(0, 1) \leq P_{EIPP}$  then
8      $S^r \leftarrow rankWithFitness(R[c], S \setminus stateArr)$ ;
9     foreach  $s \in S^r$  do
10       $stateArr.Add(s)$ ;  $len \leftarrow |stateArr|$ ;
11       $c_i[len] \leftarrow len$ ;
12   else
13      $stateArr.AddAll(S \setminus stateArr)$ ;
14      $c_i[len \dots |stateArr|] \leftarrow shuffle(\{len, \dots, |stateArr|\})$ ;
```

of the partial recovery plans. In particular, we compare the global optimality values of partial recovery plan r_p from error state s_{Err} to states s_3 and s_4 respectively. The global optimality values of partial recovery plan r_p , $G(r_p)$, is calculated using Equation 8.5 with $Q(r_p)$ calculated using Equation 8.2.

Note that assigning priority values according to $G(r_p)$ does not always provide the optimal recovery plan. Suppose the partial recovery plan from state s_{Err} to s_3 and s_4 is r_p^3 and r_p^4 respectively. Assume $G(r_p^4) > G(r_p^3)$; in such a case, it would always require invoking the ba2 from state s_4 or from state s_7 . However, ba2 has a low availability value, since it was previously unresponsive. Therefore, we would end up in getting a recovery plan of low global optimality value; we denote this as the *locality problem*.

To address this problem, the priority values are assigned based on the global optimality value of the partial recovery plan with a probability, denoted as EIPP probability $P_{EIPP} \in \mathbb{R} \cap (0.5, 1]$. The value EIPP of the probability P_{EIPP} is strictly larger than 0.5 to make the EIPP in favor of assigning the priority values for f-genes based on the global optimality

values of partial recovery plans. Suppose that $P_{EIPP} = 0.7$ and $G(r_p^4) > G(r_p^3)$, given a population of 20 chromosomes, on average 6 chromosomes would choose to evolve to state s_4 from state s_2 , which would lead to a recovery plan with better global optimality value. During the evolution, the poor recovery plans that choose state s_4 from state s_2 would be eliminated and the good recovery plans that choose state s_3 from state s_2 would be kept. Therefore, the EIPP probability could effectively mitigate the locality problem.

The algorithm for initializing the population with EIPP is provided in Algorithm 15. Line 1 initializes the population with n chromosomes, which have length l with values of genes set to 0 (line 3). The state array *stateArr* is also initialized to length l , where \emptyset_i denotes an uninitialized value for the i th position. The function *rand*(1, *len*) returns a random number $n \in \mathbb{Z} \cap [1, len]$ and assigns it to the b-gene (located at index 0) of chromosome c_i (line 3). Subsequently, we shuffle the numbers of the set $\{1, \dots, len\}$ randomly, and assign them to the f-genes from index 1 to *len* (line 5). Subsequently, we get the enable states of partial recovery plan that represented by c , denoted as $R(c)$, and assign them to variable S (line 6). If the random number $r \in \mathbb{R} \cap [0, 1]$ is not larger than P_{EIPP} (line 7), then the *rankWithFitness* function sorts the enabled states S not in the state array *stateArr* by their fitness values in ascending order, and assigns them to variable S' . S' is an ordered sequence of states ranked by the fitness values of their partial recovery plans (line 8). We add each state s to the state array, and assign the corresponding f-genes with value $|stateArr|$ just after adding s – this will effectively allow us to assign the priority values in the same order as their fitness values (lines 9–11). Otherwise, all the enable states that are not in state array *stateArr* are added to *stateArr*. Subsequently, we shuffle the numbers of the set $\{len, \dots, |stateArr|\}$ randomly, and assign them to the f-genes from index *len* to $|stateArr|$ (lines 13–14).

Algorithm 16: GA Algorithm

input : Abstract LTS LTS
output: Recovery plan R_{max} with the best global optimality value over all generations

```
1  $popul \leftarrow init\_popul(pop\_size, chromo\_size);$ 
2  $gen \leftarrow 1$ ;  $R_{max} \leftarrow \emptyset$ ;
3 repeat
4    $newPopul \leftarrow max\_ind(popul);$ 
5   if  $fit(max\_ind(popul)) > fit(R_{max})$  then
6      $R_{max} \leftarrow max\_ind(popul);$ 
7   foreach  $\langle P_1, P_2 \rangle \in sample(popul, pop\_size/2)$  do
8      $\langle C'_1, C'_2 \rangle \leftarrow crossover(P_1, P_2);$ 
9      $\langle C_1, C_2 \rangle \leftarrow \langle mutation(C'_1), mutation(C'_2) \rangle;$ 
10     $newPopul \leftarrow newPopul \cup \{C_1, C_2\};$ 
11   $popul \leftarrow newPopul;$ 
12   $gen \leftarrow gen + 1;$ 
13 until  $gen > max\_gen;$ 
14 return  $R_{max}$ 
```

8.3.10 rGA Algorithm

The rGA algorithm is given in Algorithm 16. At line 1, the initial population is initialized using the EIPP given in Algorithm 15 with the default population size pop_size and the default chromosome size $chromo_size$. At line 4, the next population $newPopul$ is extended with the chromosome with maximal fitness value in the current population $max_ind(popul)$, due to the elitist selection adopted. At lines 5–6, the recovery plan with maximal fitness value so far is assigned to R_{max} . At line 7, $pop_size/2$ pairs of chromosomes (P_1, P_2) are sampled using the selection operator discussed in Section 8.3.4. At lines 8 and 9, crossover and mutation operations are applied to (P_1, P_2) , and added to the population of the new generation $newPopul$. This process repeats until it has gone through the maximum number of generations specified by max_gen .

Soundness. For rGA to work correctly, we need to ensure that every chromosome uniquely represents a recovery plan (unique encoding property), and there does not exist any recovery plans that rGA avoid exploring (non-blocking property). We show rGA satisfies these

two properties in the following.

Lemma 3 (Unique encoding). *Given any state in LTS as a starting state, the proposed chromosome uniquely encodes a recovery plan.*

Proof. For the backward execution, given that the graph is acyclic and the compensation is deterministic (since there is a deterministic execution from the initial state to the error state s_{Err} where the failure occurs), the value of the b-gene uniquely determines the migration state. For the forward execution, once a state is added to the state array, it remains in its position in the state array. Therefore, starting from the migration state, we could choose an execution deterministically based on the priority values in f-genes. Combining both, we show that the chromosome can uniquely determine an abstract recovery execution. \square

Lemma 4 (Acyclicity). *Given any state in LTS as a starting state, rGA does not avoid the exploration of any recovery plan.*

Proof. This holds due to the fact that there are no recursive activities in BPEL, and due to the assumption on the loop activities for which the upper bound on the number of iterations is known. \square

8.4 Evaluation

We conducted experiments to evaluate our *rGA* approach. Specifically, we attempted to answer the following research questions.

RQ1. *How does the performance of rGA compare with the state-of-the-art?* We analyze how long *rGA* takes to calculate a recovery plan, and compare the performance with the state-of-the-art.

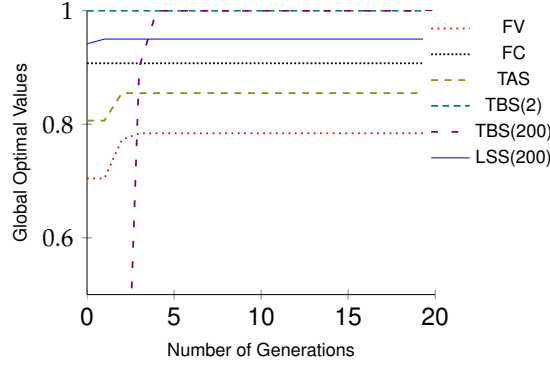


Figure 8.8: Convergence Rate

RQ2. *How is the quality of recovery plan that is selected by rGA?* We measure the quality using the formula

$$quality = \frac{G(r)}{G(r_{exact})} \quad (8.6)$$

where $G(r)$ and $G(r_{exact})$ are the global optimality values of recovery plan returned by the rGA method and the exact method (i.e., the method exhaustively enumerating every possible recovery plans), respectively.

RQ3. *How scalable is rGA?* To evaluate the scalability of rGA , we use the parameterized Large Scale Service (LSS), and Travel Booking Service (TBS) that contain the combinatorial explosion of recovery plans given large values for parameters. In such cases, it is impractical to solve by enumerating all recovery plans.

Experimental Setup. The experiments were conducted on an Intel Core I5 2410M CPU with 4 GiB RAM, running on Windows 7. The mutation and crossover rates for rGA are set to 0.01% and 0.9% respectively. In addition, the population size is set to 20, the number of generations is set to 20, and P_{EIPP} is set to 0.7. The algorithm rGA could terminate earlier if it discovers that the fitness value does not improve for over 6 generations. To evaluate rGA , we explicitly construct an execution that violates the functional properties and leads to the error state s_{Err} , and explore the recovery plan from the error state s_{Err} using rGA . Since rGA could perform differently for each experiment, we took the average of 50 experiments for

rGA				SAT	
case study	time (s)	quality	gen.	length	time (s)
FV	0.7	1	10	42	3.12
FC	0.12	1	6	20	1.38
TAS	0.22	1	6	13	0.27
TBS(2)	0.47	1	6	N/A	N/A
TBS(30)	0.54	1	8	N/A	N/A
TBS(60)	0.87	1	8	N/A	N/A
TBS(120)	1.24	1	10	N/A	N/A
TBS(200)	1.97	1	10	N/A	N/A
LSS(30)	0.85	0.97	7	N/A	N/A
LSS(60)	0.96	0.97	7	N/A	N/A
LSS(80)	1.42	0.96	8	N/A	N/A
LSS(120)	1.92	0.95	8	N/A	N/A
LSS(200)	2.57	0.94	8	N/A	N/A

Figure 8.9: Experiment with *rGA*

each case. In addition, after the fitness value is calculated for a chromosome, we cache the fitness value of the chromosome, so that the fitness value for the same chromosome does not need to be recalculated. We now introduce the case studies used for the experiments. To answer the previous research questions, we evaluate *rGA* using five case studies described in the following.

Flickr [4] is a Web application allowing users to upload and share their photos on the Web. Two known vulnerabilities in the Flickr Web application [52], namely *Flickr Visibility (FV)* and *Flickr Comment (FC)*, are used to evaluate the effectiveness of our approach. Both *FV* and *FC* have been translated to a BPEL model (see [141]).

Flickr Visibility (FV). The options for the photos' visibility are *public*, *family* and *private*; users can set the visibility through the `setPerms()` function. There is a reported issue [1] on this function to fail on changing the visibility to *family*, after uploading the photos with an initial *private* visibility. The BPEL model contains 28 activities and 8 with explicit compensation; its LTS consists of 36 states and 86 transitions. The functional property to

be monitored for FV is “Flickr guarantees the photos to have the visibility set by the user”.

Flickr Comment (FC). Flickr allows authorized users to add comments to private and family photos; for public photos, all users are allowed to comment. There is a reported issue [2] on the failing sequence of a single call to `addComment()` immediately after an `upload()` operation. The BPEL model contains 16 activities and 6 activities with explicit compensation; its LTS consists of 21 states and 51 transitions. The functional property to be monitored for FC is “if a user adds comments to a public photo, the comments should be added successfully into the photo’s comments”.

Trip Advisor System (TAS). This case study is introduced in [142]. The objective of TAS is to schedule the trip for user. It consists of 25 states and 34 transitions. The functional property to be monitored for TAS is “the user cannot book both a limousine and an expensive flight”.

Travel Booking Service (TBS). This is the example used throughout the chapter. In the case study it involves two `< onMessage >` for two airline services. We parameterize the case study using n of `< onMessage >` that for n distinct airline services, each of them involving airline and insurance booking. We denote TBS with k `< onMessage >` activities by $TBS(k)$. For example, the example used in this chapter has $k = 2$. TBS consists of $7 + 3k$ states and $8 + 5k$ transitions. The functional property to be monitored for TBS is “the service always replies to the user”. TBS contains a non-responsive airline booking service `ba2`, invoking `ba2` would lead to the error state s_{Err} , which violates the functional property.

Large Scale Service (LSS). To better evaluate the scalability of our approach, we built a BPEL example with a sequence of k `< pick >` activities. Each `< pick >` activity consists of two `< onMessage >` activities, where one has a good QoS, while the other has a bad QoS. The optimal recovery plan in such a scenario will always consist of the activities with good QoS. We denote LSS with k `< pick >` activities by $LSS(k)$. $LSS(k)$ contains at least $2k$ states and 2^k unique candidate recovery plans. The functional property to be monitored for LSS

is “the service always replies to the user”.

Results. We report the results of evaluating the case studies in Figure 8.8 and Figure 8.9. Figure 8.8 shows the global optimal values of the case studies by varying the number of generations, and we could observe the convergence rate for different case studies. It demonstrates the fast convergence rate for all case studies.

Figure 8.9 shows the details of evaluation for the cases studies. The “time (s)” column reports the time in seconds for *rGA* to produce the recovery plan. The “quality” column reports the quality of the recovery plan found by *rGA* calculated using Equation 8.6. The “gen.” column reports the number of generations that are used to search for recovery plans. We compare the results with [142], which we call *SAT*, since their approach uses a SAT solver to find the best recovery plan. The objective of *SAT* is to find a set of recovery plans that are functionality correct, and the user is responsible for selecting recovery path manually. In addition, Their method is required to specify the maximum length k for the recovery plan, i.e., only the recovery plans less than k and fulfilling the functional requirements are returned. The “length” column contains the value for k . In contrast, in our approach, all recovery plans are explored, and functionality correct recovery plan is chosen automatically in term of their QoS before returning to the user. We compare our results with theirs using their results on three case studies – FV, FC, and TAS they have reported in [142]. For other case studies, their results are unavailable (denoted by N/A). Our method searches the entire state space for optimal recovery plan, without restricting the length of the recovery plan. Therefore, to facilitate fair comparison, we only compare with *SAT*, using the largest k values for the case studies that have been reported in [142].

Our results have shown to outperform theirs for all case studies. In addition, most recovery plans that returned by *rGA* are optimal, i.e., quality=1, except in the LSS case study, which has suboptimal quality, i.e., with quality closed to 1. In addition, we observe that although LSS(80) and LSS(200) used the same number of generation, but LSS(200) spent more time

than LSS(80). This is because LSS(200) contains more states than LSS(80), which results in longer chromosome and slower processing time. The same observation can be applied, e.g., to TBS(120) and TBS(200).

Answer to Research Questions. For research questions RQ1–RQ3, the results in Figure 8.8 and Figure 8.9 show that *rGA* is efficient to offer a recovery plan of good quality, and it is scalable to large composite service.

8.5 Related Work

This work is related to fixing software faults using genetic algorithm. Weimer *et al.* [161] and Arcuri *et al.* [32] investigate genetic programming as a way to automatically fix software faults. Their approach assumes the existence of test cases to test for the functional correctness of chromosomes. In contrast, our method generates a recovery plan, and the functional correctness is checked using the monitoring automata; this is a more lightweight procedure, and it is shown to be suitable for executing it online (see Section 8.4).

This work is related to research on fault-tolerant for service composition. Dodson [68] transforms the original BPEL process into a fault-tolerant one at compiling time, by considering common fault tolerance patterns. This approach introduces redundant behavior to BPEL programs, which may slow down the performance. In contrast, our service monitoring executes at BPEL runtime, which avoids such redundancy. Carzaniga *et al.* [51] propose the use of workaround by considering the equivalent sequences of faulty action in order to provide a temporary solution to mask the effects of the faults on applications. The approach generates all possible recovery plans, without prioritizing them. In contrast, our method filters out infeasible recovery plans; as for the feasible ones, they are ranked by QoS of involved component services.

This work is related to automated recovery for service composition. Baresi *et al.* [38] propose an idea of self-supervising BPEL processes by supporting both service monitoring and recovery for BPEL processes. They propose a backward strategy, which is to restore the system to a previously correct state. However, the strategy does not consider the potential satisfaction of functional properties, and neither is it QoS-aware. Simmonds *et al.* [142] propose an approach to divide a recovery plan to compensate the failures, and guide the application towards a desired behavior. This work is the closest to ours, and our approach has several advantages over theirs. Firstly, the method in [142] requires the exhaustive LTS exploration for the BPEL process by using a SAT solver for calculating the recovery plan. Our approach only requires a partial exploration of the LTS. Also, their method does not take into account the QoS of component services explicitly. Our approach accounts for various QoS aspects of a component service explicitly, and allows users to weight them according to their preferences.

This work is related to self-adaptation of service composition. In [124], Mukhija and Glinz propose an approach to adapt an application by recomposing its components dynamically, which is implemented by providing alternative component compositions for different states of the execution environment. Ghezzi *et al.* [91] describe an ADAM model-driven framework for adaptation by choosing a path that could maximize system's non-functional properties. Denaro *et al.* [65, 66] propose a self-adaptive approach for Web services to adapt the client application dynamically, based on a mechanism for revealing possible runtime mismatches between requested and provided services. These works are orthogonal to our work, as they are related to providing runtime adaptation for normal execution based on the runtime and contextual information, while our work is related to failure recovery.

In [153], we propose an automatic approach to synthesize local time requirement based on the given global time requirement of Web service composition. In [54], we propose an approach to verify the functional and non-functional requirements of Web service compo-

sition. Different from our previous works, this work is focused on automatic synthesis of recovery plan.

8.6 Chapter Summary

In this chapter, we address the problem of service recovery by proposing a new method (*rGA*) based on genetic algorithms. The method improves the efficiency of existing methods in flow-based recovery strategy by allowing partial exploration of the LTS for near-optimal recovery plan. In addition, the recovery plan selection is QoS-aware; therefore, it allows effective recovery from the failure state.

Chapter 9

Conclusion and Future Works

9.1 Conclusion

This thesis studies formal analysis of Web service composition. It is focused on two important kinds of requirements of Web service composition, i.e., functional and non-functional requirements. This thesis is concerned with two stages of Web service composition, i.e., design time and run time.

Chapter 3, Chapter 4 and Chapter 6 are focused on the design stage. In Chapter 3, given the global time requirement we have illustrated our approach to synthesis the local time requirement for component services of a composite service. Our approach is based on parameter synthesis techniques for real-time systems. Component services are selected based on the synthesized result to compose the Web service composition. Then, we need to guarantee the new Web service composition satisfy both of the functional and non-functional requirements. Given the non-functional requirement of the composite service, it is also required to select component services. Chapter 4 proposes a new technique, namely the dynamic ranking optimization (DRO) to address the problem by considering

only a subset of representatives that are likely to succeed, before exploring a larger search space. Then, we need to guarantee the new Web service composition satisfy both of the functional and non-functional requirements. Therefore, in Chapter 6, we have proposed an automated approach to verify combined functional and non-functional requirements of Web service composition based on the semantics directly. We use LTS to capture the semantics. We have also developed a tool VeriWS to implement our approach in Chapter 7. As we are planning to extend the selection to the cloud computing environment, while the cloud computing services can be modeled using SPL, therefore, Chapter 5 proposes feature selection on service-based product lines.

At runtime, component services could behave differently after being modified by service providers, or could fail due to various reasons such as network problems, software bugs. Therefore, we propose an automated approach (*rGA*) based on a genetic algorithm to calculate the recovery plan that could guarantee the satisfaction of functional properties of the composite service after recovery in Chapter 8.

9.2 Future Work

Possible future work will be introduced in this section.

Chapter 3 has presented a novel technique for synthesizing local time constraints for the component services of a composite service CS, knowing its global time requirement. We plan to further improve and develop the technique presented in this paper. First, we will investigate the possibilities to reduce the number of states and transitions, e.g., in the line of reduction of equivalent states [28] or using convex state merging [27].

Chapter 4 has addressed the problem of QoS service composition by proposing a new technique, namely the dynamic ranking optimization (DRO). For future work, we plan to

investigate how DRO can work with other service selection approaches, such as differential evolution [147], for solving the QoS-aware service composition problem.

Chapter 6 has illustrated our approach to verify combined functional and non-functional requirements (i.e., availability, response time and cost) for Web service composition. Furthermore, our experiments show that our approach can work on real-world BPEL programs efficiently. We plan to further improve and develop the technique. Firstly, we will consider various heuristics that could be used to reduce the number of states and transitions. Secondly, we will investigate applying state reduction techniques, such as partial order reduction [77], to improve the efficiency of our approach. Lastly, our work could be extended to other domains such as sensor networks.

Chapter 8 has addressed the problem of service recovery by proposing a new method (*rGA*) based on genetic algorithms. As future work, we plan to investigate how to combine *rGA* with other techniques, such as differential evolution [147].

In future, we would investigate the extension our works to other domains, such as Web security [160] and semantic Web services [75, 69]. For example, we could verify whether there are vulnerabilities on the implementation of Web services, either in the functional or non-functional aspects, that allow an attacker to exploit.

Bibliography

- [1] <http://www.flickr.com/help/forum/46985/>. 8.4
- [2] <http://www.flickr.com/help/forum/15259/>. 8.4
- [3] Apache ODE. <http://ode.apache.org/>. 3.3.2, 3.4.3.1, 8.3.2
- [4] Flickr. <http://www.flickr.com/>. 8.4
- [5] Linux variability analysis tools (lvat) repository. <https://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas>. 5, B.1.1.3
- [6] OASIS Standards. <http://www.oasis-open.org/standards>. 2.1
- [7] PayPal SOAP API Reference. <https://developer.paypal.com/docs/classic/api/PayPalSOA-PIArchitecture>. 2.1
- [8] Salesforce SOAP API Reference. <http://nordicapis.com/rest-vs-soap-nordic-apis-infographic-comparison>. 2.1
- [9] Sat4j – the boolean satisfaction and optimization library in java. <http://www.sat4j.org/>. 5.2.2
- [10] Web Services Glossary. <http://www.w3.org/TR/ws-gloss/>. 2.1
- [11] World WideWeb Consortium. Extensible markup language (XML). <http://www.w3c.org/XML>. 2.1
- [12] XML Path Language (XPath) Version 1.0, 1999. <http://www.w3.org/TR/xpath>. 6.5.1
- [13] Simple Object Access Protocol (SOAP) 1.1. Technical report, May 2000. <http://www.w3.org/TR/SOAP/>. 2.1
- [14] Web Services Description Language (WSDL) 1.1. Technical report, March 2001. <http://www.w3.org/TR/wsdl>. 2.1
- [15] Web Service Semantics (WSDL-S) 1.0, 2005. <http://www.w3.org/Submission/WSDL-S/>. 1
- [16] Y. Adbeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 113–126. Springer-Verlag, 2002. 3.5

- [17] E. Al-Masri and Q. H. Mahmoud. Discovering the best Web service. In *WWW*, pages 1257–1258, 2007. 4.4.2
- [18] E. Al-Masri and Q. H. Mahmoud. Qos-based discovery and ranking of Web services. In *ICCCN*, pages 529–534. IEEE, 2007. 4.4.2
- [19] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *WWW*, pages 881–890. ACM, 2009. 3.5, 4, 4.2.1
- [20] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web, WWW*, pages 881–890. ACM, 2009. 6.5.2
- [21] M. Alrifai, T. Risse, and W. Nejdl. A hybrid approach for efficient web service composition with end-to-end qos constraints. *TWEB*, 6(2):7, 2012. 4.3
- [22] M. Alrifai, D. Skoutas, and T. Risse. Selecting skyline services for QoS-based Web service composition. In *WWW*, pages 11–20. ACM, 2010. 4, 4.2.1, 4.2.1, 4.4, 4.4.1, 4.5
- [23] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. A.2.3
- [24] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993. 3.5
- [25] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *IN PROCEEDINGS OF THE 25TH ANNUAL SYMPOSIUM ON THEORY OF COMPUTING*, pages 592–601. ACM Press, 1993. 6.5.2
- [26] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Golland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, IBM, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. *Web Services Business Process Execution Language Version, version 2.0*. April 2007. 3, 3.1, 8.1, 8.2.3, 8.2.5
- [27] É. André, L. Fribourg, and R. Soulat. Merge and conquer: State merging in parametric timed automata. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2013. 9.2
- [28] É. André, Y. Liu, J. Sun, and J.-S. Dong. Parameter synthesis for hierarchical concurrent real-time systems. In *ICECCS*, pages 253–262. IEEE Computer Society, 2012. 3.5, 9.2, A.2, A.2.3, A.2.3.1, A.2.3.2
- [29] E. Andre, Y. Liu, J. Sun, and J.-S. Dong. Parameter synthesis for hierarchical concurrent real-time systems. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:253–262, 2012. 6.5.2
- [30] É. André and R. Soulat. *The Inverse Method*. ISTE Ltd and John Wiley & Sons Inc., 2013. 3.5
- [31] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011. B.1.1.7

- [32] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168. IEEE, 2008. 8.5
- [33] D. Ardagna and B. Pernici. Global and local QoS guarantee in Web service selection. In *BPM Workshops*, 2005. 1.1, 3.5, 4, 4.1.4
- [34] D. Ardagna and B. Pernici. Global and local qos guarantee in web service selection. In *Proceedings of the Third international conference on Business Process Management, BPM'05*, pages 32–46, 2006. 6.5.2
- [35] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, 2007. 4, 4.5
- [36] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Formal verification of bpm4ws business collaborations. In *EC-Web*, pages 76–85, 2004. 6.5.1, 7, 7.1.5
- [37] D. Athanasopoulos, A. Zarras, and P. Vassiliadis. Service selection for happy users: making user-intuitive quality abstractions. In *SIGSOFT FSE*, pages 32–35, 2012. 4.5
- [38] L. Baresi and S. Guinea. Self-supervising BPEL processes. *IEEE Transactions on Software Engineering*, 37(2):247–263, 2011. 3.5, 8.1, 8.5
- [39] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In *CAiSE*, pages 491–503, 2005. 5.3
- [40] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. A.2.3
- [41] T. Berger, S. She, R. Lotufo, K. Czarnecki, T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the systems software domain. Technical report, University of Waterloo, 2012. B.1.1.3
- [42] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Feature-to-code mapping in two large product lines. Technical report, University of Waterloo, 2010. B.1.1.3
- [43] M. Berkelaar, K. Eikland, and P. Notebaert. Open source (mixed-integer) linear programming system. <http://lpsolve.sourceforge.net/>. 4.3
- [44] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 750–759, 2005. 6.5.1
- [45] D. Bianculli, D. Giannakopoulou, and C. S. Pasareanu. Interface decomposition for service compositions. In *ICSE*, pages 501–510, 2011. 3.5
- [46] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl. Variability issues in software product lines. In *PFE*, 2001. 5.2.2

- [47] T. Bultan, X. Fu, and J. Su. Tools for automated verification of web services. In *Automated Technology for Verification and Analysis: Second International Conference, ATVA 2004, Taipei, Taiwan, ROC, October 31-November 3, 2004. Proceedings*, pages 8–10, 2004. 6, 6.5.1
- [48] T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, 2006. 6.5.1
- [49] V. Cardellini, E. Casalicchio, V. Grassi, F. L. Presti, and R. Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *ESEC/SIGSOFT FSE*, pages 131–140, 2009. 4.5
- [50] J. Cardoso, J. Miller, A. Sheth, and J. Arnold. Quality of service for workflows and web service processes. *Journal of Web Semantics*, 1:281–308, 2004. 6.5.1
- [51] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *SIGSOFT FSE*, pages 237–246. ACM, 2010. 8.5
- [52] A. Carzaniga, A. Gorla, and M. Pezzè. Healing web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, 2008. 8.4
- [53] M. Chen, T. H. Tan, J. Sun, Y. Liu, and J. S. Dong. Veriws: a tool for verification of combined functional and non-functional requirements of web service composition. In *ICSE Companion*, pages 564–567, 2014. 1.3, 6.6
- [54] M. Chen, T. H. Tan, J. Sun, Y. Liu, J. Pang, and X. Li. Verification of functional and non-functional requirements of web service composition. In *ICFEM*, pages 313–328, 2013. 1.3, 7, 7.1.2, 7.1.3, 8.5
- [55] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *RTSS*, pages 80–89. IEEE Computer Society, 2008. 3.5
- [56] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *Proceedings of the 2008 Real-Time Systems Symposium, RTSS '08*, pages 80–89, Washington, DC, USA, 2008. IEEE Computer Society. 6.5.2
- [57] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000. 5.3
- [58] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000. 2.3, 6, 7
- [59] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, Aug. 2001. 5.1.1
- [60] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, Oct. 1992. 6.3.1
- [61] K. Czarnecki and U. W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. 5.1.2

- [62] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *RTSS*, pages 73–81. IEEE Computer Society, 1996. A.2.3
- [63] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008. 3.4.3.2, A.3.4, B.1.3
- [64] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002. 5.1.3, 2, B.1.1.2
- [65] G. Denaro, M. Pezzè, and D. Tosi. Designing self-adaptive service-oriented applications. In *Fourth International Conference on Autonomic Computing (ICAC’07), Jacksonville, Florida, USA, June 11-15, 2007*, page 16, 2007. 8.5
- [66] G. Denaro, M. Pezzè, and D. Tosi. SHIWS: A self-healing integrator for web services. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume*, pages 55–56, 2007. 8.5
- [67] J.-F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In *WCET*, 2005. 3.2
- [68] G. Dobson. Using WS-BPEL to implement software fault tolerance for Web services. In *EUROMICRO-SEAA*, pages 126–133. IEEE, 2006. 8.5
- [69] J. S. Dong, Y. Li, and H. H. Wang. TCOZ approach to semantic web services design. In *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 442–443, 2004. 9.2
- [70] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Towards verification of computation orchestration. *Formal Asp. Comput.*, 26(4):729–759, 2014. 7.1.3
- [71] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011. B.1.1.1
- [72] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba. On the effect of the steady-state selection scheme in multi-objective genetic algorithms. In *EMO*, pages 183–197, 2009. 5.1.3, 3
- [73] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, 2005. 8
- [74] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007. 3.3.6, 8.1
- [75] S. Ferndrigger, A. Bernstein, J. S. Dong, Y. Feng, Y. Li, and J. Hunter. Enhancing semantic web services with inheritance. In *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, pages 162–177, 2008. 9.2
- [76] A. Ferrara. Web services: a process algebra approach. In *ICSOC*, pages 242–251, 2004. 6.5.1

- [77] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005. 9.2
- [78] H. Foster. *A rigorous approach to engineering Web service compositions*. PhD thesis, Citeseer, 2006. 3, 6.5.1, 6.5.2, 7.1.1, 8.2.2
- [79] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE*, pages 152–163, 2003. 6.5.1
- [80] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *ICSE*, pages 771–774, 2006. 3.5, 6.5.1
- [81] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based analysis of obligations in web service choreography. In *AICT/ICIW*, page 149, 2006. 6.5.1
- [82] H. Foster, S. Uchitel, J. Magee, and J. Kramer. WS-Engineer: A model-based approach to engineering web service compositions and choreography. In *Test and Analysis of Web Services*, pages 87–119. 2007. 6, 7, 7.1.5
- [83] L. Fribourg, D. Lesens, P. Moro, and R. Soulat. Robustness analysis for scheduling problems using the inverse method. In *TIME*, pages 73–80. IEEE Computer Society Press, 2012. 3.5
- [84] X. Fu. *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University Of California, Santa Barbara, 2004. 6.5.1
- [85] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *CIAA*, pages 188–200, 2003. 6.5.1
- [86] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW*, pages 621–630, 2004. 6.5.1
- [87] X. Fu, T. Bultan, and J. Su. Model checking xml manipulating software. In *ISSTA*, pages 252–262, 2004. 7.1.5
- [88] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web services. In *CAV*, pages 510–514, 2004. 6.5.1, 7, 7.1.5
- [89] C. K. Fung, P. C. K. Hung, G. Wang, R. C. Linger, and G. H. Walton. A study of service composition with qos management. In *Proceedings of the IEEE International Conference on Web Services, ICWS '05*, pages 717–724. IEEE Computer Society, 2005. 6, 6.5.1
- [90] M. Gen, R. Cheng, and D. Wang. Genetic algorithms for solving shortest path problems. In *Evolutionary Computation*, pages 401–406. IEEE, 1997. 8.3.4
- [91] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *ICSE*, pages 33–42, 2013. 8.5
- [92] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. Simple object access protocol (SOAP) version 1.2. <http://www.w3.org/TR/soap12/>. 1

- [93] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12):2208–2221, 2011. 5, 5.3
- [94] J. Guo, E. Zulkoski, R. Olaechea, D. Rayside, K. Czarnecki, S. Apel, and J. M. Atlee. Scaling exact multi-objective combinatorial optimization by parallelization. In *ASE*, 2014. 5.3
- [95] I. Gurobi Optimization. Gurobi optimizer reference manual. <http://www.gurobi.com>. 4.3, 4.4
- [96] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemare. Runtime verification of web service interface contracts. *IEEE Computer*, 43(3):59–66, 2010. 6, 6.5.1
- [97] D. Harel and A. Naamad. The statestate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996. 7.1.5
- [98] M. Harman. The current state and future of search based software engineering. In *FOSE*, pages 342–357, 2007. 5.3
- [99] M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001. 5.3
- [100] G. P. Inc. 36 human-competitive results produced by genetic programming. <http://www.genetic-programming.com/>, 2012. 8
- [101] I. Jacobson, M. L. Griss, and P. Jonsson. *Software reuse - architecture, process and organization for business*. Addison-Wesley-Longman, 1997. 5
- [102] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. <http://www.oasis-open.org/specs/#wsbpelev2.0>, Apr 2007. 2.2, 6, 7
- [103] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, November 1990. 5, 5.1.1, 5.1.2
- [104] R. Karim, C. Ding, and C.-H. Chi. An enhanced promethee model for qos-based web service selection. In *SCC*, pages 536–543, 2011. 4.5
- [105] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc programming language. In *FMOODS/FORTE*, pages 1–25, 2009. 7.1.5
- [106] S. Koizumi and K. Koyama. Workload-aware business process simulation with statistical service analysis and timed petri net. In *ICWS '07*, pages 70–77. IEEE CS, 2007. 6, 6.5.1
- [107] M. Koshkina. Verification of Business Processes for Web Services. Master’s thesis, York University, Toronto, Ontario, 2003. 6.5.1
- [108] M. Koshkina and F. van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004. 6.5.1

- [109] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating service resource consumption from response time measurements. In *VALUETOOLS*, page 48, 2009. 3.5
- [110] T. T. H. Le, L. Palopoli, R. Passerone, Y. Ramadian, and A. Cimatti. Parametric analysis of distributed firm real-time systems: A case study. In *ETFA*, pages 1–8, 2010. 3.5
- [111] Y. Li, T. H. Tan, and M. Chechik. Management of time requirements in component-based systems. In *FM*, pages 399–415, 2014. 3.5, 7.1.5
- [112] N. Lohmann. A feature-complete petri net semantics for ws-bpel 2.0. In *WS-FM*, pages 77–91, 2007. 6.5.1
- [113] J. Magee and J. Kramer. *Concurrency - state models and Java programs* (2. ed.). Wiley, 2006. 3.5
- [114] N. Markey. Robustness in real-time systems. 6.5.2
- [115] A. Martens. On usability of web services. In *Proceedings of 4th International Conference on Web Information Systems Engineering Workshops*, pages 182–190. IEEE, 2003. 6.5.1
- [116] A. Martens. Analyzing web service based business processes. In *FASE*, pages 19–33, 2005. 6.5.1
- [117] A. Martens. Consistency between executable and abstract processes. In *EEE*, pages 60–67, 2005. 6.5.1
- [118] A. Martens. On compatibility of web services. In *Petri Net Newsletter*, pages 65:12–20. October 2003. 6.5.1
- [119] A. Martens and S. Moser. Diagnosing sca components using wombat. In *Business Process Management*, pages 378–388, 2006. 6.5.1, 7, 7.1.5
- [120] D. A. Menascé. Response-time analysis of composite Web services. *IEEE Internet Computing*, 8(1):90–92, 2004. 3.5
- [121] M. Mendonça, T. T. Bartolomei, and D. D. Cowan. Decision-making coordination in collaborative product configuration. In *SAC*, pages 108–113, 2008. B.1.1.3
- [122] M. Mendonça, M. Branco, and D. D. Cowan. S.P.L.O.T.: software product lines online tools. In *OOPSLA Companion*, pages 761–762, 2009. 5, B.1.1.3, B.1.2
- [123] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In *WWW 2008*, pages 815–824, 2008. 3.3.6, 3.5, 6.3.5
- [124] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *ARCS*, pages 124–138, 2005. 8.5
- [125] S. Nakajima. Lightweight formal analysis of web service flows. *Progress in Informatics*, 2:57–76, 2005. 6
- [126] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *Int. J. Intell. Syst.*, 24(7):726–746, 2009. 5.1.3, 4

- [127] A. Nöhner and A. Egyed. Conflict resolution strategies during product configuration. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, pages 107–114, 2010. 5
- [128] OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee. Web services business process execution language version 2.0, Apr. 2007. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>. 7.1.3
- [129] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE Computer Society, 2007. 5.3
- [130] C. Pasareanu, D. Giannakopoulou, and D. Bianculli. Interface decomposition for service compositions. 2011. 6.5.2
- [131] D. Pisinger. *Algorithms for knapsack problems*. PhD thesis, University of Copenhagen, 1995. 4.1.4
- [132] R. Pohl, K. Lauenroth, and K. Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *ASE*, pages 313–322, 2011. 5.3
- [133] R. Pohl, V. Stricker, and K. Pohl. Measuring the structural complexity of feature models. In *ASE*, pages 454–464, 2013. 5.3
- [134] C. Quinton, D. Romero, and L. Duchien. Automated selection and configuration of cloud environments using software product lines principles. In *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 144–151, 2014. 4.6
- [135] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS*, pages 43–, 2004. 6.5.1
- [136] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Optimum feature selection in software product lines: Let your model and values guide your search. In *CMSBSE*, pages 22–27, 2013. 5, 5.3, B.1.1.1, 3, B.1.1.7
- [137] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel’s back. In *ASE*, 2013. 5, 5.3, 5.4, B.1.1.3, 2, B.1.3
- [138] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: a case study in software product lines. In *ICSE*, pages 492–501, 2013. 5, 5.3, B.1.1.1, B.1.1.3, B.1.1.4, 3, B.1.2
- [139] A. Schrijver. *Theory of linear and integer programming*. John Wiley and Sons, 1986. A.1.1
- [140] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE*, pages 461–470, 2011. 5, B.1.1.3
- [141] J. Simmonds. *Dynamic Analysis of Web Services*. PhD thesis, University of Toronto, 2011. 8.4

- [142] J. Simmonds, S. Ben-David, and M. Chechik. Guided recovery for Web service applications. In *SIGSOFT FSE*, pages 247–256, 2010. 3.5, 8, 8.4, 8.5
- [143] J. Simmonds and M. Chechik. Rumor: monitoring and recovery for BPEL applications. In *ASE*, pages 345–346, 2010. 6.5.2, 8
- [144] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994. 5.2.4.1
- [145] C. Stahl. *Transformation von BPEL4WS in Petrinetze*. PhD thesis, Humboldt-Universität zu Berlin, 2006. 6.5.1
- [146] F. Stephanie. Genetic algorithms – Principles of natural selection applied to computation. *Science*, 261(5123):872–878, 1993. 8.3.1
- [147] R. Storn and K. Price. Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997. 9.2
- [148] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André. Modeling and verifying hierarchical real-time systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology*, 22(1):3.1–3.29, 2013. A.2, A.2.3, A.2.3.1, A.2.3.2
- [149] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, pages 702–708, Grenoble, France, June 2009. 6.3.1, 7.1.3
- [150] J. Sun, Y. Liu, J. S. Dong, G. Pu, and T. H. Tan. Model-based methods for linking web service choreography and orchestration. In *APSEC*, pages 166–175, 2010. 7
- [151] T. H. Tan. Towards verification of a service orchestration language. In *Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010 - Companion Volume*, pages 36–37, 2010. 7.1.5
- [152] T. H. Tan, É. André, M. Chen, J. Sun, Y. Liu, J. S. Dong, and L. Yuan. Automated synthesis of local time requirement for service composition. *Submitted to TSE*. 1.3
- [153] T. H. Tan, É. André, J. Sun, Y. Liu, J. S. Dong, and M. Chen. Dynamic synthesis of local time requirement for service composition. In *ICSE*, pages 542–551, 2013. 4.5, 7.1.5, 8.5
- [154] T. H. Tan, M. Chen, É. André, J. Sun, Y. Liu, and J. S. Dong. Automated runtime recovery for qos-based service composition. In *WWW*, pages 563–574, 2014. 1.3
- [155] T. H. Tan, M. Chen, J. Sun, Y. Liu, and J. S. Dong. Dynamic ranking optimization for qos-aware service composition. *Submitted to ICFEM*. 1.3
- [156] T. H. Tan, Y. Liu, J. Sun, and J. S. Dong. Verification of orchestration systems using compositional partial order reduction. In *ICFEM*, 2011. 7.1.5
- [157] T. H. Tan, Y. Xue, M. Chen, J. Sun, Y. Liu, and J. S. Dong. Optimizing selection of competing features via feedback-directed evolutionary algorithms. *Accepted to ISSTA 2015*. 1.3

- [158] L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009. 3.5
- [159] L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric model-checking of stopwatch petri nets. 15(17):3273–3304, 2009. 6.5.2
- [160] J. Wang, Y. Xue, Y. Liu, and T. H. Tan. JSDC: A hybrid approach for javascript malware detection and classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 109–120, 2015. 9.2
- [161] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *ICSE*, pages 364–374. IEEE, 2009. 8.5
- [162] J. White, B. Dougherty, and D. C. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009. 5.3
- [163] H. Xiao, B. Chan, Y. Zou, J. W. Benayon, B. O’Farrell, E. Litani, and J. Hawkins. A framework for verifying sla compliance in composed services. *ICWS ’08*, pages 457–464, Washington, DC, USA, 2008. IEEE Computer Society. 6.5.1
- [164] Y. Xue, Z. Xing, and S. Jarzabek. Understanding feature evolution in a family of product variants. In *WCRE*, pages 109–118, 2010. B.1.1.3
- [165] S. S. Yau and Y. Yin. Qos-based service ranking and selection for service-based systems. In *IEEE SCC*, pages 56–63, 2011. 4.5
- [166] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ESEC/FSE ’11*, pages 26–36. ACM, 2011. 6, 7
- [167] K. Yoon and C. Hwang. *Multiple attribute decision making: An introduction*. Sage Publications, Incorporated, 1995. 4.1.3, 22
- [168] T. Yu, Y. Zhang, and K. Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *TWEB*, 1(1):6, 2007. 3.5, 4.5
- [169] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1), May 2007. 6.5.2
- [170] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven Web services composition. In *WWW*, pages 411–421, 2003. 4.5
- [171] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for Web services composition. *Software Engineering, IEEE Transactions on*, 30(5):311–327, 2004. 4.5, 7, 7.1.5
- [172] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In *PPSN*, pages 832–842, 2004. 5, 5.1.3, 1

- [173] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. Evolutionary Computation*, 3(4):257–271, 1999.
- B.1.1.2

Appendix A

Appendix of Chapter 3

This chapter introduces the necessary definitions and terminologies, the formal model for parametric composite services and the synthesis algorithms for dLTC, which are complementary for Chapter 3.

A.1 A Formal Model for Parametric Composite Services

A composite service CS makes use of a finite number of component services to accomplish a task. Let $C = \{s_1, \dots, s_n\}$ be the set of all component services that are used by CS . The response time of a composite service is largely reflected on the time spending on the communication activities. In this work, we assume that the response time of a composite service is the sum of the time spent on individual communication activities, and the time incurred by internal operations of the composite service is negligible (See Section 6.3.5 for discussion on time incurred for internal operations).

For example, assume that the only communication activity that communicates with component service S is the synchronous invocation activity $sInv(S)$. Upon invoking of service S , the activity $sInv(S)$ waits for the reply. The response time of S is equivalent to the waiting time in $sInv(S)$. Therefore, by analyzing the time spent in $sInv(S)$, we can get the response time of component service S . Given a composite service CS , let $t_i \in \mathbb{R}_{\geq 0}$ be the response time of component service S_i for $i \in \{1, \dots, n\}$, and let $C_t = \{t_1, \dots, t_n\}$ be a set of component service response times that fulfill the global time requirement of service CS . Because t_i , for $i \in \{1, \dots, n\}$, is a real number, there are infinitely many possible values, even in a bounded

interval (and even if one restricts to rational numbers). A method to tackle this problem is to reason *parametrically*, by considering these response times as unknown constants, or *parameters*. Let $u_i \in \mathbb{R}_{\geq 0}$ be the parametric response time of component service S_i for $i \in \{1, \dots, n\}$, and let $C_u = \{u_1, \dots, u_n\}$ be the set of component service parametric response times. Using constraints on C_u , we can represent an infinite number of possible response times symbolically. The local time requirement of composite service CS is specified as a constraint over C_u . An example of local time requirement is $(u_1 \leq 6) \wedge (u_2 \leq 5)$. This local time requirement specifies that, in order for CS to satisfy the global time requirement, service S_1 needs to respond within 6 time units, and service S_2 needs to respond within 5 time units. The local time requirement could contain dependency between parametric response times, an example is $(u_2 \leq u_1 \implies u_1 + u_2 \leq 6) \wedge (u_1 \leq u_2 \implies u_1 \leq 6)$. The example requires that if service S_2 responses not slower than service S_1 , then the summation of response times of services S_1 and S_2 are required to be within 6 seconds; if service S_1 responses not slower than service S_2 , then service S_1 is required to response within 6 seconds.

In the following subsection, we introduce notations used throughout the paper.

A.1.1 Variables, Clocks, Parameters, and Constraints

We assume a finite set \mathcal{Var} of finite-domain *variables*. Given $Var \subset \mathcal{Var}$, a *variable valuation* for Var is a function assigning to each variable a value in its domain. We denote by $\mathcal{V}(Var)$ the set of all variable valuations.

Clocks are variables (disjoint with \mathcal{Var}) with values in the set of non-negative real numbers $\mathbb{R}_{\geq 0}$. A clock is used to record the time passing of activities. All clocks are progressing at the same rate. \mathcal{X} is defined as a universal set of clocks. Let $X = \{x_1, \dots, x_H\} \subset \mathcal{X}$ (for some integer H) be a finite set of clocks. A *clock valuation* is a function $w : X \rightarrow \mathbb{R}_{\geq 0}$, that assigns a non-negative real value to each clock.

A *parameter* is an unknown constant. Let \mathcal{U} denote the universal set of parameters, disjoint with \mathcal{X} and \mathcal{Var} . Given a finite set of parameters $U = \{u_1, \dots, u_m\} \subset \mathcal{U}$ (for some integer m), a *parameter valuation* is a function $\pi : U \rightarrow \mathbb{R}_{\geq 0}$ assigning a non-negative real value to each parameter. We can identify a valuation π with the point $(\pi(u_1), \dots, \pi(u_m))$.

A *linear term* over $X \cup U$ is an expression of the form $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ for some $N \in \mathbb{N}$, with $z_i \in X \cup U$, $\alpha_i \in \mathbb{R}_{\geq 0}$ for $1 \leq i \leq N$, and $d \in \mathbb{R}_{\geq 0}$. We denote by $\mathcal{L}_{X \cup U}$ the set of all linear terms

over X and U respectively. Similarly, we define \mathcal{L}_X and \mathcal{L}_U . Given $X \subset \mathcal{X}$ and $U \subset \mathcal{U}$, an *inequality* over X and U is $e < e'$ with $< \in \{<, \leq\}$, where $e, e' \in \mathcal{L}_{X \cup U}$.

A *convex constraint* (or *constraint*) is a conjunction of inequalities. We denote by $C_{X \cup U}$ be the set of all convex constraints over X and U respectively. Similarly, we define C_X and C_U . A *non-necessarily convex constraint* (or NNCC) is a conjunction of disjunction of inequalities¹. Note that the negation of an inequality remains an inequality; however, the negation of a convex constraint becomes (in the general case) an NNCC. We denote by \mathcal{NC}_U the set of all NNCCs over U . Henceforth, we use w (resp. π) to denote a clock (resp. parameter) valuation. Let $C \in \mathcal{NC}_U$, $C[\pi]$ denotes the constraint over X obtained by replacing each $u \in U$ with $\pi(u)$ in C . We write $\pi \models C$, if $C[\pi]$ evaluates to true. C is *empty*, if there does not exist a parameter valuation π , such that $\pi \models C$; otherwise C is *non-empty*. We define $C^\dagger = \{x + d \mid x \in C \wedge d \in \mathbb{R}_{\geq 0}\}$, as *time elapsing* of C , i.e., the constraint over X and U obtained from C by delaying an arbitrary amount of time d . Given two constraints $C_1, C_2 \in \mathcal{NC}_U$, C_1 is *included* in C_2 , denoted by $C_1 \subseteq C_2$, if $\forall \pi : \pi \models C_1 \Rightarrow \pi \models C_2$. Similarly, C_1 is *strictly included* in C_2 , denoted by $C_1 \subset C_2$, if $C_1 \subseteq C_2$ and $C_1 \neq C_2$. Given $C \in C_{X \cup U}$ and $X' \subseteq X$, we denote by $\text{prune}_{X'}(C)$ the constraint in $C_{X \cup U}$ that is obtained from C by pruning the clocks in X' ; this can be achieved using variable elimination techniques such as Fourier-Motzkin (see, e.g., [139]).

A.1.2 Syntax of Composite Services

Composite services are expressed using *processes*. We define a formal syntax definition in Figure A.1, where S is a component service, P and Q are composite service processes, b is a Boolean expression, and $a \in \mathbb{R}_{>0}$ is a non-negative real number.

We informally describe the BPEL syntax notations below:

- $\text{rec}(S)$ and $\text{reply}(S)$ are used to denote “receive from” and “reply to” a service S , respectively;
- $s\text{Inv}(S)$ (resp. $a\text{Inv}(S)$) denotes the synchronous (resp. asynchronous) invocation of a component service S ;
- $P \parallel Q$ denotes the concurrent composition of BPEL activities P and Q ;

¹Without loss of generality, we assume here that all NNCCs are in conjunctive normal form (CNF).

$P := \text{rec}(S)$	receive activity
$\text{reply}(S)$	reply activity
$s\text{Inv}(S)$	synchronous invocation
$a\text{Inv}(S)$	asynchronous invocation
$P \parallel Q$	concurrent activity
$P ; Q$	sequential activity
$P \triangleleft b \triangleright Q$	conditional activity
$\text{pick}(S \Rightarrow P, \text{alarm}(a) \Rightarrow Q)$	pick activity

Figure A.1: Syntax of composite service processes

- $P ; Q$ denotes the sequential composition of BPEL activities P and Q ;
- $P \triangleleft b \triangleright Q$ denotes the conditional composition, where b is a guard condition. If b is evaluated as true, BPEL activity P is executed, otherwise activity Q is executed;
- $\text{pick}(S \Rightarrow P, \text{alarm}(a) \Rightarrow Q)$ denotes the BPEL *pick* composition, which contains two branches of activities: *onMessage* activity and *onAlarm* activity, where either branch of the activity will be executed. *onMessage* activity is activated when the message from service S arrives within a seconds, where $a \in \mathbb{R}_{>0}$, and BPEL activity P is subsequently executed; *onAlarm* activity is activated at a seconds, and BPEL activity Q is subsequently executed. If the message arrives at exactly a seconds, then P or Q executes non-deterministically. Given a *pick* activity P , we use $P.\text{onMessage}$ and $P.\text{onAlarm}$ to denote the *onMessage* and *onAlarm* branches of P respectively.

A *structural activity* is an activity that contains other activities. Concurrent, sequential, conditional, and pick activities are examples of structural activity. An activity that does not contain other activities is called an *atomic activity*, which includes receive, reply, synchronous invocation and asynchronous invocation activities.

A.1.3 Parametric Composite Services

Definition 13 (Composite Service Model). A composite service model CS is a tuple (Var, V_0, N_0) , where Var is a finite set of variables, V_0 is an initial valuation that maps each variable to its initial value, and N_0 is the composite service process.

We now extend the definitions of services, composite service processes and composite service model to the parametric case. That is, the response time of a parametric service i is

now a parameter $u_i \in U$. Then, a parametric composite service process is a service process whose services (“ S ” in Figure A.1) are parametric services. We denote by \mathcal{P} the set of all possible parametric composite service processes. We can now extend composite service models to the parametric case, by replacing composite service processes with parametric composite service processes.

Definition 14 (Parametric Composite Service Model). A parametric composite service model CS is a tuple (Var, V_0, U, P_0, C_0) , where Var is a finite set of variables; $V_0 \in \mathcal{V}(Var)$ is an initial valuation that maps each variable to its initial value; U is a finite set of parameters; $P_0 \in \mathcal{P}$ is the parametric composite service process; and $C_0 \in C_U$ is the initial (convex) parametric constraint.

Given a service model CS with a parameter set $U = \{u_1, \dots, u_m\}$, and given a parameter valuation $(\pi(u_1), \dots, \pi(u_m))$, $CS[\pi]$ denotes the *valuation* of CS with π , viz., the model (Var, V_0, U, P_0, C) , where C is $C_0 \wedge \bigwedge_{i=1}^m (u_i = \pi(u_i))$. Note that $CS[\pi]$ can be seen as a non-parametric service model $(Var, V_0, P_0[\pi])$, where $P_0[\pi]$ corresponds to P_0 where each occurrence of a parameter u_i in a service has been replaced with its valuation $\pi(u_i)$.

A.1.4 Bad Activity

Given a BPEL service CS , we define a *bad activity* as an atomic activity such that its execution signaling that the composite service CS has violated the global time requirement. In the case of the SMIS example, it is the reply activity that is triggered once after the component service PS fails to respond within one second.

An activity (include both atomic and structural activities) that is not a bad activity is called a *good activity*. To distinguish the bad activities, we allow the user to annotate a BPEL activity A as a bad activity, denoted by $[A]_{bad}$. The annotation can be achieved, for example, by using extension attribute of BPEL activities. The execution of activity $[A]_{bad}$ will result the LTS of CS to end in an undesired terminal state, which we denote as a *bad state*. A terminal state which is not a bad state is called a *good state*. The synthesized local time requirement needs to guarantee the avoidance of all bad states and the reachability of at least one good state.

A.2 A Formal Semantics for Parametric Composite Services

In this section, we provide our parametric composite service model with a formal semantics, given in the form of a labeled transition system (LTS). The semantics we use is inspired by the one proposed for (parametric) stateful timed Communicating Sequential Processes (CSP) [148, 28], that makes use of implicit clocks.

We first recall LTS (Section A.2.1) and define symbolic states (Section A.2.2). Following that, we define implicit clocks and the associated functions, i.e., activation and idling (Section A.2.3). We then introduce our formal semantics (Section A.2.4), and give an application to an example (Section A.2.5).

A.2.1 Labeled Transition Systems

Definition 15 (Labeled Transition System (LTS)). *A labeled transition system (LTS) of a composite service CS is a tuple $\mathcal{LCS} = (Q, s_0, \Sigma, \delta)$, where*

- Q is a set of states;
- $s_0 \in Q$ is the initial state;
- Σ is the universal set of actions;
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation.

Given an LTS of a composite service CS, $\mathcal{LCS} = (Q, s_0, \Sigma, \delta)$, a state $s \in Q$ is a *terminal state* if there does not exist a state $s' \in Q$ and an action $a \in \Sigma$ such that $(s, a, s') \in \delta$; otherwise, s is said to be a *non-terminal state*. There is a *run* from a state s to state s' , where $s, s' \in Q$, if there exist a set of states $\{s_1, \dots, s_n\} \subseteq Q$ and a set of actions $\{a_1, \dots, a_n\} \subseteq \Sigma$ such that $s_1 = s$, $s_n = s'$, and $\forall i \in \{1, \dots, n-1\}, (s_i, a_i, s_{i+1}) \in \delta$. A *complete run* is a run that starts in the initial state s_0 and ends in a terminal state. Given a state $s \in Q$, we use $\text{succ}(s)$ to denote the set of states reachable from s ; formally, $\text{succ}(s) = \{s' \mid \exists a \in \Sigma : s' \in Q \wedge (s, a, s') \in \delta\}$. A *sub-LTS* of \mathcal{LCS} is an LTS that starts from a state $s \in Q$. We denote the sub-LTS that starts from the state $s \in Q$ of a composite service CS as $\text{subLTS}_{\mathcal{LCS}, s}$.

Definition 16 (sub-LTS). $\text{subLTS}_{\mathcal{LCS}, s} = (Q', s, \Sigma', \delta')$, where $Q' \subseteq Q$ is the set of states that are reachable from $s \in Q$ in \mathcal{LCS} . For $\delta' \subseteq \delta$, it is the transition relation that satisfies the following condition: $s_1 \xrightarrow{a} s_2 \in \delta'$ if $s_1, s_2 \in Q'$ and $s_1 \xrightarrow{a} s_2 \in \delta$. $\Sigma' \subseteq \Sigma$ is the set of all actions that are used in δ' .

A.2.2 Symbolic States

We define here a symbolic state for a parametric composite service model.

Definition 17 (State). A state $s \in Q$ is a tuple (V, P, C, D) , where V is a valuation of the variables (i.e., a function that maps a variable name to its value), and we write $v = \perp$ to denote a variable $v \in \text{Var}$ that is uninitialized; P is a composite service process; C is a constraint over $C_{X \cup U}$; and $D \in \mathcal{L}_U$ is the (parametric) duration from the initial state s_0 to the beginning of the state s .

Given a state $s = (V, P, C, D)$, we use the notation $s.V$ to denote the component V of s , and similarly for $s.P$, $s.C$ and $s.D$. When a parametric composite service model CS has no variable, we denote each state $s \in Q$ as (P, C, D) for the sake of brevity. Given the initial state s_0 of a composite service CS , we denote $s_0.P$ as the *initial activity*.

A.2.3 Implicit Clocks

In order to analyze the LTS with real-time semantics, we use *clocks* to record the elapsing of time. Clocks are used to record the time elapsing in several formalisms, in particular in timed automata (TA) [23]. In TAs, the clocks are defined as part of the models and state space. It is known that the state space of the system could grow exponentially with the number of clocks and that the fewer clocks, the more efficient real-time model checking is [40]. In timed automata, it is then possible to dynamically reduce the number of clocks [62]. An alternative approach is to define a semantics that creates clocks on the fly when necessary, and prunes them when no longer needed. This approach was initially proposed for (parametric) stateful timed Communicating Sequential Processes (CSP) [148, 28]. This allows smaller state space compared to the explicit clock approach; we refer to this second approach as the *implicit clock approach*. We use here the implicit clock approach.

A.2.3.1 Clock Activation

Clocks are implicitly associated with timed processes. For instance, given a communication activity $sInv(S)$, a clock starts ticking once the activity becomes activated. To introduce clocks on the fly, we define an activation function Act in Fig. A.2, similar to the one defined in [148, 28]. Given a process P , we denote by P_x the corresponding process that has been

$Act(A(S), x)$	$= A(S)_x$	A1
$Act(mpick, x)$	$= mpick_x$	A2
$Act(A(S)_{x'}, x)$	$= A(S)_{x'}$	A3
$Act(mpick_{x'}, x)$	$= mpick_{x'}$	A4
$Act(P \oplus Q, x)$	$= Act(P, x) \oplus Act(Q, x)$	A5
$Act(P; Q, x)$	$= Act(P, x); Q$	A6

where $A \in \{rec, sInv, aInv, reply\}$, $\oplus \in \{\parallel, \triangleleft b \triangleright\}$,
and $mpick = pick(S \Rightarrow P_t, alrm(a) \Rightarrow P_a)$.

Figure A.2: Activation function

associated with clock x . The activation function will be called when a new state s is reached to assign a new clock for each newly activated communication activity. Rules A1 and A2 state that a new clock is associated with BPEL communication activity if it is newly activated. Rules A3 and A4 state that if a BPEL communication activity has already been assigned a clock, it will not be reassigned. Rules A5 and A6 state that function Act is applied recursively for activated child activities for BPEL structural activities.

Given a process P , we denote by $cl(P)$ the set of *active clocks* associated with P . For instance, the set of active clocks associated with process $P = rec(S)_x \parallel rec(S_1)_{x'}$ contains x and x' .

A.2.3.2 Idling Function

In Fig. A.3, we define the function *idle* that, given a state s , returns a constraint that specifies how long an activity can idle at state s . The result is a constraint over $X \cup U$. This function is inspired by its counterpart in [148, 28]. Rule I1 considers the situation when the communication requires waiting for the response of a component service S , and the value of clock x must not be larger than the response time parameter t_S of the service. Rule I2 considers the situation when no waiting is required. Rules I3 to I5 state that the function *idle* is applied recursively for activated child activities of a BPEL structural activity.

A.2.4 Operational Semantics

We now define the semantics of a parametric composite service model in the form of an LTS. Let $Y = \langle x_0, x_1, \dots \rangle$ be a sequence of clocks.

Definition 18. Let $CS = (Var, V_0, U, P_0, C_0)$ be a parametric composite service model. The se-

$idle(A(S)_x)$	$=$	$x \leq t_S$	I1
$idle(B(S)_x)$	$=$	$(x = 0)$	I2
$idle(P \oplus Q)$	$=$	$idle(P) \wedge idle(Q)$	I3
$idle(P; Q)$	$=$	$idle(P)$	I4
$idle(mpick_x)$	$=$	$x \leq t_S \wedge x \leq a$	I5

where $A \in \{rec, sInv\}$, $B \in \{aInv, reply\}$, $\oplus \in \{|||, \blacktriangleleft b \blacktriangleright\}$,
 $mpick = pick(S \Rightarrow P_t, alrm(a) \Rightarrow P_a)$, and t_S is the
parametric response time of service S .

Figure A.3: Idling function

mantics of CS , represented by \mathcal{LCS} , is an LTS (Q, s_0, Σ, δ) where

$$Q = \{(V, P, C, D) \in \mathcal{V}(Var) \times \mathcal{P} \times C_{X \cup U} \times \mathcal{L}_U\},$$

$$s_0 = (V_0, P_0, C_0, 0)$$

and the transition relation δ is the smallest transition relation satisfying the following. For all $(V, P, C, D) \in Q$, if x is the first clock in the sequence Y which is not in $cl(P)$, and $(V, Act(P, x), C \wedge x = 0, D) \xrightarrow{a} (V', P', C', D')$ where C' is satisfiable, then we have:
 $((V, P, C, D), a, (V', P', prune_{X \setminus cl(P')}(C'), D')) \in \delta$.

The transition relation \hookrightarrow is specified by a set of rules, given in Fig. A.4. Let us first explain some of these rules.

- Rule $rSInv$ states that a state $s = (V, sInv(S)_x, C, D)$ may evolve into the state $s' = (V', Stop, x = t_S \wedge C^\uparrow, D + t_S)$ via action $e \in \Sigma$, where $Stop$ is the activity that does nothing, and t_S is the parametric response time of component service S . Note that, from Definition 18, the condition $x = t_S \wedge C^\uparrow$ is necessarily satisfied (otherwise this evolution is not possible). The resulting constraint is obtained from C by applying time elapsing and intersection with the equality $x = t_S$; furthermore, the parametric duration from the initial state (D) is augmented with t_S . Rules $rSInv$, $rReply$ and $rAInv$ are similar.
- Rule $rPick1$ encodes the transition that takes place due to an *onMessage* activity. Let us explain the constraint $(x = t_S) \wedge idle(mpick_x) \wedge C^\uparrow$. First, after the transition, the current clock x needs to be equal to the parametric response time of service S , i.e., $x = t_S$. Second, the constraint $idle(mpick_x)$ is added to ensure that x remains smaller or equal to the maximum duration of the $mpick_x$ activity. Third, the constraint C^\uparrow represents the time elapsing of C . Rule $rPick2$ is similar.

$$\begin{array}{c}
\frac{}{(V, sInv(S)_x, C, D) \xrightarrow{e} (V', Stop, x = t_S \wedge C^\dagger, D + t_S)} [rSInv] \\
\frac{}{(V, rec(S)_x, C, D) \xrightarrow{e} (V', Stop, x = t_S \wedge C^\dagger, D + t_S)} [rRec] \\
\frac{}{(V, reply(S)_x, C, D) \xrightarrow{e} (V', Stop, x = 0 \wedge C^\dagger, D)} [rReply] \\
\frac{}{(V, aInv(S)_x, C, D) \xrightarrow{e} (V', Stop, x = 0 \wedge C^\dagger, D)} [rAInv] \\
\frac{let\ mpick = pick(S \Rightarrow P_t, alm(a) \Rightarrow P_a)}{(V, mpick_x, C, D) \xrightarrow{e} (V', P_t, (x = t_S) \wedge idle(mpick_x) \wedge C^\dagger, D + t_S)} [rPick1] \\
\frac{}{(V, mpick_x, C, D) \xrightarrow{e} (V', P_a, (x = a) \wedge idle(mpick_x) \wedge C^\dagger, D + a)} [rPick2] \\
\frac{V(b) = \perp}{(V, A \triangleleft b \triangleright B, C, D) \xrightarrow{e} (V', A, C, D)} [rCond1] \\
\frac{V(b) = \perp}{(V, A \triangleleft b \triangleright B, C, D) \xrightarrow{e} (V', B, C, D)} [rCond2] \\
\frac{V(b) = true}{(V, A \triangleleft b \triangleright B, C, D) \xrightarrow{e} (V', A, C, D)} [rCond3] \\
\frac{V(b) = false}{(V, A \triangleleft b \triangleright B, C, D) \xrightarrow{e} (V', B, C, D)} [rCond4] \\
\frac{(V, A, C, D) \xrightarrow{e} (V', A', C', D'), A' \neq Stop}{(V, A; B, C, D) \xrightarrow{e} (V', A'; B, C', D')} [rSeq1] \\
\frac{(V, A, C, D) \xrightarrow{e} (V', Stop, C', D')}{(V, A; B, C, D) \xrightarrow{e} (V', B, C', D')} [rSeq2] \\
\frac{(V, A, C, D) \xrightarrow{e} (V', A', C', D')}{(V, A ||| B, C, D) \xrightarrow{e} (V', A' ||| B, C' \wedge idle(B), D')} [rFlow1] \\
\frac{(V, B, C, D) \xrightarrow{e} (V', B', C', D')}{(V, A ||| B, C, D) \xrightarrow{e} (V', A ||| B', C' \wedge idle(A), D')} [rFlow2]
\end{array}$$

Figure A.4: Set of rules for the transition relation \xrightarrow{e}

- Given a conditional composition $A \triangleleft b \triangleright B$, we denote by $V(b) \in \{true, false, \perp\}$ the evaluation of the guard condition b . We have that $V(b) = \perp$ when the evaluation of b is unknown, due to the fact that there may be uninitialized variables in b . Since b might be evaluated to either true or false at certain stages during runtime, we explore both activities A and B when $V(b) = \perp$ so as to reason about all possible scenarios. The case of $V(b) = \perp$ is captured by rules $rCond1$ and $rCond2$, and the cases where $V(b) \in \{true, false\}$ are captured by rules $rCond3$ and $rCond4$.
- $rSeq1$ states that if activity A' is not a *Stop* activity (i.e., activity A' has not finished its execution), then a state containing activity $A \rightarrow B$ could evolve into a state containing activity $A' \rightarrow B$. Otherwise, if A is a *Stop* activity (i.e., activity A has finished its execution), then a state containing activity $A \rightarrow B$ could discharge activity A and evolve into a state containing B . This is captured by $rSeq2$.
- For concurrent activity $A ||| B$, either activity A or activity B can be executed. This is captured by $rFlow1$ and $rFlow2$ respectively. $rFlow1$ states that if state (V, A, C, D) can evolve into (V', A', C', D') via action $e \in \Sigma$, then a state containing $A ||| B$ can evolve into a state containing $A' ||| B$ via action $e \in \Sigma$, if $C' \wedge idle(B)$ holds. That is, the clock constraints in C' could not exceed the duration where activity B can last for. Rule $rFlow2$ is similar.

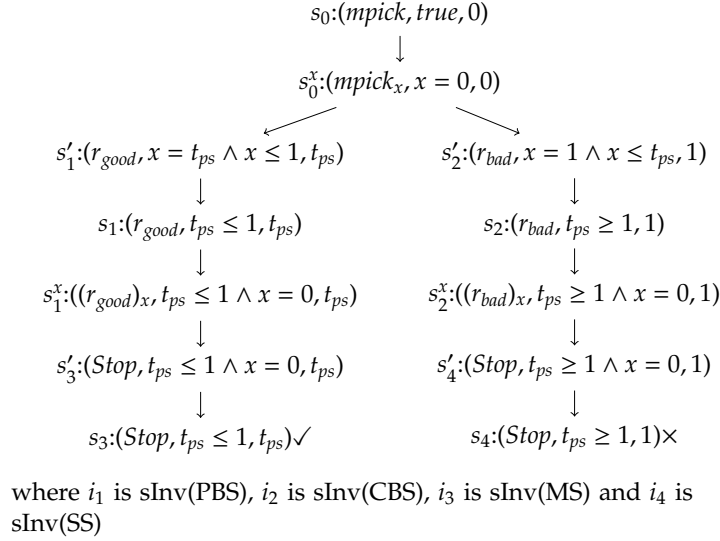


Figure A.5: LTS of service CS

Let us now explain Definition 18. Starting from the initial state $s_0 = (P_0, C_0, 0)$, we iteratively construct successor states as follows. Given a state (V, P, C, D) , a fresh clock x which is not currently associated with P is picked from Υ . The state (V, P, C, D) is transformed into $(V, \text{Act}(P, x), C \wedge x = 0, D)$, i.e., timed processes which just become activated are associated with x and C is conjuncted with $x = 0$. Then, a firing rule is applied to get a target state (V', P', C', D') . Lastly, clocks which do not appear within P' are pruned from C' . Observe that one clock is introduced and zero or more clocks may be pruned during a transition. In practice, a clock is introduced only if necessary; if the activation function does not activate any subprocess, this new clock is not created.

A.2.5 Application to an Example

Consider a composite service CS (which is a part of the SMIS example), the LTS of CS is shown in Figure A.5. Since CS has no variable, $V = \emptyset$ from all states; therefore, we omit the component V from all states for the sake of brevity.

- At state s_0 , the activation function assigns clock x to record time elapsing of pick activity mpick , with x initialized to zero. The tuple becomes the intermediate state $s_0^x = (\text{mpick}_x, x = 0, 0)$.

- From intermediate state s_0^x , it could evolve into the intermediate state s_1' by applying the rule $rPick1$, if the constraint $c_1 = (x = t_{PS} \wedge idle(mpick_x) \wedge (x = 0)^\uparrow)$, where $idle(mpick_x) = (x \leq t_{PS} \wedge x \leq 1)$ and $(x = 0)^\uparrow = x \geq 0$, is satisfiable. Intuitively, c_1 denotes the constraint where t_{PS} time units elapsed since clock x has started. In fact, c_1 is satisfiable (for example with $t_{PS} = 0.5$ and $x = 0.5$). Therefore, it could evolve into the intermediate state $s_1' = (r_{good}, x = t_{PS} \wedge idle(mpick_x) \wedge (x = 0)^\uparrow, t_{PS}) = (r_{good}, x = t_{PS} \wedge x \leq 1, t_{PS})$. Since clock x is not used anymore in $s_1'.P$ which is r_{good} , it is pruned. After pruning of clock variable x and simplification of the expression, the intermediate state s_1' becomes the state $s_1 = (r_{good}, t_{PS} \leq 1, t_{PS})$.
- From intermediate state s_0^x , it could evolve into the intermediate state s_2' , by applying the rule $rPick2$, if the constraint $c_2 = (x = 1 \wedge idle(mpick_x) \wedge (x = 0)^\uparrow)$, where $idle(mpick_x) = (x \leq t_{PS} \wedge x \leq 1)$ and $(x = 0)^\uparrow = x \geq 0$, is satisfiable. It is easy to see that c_2 is satisfiable; therefore, it could evolve into the intermediate state $s_2' = (r_{bad}, x = 1 \wedge x \leq t_{PS}, 1)$. After clock pruning from intermediate state s_2' , it becomes state $s_2 = (r_{bad}, t_{PS} \geq 1, 1)$.
- From state s_1 , activation function assigns clock x for reply activity r_{good} , and it evolves into intermediate state s_1^x . From intermediate state s_1^x , it could evolve into intermediate state s_3' by applying rule $rReply$, if the constraint $c_3 = (x = 0 \wedge (t_{PS} \leq 1)^\uparrow)$ is satisfiable, where $(t_{PS} \leq 1)^\uparrow = t_{PS} \leq 1$. In fact it is, and therefore it evolves into state $s_3' = (Stop, t_{PS} \leq 1 \wedge x = 0, t_{PS})$. After pruning of the used clock, it evolves into the terminal state $s_3 = (Stop, t_{PS} \leq 1, t_{PS})$. Since the terminal state is not caused by a bad activity, s_3 is considered as a good state, denoted by \checkmark in Figure A.5.
- From state s_2 , it could also evolve into terminal state $s_4 = (Stop, t_{PS} \geq 1, 1)$ with similar reason as above. Since the terminal state is caused by a bad activity, it is considered as a bad state, denoted by \times in Figure A.5.

Note that all states s_i^x and s_j' , where $i, j \in \mathbb{N}$ and $0 \leq i \leq 4$, are intermediate states. State s_i^x is the state s_i after clock assignment operations are applied. State s_j' is the state s_j before clock pruning operations are applied. These intermediate states are for illustrative purpose. They are not kept for the state space of composite service CS for space efficiency.

A.3 Synthesis of Design-time LTC

Given $CS = (Var, U, P_0, C_0)$, the *global time requirement* for CS requires that, for every state (V, P, C, D) reachable from the initial state $(V_0, P_0, C_0, 0)$ in the LTS, the constraint $D \leq T_G$ is satisfied, where $T_G \in \mathbb{R}_{\geq 0}$ is the *global time constraint*. The *local time requirement* requires that if the response times of all component services of CS satisfy the *local time constraint* (LTC) $C_L \in C_U$, then the service CS satisfies the global time requirement.

In this section, given a global time constraint T_G for a service CS , we present an approach to synthesize design-time LTC (dLTC) C_L based on the LTS. The dLTC will be given in the form of an NNCC over U . We show that if the response times of all component services of CS satisfy the local time requirement, the service CS would end in a good state within T_G time units.

A.3.1 Addressing the Good States

We assume a composite service CS and its LTS $L_{CS} = (Q, s_0, \Sigma, \delta)$; let Q_{good} be the set of all good states of service L_{CS} . In this section, we assume there are no bad states; we will discuss bad states in Section A.3.2.

Given L_{CS} , our goal is to synthesize the local time requirement for service CS . We make two observations here. First, a good state $s_g = (V_g, P_g, C_g, D_g) \in Q_{good}$ is reachable from the initial state s_0 iff C_g is satisfiable. Second, whenever the good state s_g is reached, we require that the total delay from initial state s_0 to state s_g must be no larger than the global time constraint T_G , i.e., $D_g \leq T_G$. To sum up, given a good state $s_g = (V_g, P_g, C_g, D_g)$ where $s_g \in Q_{good}$, we require the constraint $(C_g \implies (D_g \leq T_G))$ to hold. The constraint means that whenever state s_g is reachable from initial state s_0 , the total (parametric) delay from initial state s_0 to state s_g must be less than the global time constraint T_G . The synthesized dLTC for CS is the conjunction of such constraints for each good state $s_g \in Q_{good}$, i.e., $\bigwedge_{(V_g, P_g, C_g, D_g) \in Q_{good}} (C_g \implies (D_g \leq T_G))$.

Example

Let us consider a composite service CS whose process component is $pick(FS \Rightarrow i_1, alrm(1) \Rightarrow i_2)$ (henceforth referred to as $mpick$). Suppose the global time requirement of the composite

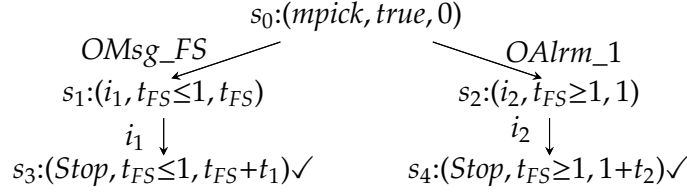


Figure A.6: LTS of composite service CS

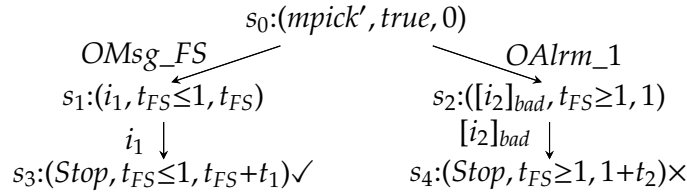


Figure A.7: LTS of composite service CS'

service CS is to response within five seconds. Figure A.6 shows the LTS of CS, where i_j denotes $sInv(S_j)$, such that S_j is a component service with parametric response time t_j , for $j \in \{1, 2\}$.

For composite service CS in Figure A.6, we have two good states (states s_3 and s_4), and the synthesized local time requirement for composite service CS is:

$$(t_{FS} \leq 1) \implies (t_{FS} + t_1 \leq 5) \wedge (t_{FS} \geq 1) \implies (1 + t_2 \leq 5)$$

A.3.2 Addressing the Bad States

Consider a variant of the example in Figure A.6, where i_2 is replaced with $[i_2]_{bad}$. That is, the composite service becomes a composite service CS' whose process component is $pick(FS \Rightarrow i_1, alrm(1) \Rightarrow [i_2]_{bad})$ (henceforth referred to as $mpick'$). This service results in the LTS shown in Figure A.7, where state s_4 is a bad state. We use this example to provide the intuition how to modify the synthesized NNCC to avoid reaching bad states. Note that the constraint $s_4.C = t_{FS} \geq 1$ is introduced by the $pick$ activity. A way to avoid the reachability of s_4 is to prevent the transition $OAlrm_1$ from firing. An effective way to achieve this is by adding the negation $\neg s_4.C$ to the synthesized NNCC. Therefore, the local time requirement for composite service CS' would be $(s_3.C \implies (s_3.D \leq T_G)) \wedge \neg s_4.C$. This NNCC can ensure the reachability of at least one of the good states and avoid the reachability of all bad states. (This will be proved in Section A.3.6.)

Algorithm 17: Algorithm *LocalTimeConstraint*(CS)

input : Composite service model CS with initial state s_0

output: The dLTC $C_L \in \mathcal{NC}_U$

1 $Cons \leftarrow synConsLTS(s_0);$

2 **return** $Cons \wedge K_{bad};$

Algorithm 18: Algorithm *synConsLTS*(s)

input : State s of LTS

output: The constraint for LTS that starts at s

1 **if** s is good state **then**

2 **return** $(s.C \implies (s.D \leq T_G));$

3 **if** s is bad state **then**

4 $K_{bad} = K_{bad} \wedge \neg(s.C);$

5 **return** *true*;

6 **if** s is non-terminal state **then**

7 $SC \leftarrow \{synConsLTS(s') | s' \in succ(s)\};$

8 **return** $\bigwedge \{C | C \in SC\};$

A.3.3 Synthesis Algorithms

Algorithm 17 presents the entry algorithm for synthesizing the dLTC for a given service CS, by traversing the $LTS = (Q, s_0, \Sigma, \delta)$ of CS.

This algorithm makes use of a second algorithm *synConsLTS*(s) shown in Algorithm 18. Given a state $s = (V, P, C, D)$ in the LTS of service CS, *synConsLTS*(s) returns a constraint $C \in C_U$. If state s is a good state (line 1), then it returns the constraint $s.C \implies (s.D \leq T_G)$ (line 2), where T_G is the given global time constraint of the service CS. $K_{bad} \in C_U$ is a static variable that is used to collect the negation of the constraint associated to states that are marked with a bad status. If state s is a bad state (line 3), then K_{bad} is conjuncted with negation of $s.C$ (line 4), and returns *true* as the constraint (line 5). The reason for returning *true* is to ensure that the returned constraint does not subsequently modify the constraint returned by *synConsLTS*(s_0), since $true \wedge C' = C'$, for any $C' \in C_U$. If s is a non-terminal state (line 6), $SC \in C_U$ is populated with the result of *synConsLTS*(s'), for each enabled state s' from non-terminal state s (line 7). Given $A = \{C_1, \dots, C_n\} \subset C_U$, we denote by $\bigwedge A$ the conjunction of constraints in A , i.e., $C_1 \wedge \dots \wedge C_n$. In line 8, the conjunction of all the elements in SC is returned as the constraint.

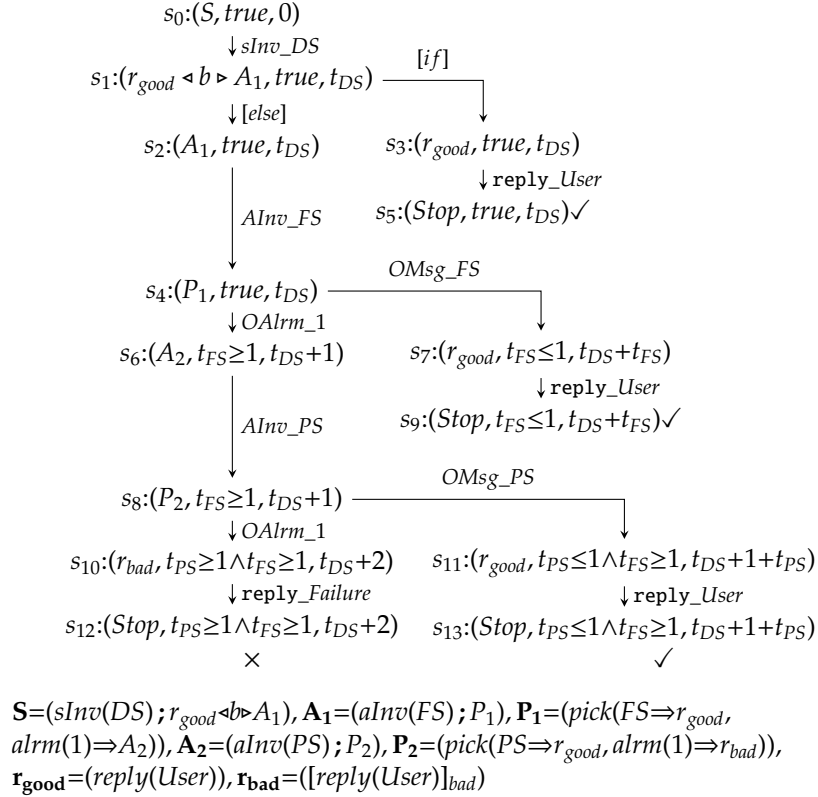


Figure A.8: LTS of the SMIS

Let us now return to the description of Algorithm 17. Upon getting the constraint of $Cons = synConsLTS(s_0)$ (line 1), the synthesized dLTC of service CS, which is $Cons \wedge K_{bad}$, is returned as the final result (line 2).

A.3.4 Application to the Running Example

Figure A.8 shows the LTS of the running example introduced in Section 3.1. Algorithm $synConsLTS(s)$ is used to synthesize the local time requirement for SMIS based on the LTS. The dLTC of the running example is shown in Figure A.9. After simplification² using Z3 [63], it becomes dLTC shown in Figure A.10.

This result provides us useful information on how the component services collectively satisfy the global time constraint. That is useful in selecting component services. For the

²For readability, we give the constraint as output in disjunctive normal form (DNF), instead of the usual conjunctive normal form (CNF).

$$\begin{aligned}
& (t_{DS} \leq 3) \wedge \\
& (t_{FS} \leq 1) \implies (t_{DS} + t_{FS} \leq 3) \wedge \\
& (t_{FS} \geq 1 \wedge t_{PS} \leq 1) \implies (t_{DS} + t_{PS} \leq 2) \wedge \\
& \neg(t_{FS} \geq 1 \wedge t_{PS} \geq 1)
\end{aligned}$$

Figure A.9: dLTC of SMIS

$$\begin{aligned}
& (t_{FS} < 1 \wedge t_{DS} + t_{FS} \leq 3) \vee \\
& (t_{FS} < 1 \wedge t_{DS} + t_{PS} \leq 2) \vee \\
& (t_{PS} < 1 \wedge t_{FS} > 1 \wedge t_{DS} + t_{PS} \leq 2) \vee \\
& (t_{DS} + t_{FS} < 3 \wedge t_{DS} + t_{PS} \leq 2)
\end{aligned}$$

Figure A.10: dLTC of SMIS after simplification

case of SMIS, one way to fulfill the global time requirement of SMIS is to select component service FS with response time that is less than 1 second, and component services DS and PS where the summation of their response times should be less than or equal to 3 seconds.

A.3.5 Service Selection

Recall that the *stipulated response time* of a component service S denotes the upper bound on its response time with respect to the synthesized constraint. dLTC could be used to select a set of services that could collectively satisfy the global time requirement of a composite service. Given a composite service CS with n component services $C = \{c_1, c_2, \dots, c_n\}$. Let $\{t_1, t_2, \dots, t_n\}$ and $\{v_1, v_2, \dots, v_n\}$, where $t_i \in \mathcal{L}_U$ and $v_i \in \mathbb{R}_{\geq 0}$, be the set of parametric response times and stipulated response times for component services in C respectively. One can check that whether the component services could collectively satisfy the dLTC of the composite service CS , by checking the satisfiability of the formula $(\bigwedge_{1 \leq i \leq n} t_i \leq v_i) \implies dLTC(CS)$.

For SMIS example, we have $(t_{FS} \leq 1.5 \wedge t_{PS} \leq 1.5 \wedge t_{DS} \leq 0.5) \implies dLTC(SMIS)$. This means that if we could select component services FS , PS , and DS that respond within 1.5 seconds, 1.5 seconds, 0.5 seconds respectively, we could always guarantee the dLTC of SMIS.

A.3.6 Termination and Soundness

We show the termination and soundness of synthesis of dLTC.

A.3.6.1 Termination

We first make the following assumption.

Assumption 1. *All loops have a bound on the number of iterations and the execution time.*

This assumption, which is necessary for termination, is reasonable in practice (see Section 6.3.5 for a discussion).

Lemma 5. *Let CS be a service model. Then \mathcal{LCS} is acyclic and finite.*

Proof: *This holds due to the assumption on the loop activities, such that the upper bound on the number of iterations and the time of execution, is known, and there are no recursive activities in BPEL.* \square

Proposition 6. *Let CS be a service model. Then $LocalTimeConstraint(CS)$ terminates.*

Proof: *From Lemma 5, \mathcal{LCS} is acyclic. Algorithm 17 is obviously acyclic too. Now, Algorithm 18 is recursive (on Line 7). However, due to the acyclic nature of \mathcal{LCS} , then no state is explored more than once. This ensures termination.* \square

A.3.6.2 Soundness

In this section, we prove Theorem 11 that will show that for any parameter valuation satisfying the output of $LocalTimeConstraint$, then no bad activity is reachable, at least one good activity is reachable, and all reachable good activities are reachable within the global delay T_G .

First, we introduce the notion of deadlockable LTS. Given a parameter valuation π , a state in a non-parametric service model $CS[\pi]$ is said to be an intermediate (resp. final) state if it is a non-terminal (resp. terminal) state in parametric service model CS . The LTS of $CS[\pi]$ is *deadlockable* if and only if there exists an intermediate state s in the LTS of $CS[\pi]$ such that $succ(s) = \emptyset$. This happens because $\pi \not\models s'.C$, for each $s' \in succ(s)$ in the LTS of parametric service model CS . We show that $CS[\pi]$ is not deadlockable in the next lemma.

Lemma 7. *Let CS be a service model, let $\pi \models LocalTimeConstraint(CS)$. Then $\mathcal{LCS}[\pi]$ is non-deadlockable. Given a service model CS , there do not exist a non-empty constraint C and a parameter valuation $\pi \models C$, such that the LTS of $CS[\pi]$ is deadlockable.*

Proof: Consider the LTS, $\mathcal{LCS} = (Q, s_0, \Sigma, \delta)$, of a composite service CS. The constraint of initial state is true, i.e., $s_0.C = \text{true}$, therefore it is always satisfiable. Given a state s , and a state s' such that $s' \in \text{succ}(s)$, the situation where $\pi \models s.C$ and $\pi \not\models s'.C$ could only happen when if $s'.C$ is stronger than $s.C$, i.e., $s'.C \subset s.C$. In such a case, the additional constraints in $s'.C$ could only be introduced by pick or flow activity using the idle function (cf. Figure A.4), for the purpose of constraining the relative speed of the services. Assume the pick construct as $\text{mpick} = \text{pick}(S \Rightarrow P, \text{alm}(a) \Rightarrow Q)$, where S is a service with parameter response time t_S , $a \in \mathbb{R}_{\geq 0}$ and P, Q are composite service activities. For the activity P to be enabled, the satisfaction of constraint $t_S \leq a$ is required, while for the activity Q to be enabled, the satisfaction of constraint $t_S \geq a$ is required. Since given any parameter valuation π , mpick will be able to execute either of the branches, therefore it cannot be deadlocked. Assume the concurrent activity as $\text{conc} = P \parallel Q$, where P, Q are composite service activities. If P (resp. Q) is a reply or asynchronous invocation activity, then P (resp. Q) is always executable, since it takes no time. If P and Q are synchronous invocation activity or receive activity, which takes parameter response time t_P and t_Q respectively, then activity P is executable, if $t_P \leq t_Q$, and activity Q is executable if $t_Q \leq t_P$. Since given any parameter valuation π , either of the branches in conc is executable, therefore it cannot be deadlocked. \square

The following lemmas will be used to prove Theorem 11.

Lemma 8. Let CS be a service model. Let $\pi \models \text{LocalTimeConstraint}(\text{CS})$. Then no bad activity is reachable in $\mathcal{LCS}[\pi]$.

Proof: K_{bad} contains the negated constraints from each of the bad states associated with all the bad activities, i.e., $K_{\text{bad}} = \{\bigwedge (\neg s_b.C_i) \mid s_b \in s_{\text{bad}}\}$. Hence, the bad activities are unreachable for any $\pi \models K_{\text{bad}}$. Now, since the result K of $\text{LocalTimeConstraint}(\text{CS})$ is $\text{Cons} \wedge K_{\text{bad}}$, then bad activities are unreachable for any $\pi \models \text{Cons} \wedge K_{\text{bad}}$, hence for any $\pi \models \text{LocalTimeConstraint}(\text{CS})$. \square

Lemma 9. Let CS be a service model. Let $\pi \models \text{LocalTimeConstraint}(\text{CS})$. Then there exists at least one reachable good state (V, P_g, C, d) in $\mathcal{LCS}[\pi] = (Q, s_0, \Sigma, \delta)$.

Proof: First, note that the initial state s_0 is reachable in $\mathcal{LCS}[\pi]$ (since $s_0.C = \text{true}$). If the initial state is the only state, then from Lemma 8, it is also not a bad state; hence it is a good state. Now, if it is not the only state, from the absence of deadlocks (Lemma 7), from the finiteness of the LTS (Lemma 5) and from the absence of bad states (Lemma 8), then at least one good state is reachable. \square

Lemma 10. Let CS be a service model. Let $\pi \models \text{LocalTimeConstraint}(\text{CS})$. Then for all good state (V, P_g, C, d) of $\mathcal{LCS}[\pi]$, $d \leq T_G$.

Proof: Let $s_g = (V, P_g, C, D)$ be a reachable state in \mathcal{LCS} such that P_g is a good activity. From Definition 18, C is satisfiable. Since P_g is a good activity, Algorithm *synConsLTS* adds a constraint $C \implies D \leq T_G$ to the result. Hence, $\text{LocalTimeConstraint}(CS) \subseteq (C \implies D \leq T_G)$. Now, for any $\pi \models \text{LocalTimeConstraint}(CS)$, we have that $\pi \models (C \implies D \leq T_G)$, and hence all reachable states in $\mathcal{LCS}[\pi]$ are such that $d \leq T_G$. \square

We can now formally state the soundness of *LocalTimeConstraint*.

Theorem 11. Let CS be a service model. Let $\pi \models \text{LocalTimeConstraint}(CS)$. Then:

1. No bad activity is reachable in $\mathcal{LCS}[\pi]$,
2. There exists at least one reachable good state (V, P_g, C, d) in $\mathcal{LCS}[\pi]$,
3. For all good state (V, P_g, C, d) of $\mathcal{LCS}[\pi]$, $d \leq T_G$.

Proof: From Lemmas 8, 9, and 10. \square

Given a composite service CS , and assume $S_g = \{s_1, \dots, s_n\}$ be the set of all good states in $\mathcal{LCS}[\pi]$ with $\pi \models \text{LocalTimeConstraint}(CS)$. In the following proposition, we show that π necessarily satisfies (at least) one of the good states' constraints, i.e., $\pi \models s_i.C$ for some $s_i \in S_g$.

Proposition 12. Let CS be a service model, and Q_{good} be the set of all good states in the $\mathcal{LCS}[\pi]$ with $\pi \models \text{LocalTimeConstraint}(CS)$. Assume $\text{LocalTimeConstraint}(CS) = (C_g \wedge K_{\text{bad}})$, where $C_g = \bigwedge_{(P_i, C_i, D_i) \in Q_{\text{good}}} (C_i \implies (D_i \leq T_G))$, and T_G be the global time constraint. Given the constraint C_g as non-empty, there does not exist a parameter valuation π such that $\pi \models C_g$ and $\pi \not\models s_g.C_i$ for all $s_g \in Q_{\text{good}}$.

Proof: Assume there exists such a parameter valuation π . According to Theorem 11, at least a good state is reachable in $\mathcal{LCS}[\pi]$. Without loss of generality, assume that a good state $(P_i, C_i, D_i) \in Q_{\text{good}}$ is reachable, which implies that $\pi \models C_i$. This contradicts the assumption. \square

A.3.7 Incompleteness of dLTC

A limitation of dLTC is that it is incomplete, i.e., it does not include all parameter valuations that could give a solution to the problem of the local time requirement. Given

an expression $A \triangleleft a = 1 \triangleright B$, since a might be unknown during design time; we explore both branches (activities A and B) for synthesizing dLTC. Nevertheless, only exactly one of the activities will be executed during runtime. Including constraints from activities A and B will make the constraints stricter than necessary; therefore some of the feasible parameter valuations are excluded – this makes the synthesis of dLTC incomplete. This can be seen as a tradeoff of making the synthesized local time requirement more general, i.e., to hold in any composite service instances. In Section 3.3, we will introduce a method that leverages on runtime information to mitigate this problem.

Appendix B

Appendix of Chapter 5

This chapter provides the evaluation of the approach introduced in Chapter 5.

B.1 Evaluation for Optimizing Selection of Competing Features

We conducted experiments to evaluate our approach. Specifically, we attempted to answer the following questions:

- RQ1.** How is the *improvement* of the solutions that found by our method compared to the existing state-of-the-art methods in terms of the correctness of the solutions?
- RQ2.** What is the *runtime* of our method compared to the existing state-of-the-art methods?
- RQ3.** Can our method be *generalized* to different EAs?
- RQ4.** How *scalable* is our method in terms of the size of feature models?

B.1.1 Setup

Algorithm	Population	Operators	Criteria for Domination	Objective of the Criteria
IBEA	Main and Archive	Crossover, Mutation, Environmental Selection	The amount of domination are calculated based on quality indicator, e.g., hypervolume.	Favors user preferences.
NSGA-II	Main	Crossover, Mutation, Tournament Selection	Distances to closest point of each objective are calculated. Favors the point with greater distance from other objectives.	Favors more spread out solutions and absolute domination.
ssNSGA-II	Main	Crossover, Mutation, Tournament Selection	Similar to NSGA, with the exception that only one new individual inserted into population at a time.	Favors more spread out solutions and absolute domination.
MOCeII	Main and Archive	Crossover, Mutation, Tournament Selection, Random Feedback	Similar to NSGA, a ranking and a crowding distance estimator is used, but bigger distance values are favored.	Favors more spread out solutions and absolute domination.

Table B.1: Brief overview of EAs

B.1.1.1 Implementation

We have implemented our approach based on jMetal [71], which is a Java-based open source framework that supports multi-objective optimization with EAs. Sayyad *et.al* [138, 136] have made an extensive experiments to test how different EAs implemented on jMetal could contribute to the optimal feature selection. We use the EAs that are reported to work well in their experiments, and evaluate how the preprocessing and feedback-directed mechanisms affect these EAs. The EAs we are using for the evaluation are:

1. **IBEA:** Indicator-Based Evolutionary Algorithm [172]
2. **NSGA-II:** Nondominated Sorting Genetic Algorithm [64]
3. **ssNSGA-II:** Steady-state NSGA-II [72]
4. **MOCeII:** A Cellular Genetic Algorithm for Multi-objective Optimization [126]

A brief overview of these EAs are provided in Table B.1.

Repo.	Model	Fea.	Cons.	F_p	F'_p
–	JCS	12	13	2	–
SPLOT	Web Portal	43	36	4	–
	E-Shop	290	186	28	–
LVAT	eCos	1244	3146	54	19
	FreeBSD	1396	62183	41	3
	uClinux	1850	2468	1244	1244
	Linux X86	6888	343944	156	94

Table B.2: Feature Models

B.1.1.2 Quality Indicators

To measure the quality of Pareto front, we make use of two indicators in this chapter: hypervolume [173] and spread [64].

- a) **Hypervolume (HV):** Hypervolume of the solution set $S = (x_1, \dots, x_n)$ is the volume of the region that is dominated by S in the objective space. In jMetal, although all objectives are minimized, but the Pareto front is inverted before the hypervolume is calculated. Therefore, the preferred Pareto front would be with the most hypervolume.
- b) **Percentage of Correctness (%Correct):** There might be solutions that violate some constraints in the Pareto front, since the correctness is an optimization objective that evolves over time. Solutions that are correct (i.e., without violating any constraint) are more useful to the user; therefore we are interested in the percentage of solutions that are correct in the Pareto front.

B.1.1.3 Feature Models and Attributes

The details of feature models used in the experiment are summarized in Table B.2, with the repository information (*Repo.*), number of features (*Fea.*), number of constraints (*Cons.*), number of prunable features with the preprocessing method in Algorithm 6 (F_p), number of prunable features with the preprocessing method in [137] (F'_p), and literatures (*Ref.*) associated with each feature model.

JCS feature model is the feature model that we have used throughout the paper. Two feature models *Web Portal* and *E-Shop* are from SPLOT repository [122], which is a repository

used by many researchers as a benchmark. The *Web Portal* [121] model captures the configurations of Web portal product line, and the *E-Shop* [164] model, which is one of the largest feature models in SPLOT, captures a B2C system with fixed priced products. These two models are chosen to facilitate the comparison with [138]. To further evaluate the scalability of our methods, we make use of feature models from the Linux Variability Analysis Tools (LVAT) feature model repository [5]. The models in LVAT were reversed-engineered by making use of source code, comments and documentations of big projects such as *eCos* [140, 164] operating system, *FreeBSD* [140, 42], *uClinux* [41] and *Linux X86* [140, 42] operating system. Compared to the feature models in SPLOT, the feature models in LVAT contain a significant larger number of features and constraints, and have higher branching factors, but they have lower ratios of feature groups, and hence shallower tree structures in general.

Note that F_p always contains more features than F'_p – this shows that our preprocessing method with Algorithm 6 has found more prunable features than [137]. In [137], their preprocessing method is based on static analysis. In particular, they detect disjunctions (rules) with only one feature, which means the feature is either a common feature or a dead feature. In addition, they investigate the disjunctions (rules) that include two features, if one of them is prunable in the first round, and the other one could be prunable as well. It is easy to see that our method based on SAT solving could detect all features that could be found by preprocessing method in [137], and it can be shown that F_p is always not lesser than F'_p .

B.1.1.4 Feature Attribute

Each feature in the feature models has the following attributes, which are the same as the attributes used in [138]:

1. **Cost** $\in \mathbb{R}$, records the number of cost incurred to use the feature. For each feature, the *Cost* value is assigned with a real number that is normally distributed between 5.0 and 15.0.
2. **Used_Before** $\in \{true, false\}$, indicates whether this feature was used before. The value of *Used_Before* is *true* if the feature has been used before, otherwise it is *false*. For each feature, the *Used_Before* value is assigned with a Boolean value that is distributed uniformly.

3. *Defects* $\in \mathbb{Z}$, records the number of defects known in the feature. For each feature, the *Defects* value is assigned with an integer number that is normally distributed between 0 and 10. However, if the feature has not been used before, the *Defects* value is set to 0.

B.1.1.5 Optimization Objectives

We introduce the five optimization objectives that we use in the experiment in the following. Note that since jMetal requires minimization of the objectives; all objectives listed here are objectives to be minimized.

Obj1. *Correctness*: minimize the number of violated constraints of the feature model.

Obj2. *Richness of features*: minimize the number of features that are not selected.

Obj3. *Cost*: minimize the total cost.

Obj4. *Feature used before*: minimize the number of features that have not been used before.

Obj5. *Defects*: minimize the number of known defects.

We specify correctness as an objective, rather than a constraint. The reason is that this allows EA to nudge the search towards feature models that contain lesser violated constraints, which eventually lead to valid feature models that do not contain violated constraints. Furthermore, note that some objectives are conflicting, e.g., Obj2 and Obj3, because the richness of features would imply a higher cost, but at the same time the cost needs to be minimized.

B.1.1.6 Configurations of EAs

Given an EA, we introduce the configurations for comparison.

1. **F+P**: This is the EA that makes use of feedback-directed crossover and mutation (Section 5.2.4) and preprocessing (Section 5.2.2) is applied before the execution of the feedback-directed EA.
2. **U+P**: The unguided version of EA with preprocessing (Section 5.2.2) applied before the execution of the unguided EA. We have demonstrated that, our method has found

Model		IBEA			NSGAI			ssNSGAI			MOCe		
		F+P	U+P	U	F+P	U+P	U	F+P	U+P	U	F+P	U+P	U
E-shop	Time (ms)	6994	6369	7401	1906	2150	2548	15214	16863	17541	2822	3964	4463
	HV	0.3	0.18	0.19	0.26	0.2	0.17	0.24	0.22	0.22	0.24	0.19	0.22
	%Correct	100.0	0.0	0.0	12.0	0.0	0.0	15.0	0.0	0.0	14	0.0	0.0
Web Portal	Time (ms)	5678	4596	4646	433	483	546	8309	8315	8221	1033	1793	1857
	HV	0.32	0.2	0.23	0.3	0.24	0.21	0.3	0.21	0.24	0.31	0.21	0.22
	%Correct	100.0	1.0	0.0	28.0	0.0	0.0	20.0	1.0	0.0	41	0.0	0.0
JCS	Time (ms)	4681	4318	4735	271	269	301	7289	6834	6890	271	432	595
	HV	0.31	0.3	0.28	0.3	0.29	0.28	0.29	0.29	0.29	0.33	0.31	0.3
	%Correct	96.0	78.0	54.0	27.0	22.0	16.0	31.0	24.0	14.0	34.0	21.0	18.0

Table B.3: Evaluation with SPLOT

more prunable features than the preprocessing method of [137] in Section B.1.1.3; therefore, U+P can be seen as an improved version of [137] with smaller search space.

3. U: The unguided version of EA without preprocessing, which is used by [138, 136].

B.1.1.7 Parameter Settings

For U , the same as [136], single-point crossover and bit-flip mutation are used as crossover and mutation operators, with crossover and mutation probabilities set to 0.1 and 0.01 respectively. These operators and probabilities also apply to $U + P$. For $F + P$, the feedback-directed crossover (Algorithm 12) and feedback-directed mutation (Algorithm 7) operators are used. The error mutation probability P_{emut} , mutation probability P_{mut} , and crossover probability P_{cross} are set to 1.0, 0.0000001, and 0.1 respectively. All other parameter settings for each EA are default settings of jMetal (e.g., population size is set to 100), and therefore are omitted here.

For SPLOT case study, we make use of 25000 evaluations using four EAs (IBEA, NSGAI, ssNSGAI, and MoCell). For the larger LVAT case study, we make use of 1000000 evaluations using IBEA. For both case studies, we generate 10 sets of attributes. For each set of attributes, we run each EA repeatedly for 30 times, and report the medium values of the metrics. The evaluation results for SPLOT and LVAT are reported in Table B.3 and Table B.4 respectively.

We make use of Mann Whitney U-test [31] to test the statistical significant of %Correct indicator. We highlight the %Correct in **bold** for $F + P$, if the confidence level exceeds 95% when comparing $F + P$ and $U + P$.

Model		IBEA		
		F+P	F'+P	U+P
eCos	Time (ms)	33245	51279	58561
	HV	0.25	0.21	0.18
	%Correct	100	61.45	0.0
	E50	6300	62400	–
FreeBSD	Time (ms)	64042	82087	85750
	HV	0.33	0.33	0.35
	%Correct	100	100	0.0
	E50	1500	1600	–
uClinux	Time (ms)	50668	43876	46986
	HV	0.31	0.29	0.28
	%Correct	100	100	0.0
	E50	600	2100	–
Linux X86	Time (ms)	32758	31396	37472
	HV	0.2	0.2	0.22
	%Correct	0.0	0.0	0.0
	E50	–	–	–

Table B.4: Evaluation with LVAT

The experiments were conducted on an Intel Core I7 4600U CPU with 8 GB RAM, running on Windows 7.

B.1.2 Evaluation with SPLOT

Table B.3 demonstrates our results with SPLOT case study, where *Time(ms)*, *HV*, and *%Correct* represent execution time in milliseconds, hypervolume and percentage of correct solutions in the Pareto front.

RQ1: We notice that the IBEA has outperformed other methods on the percentage of correctness. This is conformed to the observation in [138]. According to [138], this is because all EAs used in this case study (other than IBEA) use diversity-based selection criteria, which favor higher distances between solutions. For this reason, non-IBEA methods tend to remove solutions that crowded towards the zero-violation point, thus achieving lower scores on the percentage of correctness measure.

We also notice that for each EA, the configuration *U + P* outperforms the configuration *U* on the percentage of correctness. This is because the preprocessing method has filtered away

	Time (ms)	HV	%Correct
IBEA	-690	0.08	72.33%
NSGA-II	97.33	0.05	15%
ssNSGA-II	400	0.04	13.67%
MOCeII	687.67	0.06	22.67%

Table B.5: Improvement of EAs on SPLOT

	Time (ms)	HV	%Correct
IBEA	9718	0.015	75%

Table B.6: Improvement of EAs on LVAT

the prunable features, which makes the search space smaller. Hence EAs are more effective in the optimal feature selection. We also observe that $F + P$ outperforms $U + P$ constantly on the percentage of correctness. This is attributed to the feedback-directed crossover and mutation, which have effectively guided EAs to explore more promising region of the solution space for locating the optimal feature selection. The average improvement for the configuration $F + P$ over $U + P$ is summarized in Table B.5, where the values are calculated by summing up the differences of *%Correct* between $F + P$ and U for all tested EAs, and divided by four (the number of tested EAs). Positive values mean improvements, while negative value mean the opposite. This has shown that our methods have provided an improvement on the percentage of correctness for all case studies using different EAs, especially in IBEA which has 72.33% improvement of correctness. These results answer research question RQ1.

RQ2: The runtime of configurations $F + P$, $U + P$, and U are comparable. There does not exist a configuration that has a clear advantage over the others in terms of the runtime. The reason is that all configurations go through the same number of evaluations. One might think that the configuration $F + P$ requires an extra calculation of the error position using Algorithm 8. In fact, the constraints also need to be enumerated for configurations $U + P$ and U during each round of evolution, in order to calculate the number of violated constraints. Therefore, the extra operation of $F + P$ is only the *getFeatures* function that is used in line 10 of Algorithm 8, which has a low complexity. On the other hand, $F + P$ and $U + P$ have shorter chromosome than U due to the preprocessing. However, these does not reflect much on the results, because the selection, mutation, and crossover for chromosomes could be done efficiently. These results answer research question RQ2.

RQ3: To answer research question RQ3, we notice that the percentage of correctness of

all tested EAs (IBEA, NSGA-II, ssNSGA-II and MOCell) have been improved by using $F + P$. These results convey to us that, the preprocessing method and feedback-directed crossover and mutation have provided an advantage on the percentage of correctness and HV, regardless of the underlying EAs. The reason is that the preprocessing method effectively prunes the search space, and the feedback-directed crossover and mutation allow underlying EAs to use the feedback for faster finding of valid solutions. This also shows that the preprocessing method and feedback-directed crossover and mutation are general methods that could be applied for different EAs.

RQ4: To answer the research question RQ4, we make use of the E-shop model. E-shop model contains one of the largest set of features in the SPLOT repository [122]. The results show that, with $U + P$ and U , none of the EAs could locate a correct solution. On the other hand, with $F + P$, IBEA has achieved 100% of correctness, while NSGA-II, ssNSGA-II and MOCell have achieved 12–14% of correctness. We have also further evaluated for 50 millions rounds of evolution for $U + P$ for IBEA. It has only achieved 46% of correctness after 50 millions rounds which takes 3.25 hours. In contrast, the configuration $F + P$ has achieved 100% of correctness by just 6.9 seconds.

To confirm the scalability of feedback-directed IBEA, we conduct the evaluation using LVAT in the next section.

B.1.3 Evaluation with LVAT

Table B.4 demonstrates our results with LVAT case study with IBEA, where $E50$ represents the number of executions required to obtain 50% of correct solutions in the Pareto front. Configuration $F' + P$ is the same as $F + P$, with the exception that the mutation probability P_{mut} is set to 0.01 (for $F + P$, $P_{mut} = 0.0000001$). The average improvement for the configuration $F + P$ over $U + P$ is summarized in Table B.6. We notice that for *eCos*, *freeBSD*, and *uCLinux*, $F + P$ achieves 100% correctness for all cases, while $U + P$ does not find any correct solution after 100000 executions. Although $F + P$ achieves overall better runtime, it does not has clear advantage over $U + P$ for all models. These results have confirmed for the better percentage of correctness (RQ1) and comparable runtime (RQ2) of $F + P$ over $U + P$. For *Linux X86* which contains 6888 features, none of the methods ($F + P$, $U + P$, and U) have found a correct solution. Therefore, we resort to the “seeding method” proposed by [137].

In [137], the authors make use of two methods, i.e., SMT solver and IBEA of two objectives, for finding a correct solution (the “seed”), and plant the seed in the initial population of

IBEA with the hope to find more valid solutions. We run two seeding methods proposed by [137] with $F + P$, and compare the results with the improved version of method proposed by [137], i.e., $U + P$.

First, Microsoft Z3 SMT solver [63] is used to find a seed. In our case, Z3 successfully finds a valid solution in around three seconds (we repeat for 30 times, and medium of the number of selected features is 1455). With the seed, $F + P$ successfully find 34 correct solutions using no longer than 30 seconds. In contrast, $U + P$ does not find any new solution after 30 minutes.

Second, IBEA with two objectives is used to generate the seed. In our case, $F + P$ uses less than 40 seconds to get 36 correct solutions. While for $U + P$, it spends a total of 3.5 hours of execution time for 30 correct solutions and 4 hours of execution time for 36 correct solutions. $F + P$ has shorten the search time of $U + P$ for more than 200 times. In particular, $U + P$ spends 3 hours to generate the seed, and spends half an hour to obtain 30 correct solutions. And given another half an hour, $U + P$ finally obtains 36 correct solutions.

These results have shown that $F + P$ outperformed $U + P$ given both seeding methods in [137]. Note that the seed generated by IBEA with two objectives, is better than the seed generated by the Z3 SMT solver. Both $F + P$ and $U + P$ find more solutions using seed generated from IBEA with two objectives. This is conformed to the observation in [137]. According to [137], it is because the seed generated by IBEA with two objectives has more selected features, and the "feature-rich" seed allows the effective search of other valid solutions.

We also compare how the mutation parameter P_{mut} affects feedback-directed IBEA. Out of five models, $F' + P$ only performs poorer than $F + P$ in *eCos*. To better observe the effect, we make us of *E50*. It shows that $F + P$ obtained 50% of correct solutions in Pareto front in a smaller number of evaluations for all models, except *LinuxX86*. The results show that smaller P_{mut} leads to faster convergence of correct solutions in Pareto front. This is because smaller P_{mut} minimizes the modification of non-error positions; therefore, it allows IBEA to focus more on the correction of constraint violations.

B.1.4 Threats to validity

There are several threats to validity. The first threat of validity is due to the fact that values for the feature attributes (i.e., *Cost*, *Defects*, and *Used_Before*) were randomly generated. This

is due to difficulty in obtaining the attributes that are associated with real-world products since many of them are proprietary. To mitigate the effect of randomness, we generate 10 set of attributes for each case study. Furthermore, for each set of attributes, we run each EA repeatedly for 30 times, and report the medium values of the metrics. Future work should involve the use of real data for the evaluation.

The second threat of validity stems from our choice of using an exemplar parameter set (e.g., for crossover and mutation probability), which comes with the default setting of jMetal, in order to cope with the combinatorial explosion of options. To address these threats, it is clear that more experimentations with different feature models and experimental parameters are required, so that we could investigate effects that have not been made explicit by our dataset and experimental parameters.