Seema Jehan

Model-Based Testing and Debugging of SOA Business Processes

Dissertation

Graz University of Technology

Institute for Software technology

Supervisor: Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa
Evaluator: Prof. Dr. Inmaculada Medina-Bulo

Graz, April 2017

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

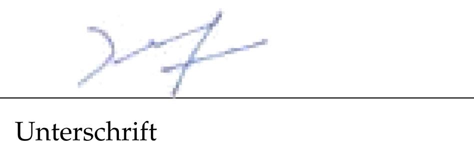
I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, 01/08/2017

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am 01/08/2017 

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Testing of SOA-based processes still face issues of limited controllability and observability. There are number of solutions suggested to tackle this problem. However, most of these approaches suffer from high test case generation and execution costs. The target here is to come up with a cost-effective functional testing approach for SOA workflows. In this respect two model-based test case generation algorithms for sequential and concurrent BPEL processes have been proposed. These algorithms are based on constraint representation of BPEL control flow graph, where one is structure-based and other follows random test case generation approach. Moreover, this thesis also analyzes three test suite reduction algorithms. The aim here is to reduce regression testing cost, which keeps on increasing as business processes evolve. In the end, we present a light-weight model-based debugging approach for locating functional faults in SOA business processes.

Abstract (German)

Das Testen von SOA-basierten Prozessen steht nach wie vor mit Fragen der eingeschränkten Steuerbarkeit und Beobachtbarkeit. Es gibt eine Reihe von Lösungen, die vorgeschlagen werden, um dieses Problem anzugehen. Die meisten dieser Ansätze leiden jedoch unter hohen Testfall- und Ausführungskosten. Ziel ist es, mit einem kostengünstigen Funktionstest-Ansatz für SOA-Workflows zu kommen. In dieser Hinsicht wurden zwei modellbasierte Testfall-Erzeugungsalgorithmen für sequentielle und gleichzeitige BPEL-Prozesse vorgeschlagen. Diese Algorithmen basieren auf der Einschränkungsdarstellung des BPEL-Kontrollflussgraphen, wobei eine strukturbasierte und andere der zufälligen Testfallgenerierungsansatz folgt. Darüber hinaus analysiert diese Arbeit auch drei Test-Suite Reduktionsalgorithmen. Ziel ist es, die Regressions-Testkosten zu senken, die sich weiter steigern, wenn sich die Geschäftsprozesse entwickeln. Am Ende präsentieren wir einen leichten modellbasierten Debugging-Ansatz zur Lokalisierung von Funktionsstörungen in SOA-Geschäftsprozessen.

Acknowledgements

First and foremost, I am deeply indebted to ALLAH (the exceedingly merciful) for all blessings in my life. My entire family also deserves my sincere gratitude for their love, support and encouragement.

I am extremely grateful to my advisor, Franz Wotawa, for his perpetual support through out Ph.D. studies. His motivation and guidance helped me through many difficult times in my research and writing of this thesis. Besides my advisor, I would like to thank the members of my defense committee, Prof. Inmaculada Medina-Bulo and Prof. Denis Helic, for reviewing my thesis and to supervise the exam.

Further thanks go to Ingo Pill and Birgit Hofer for their professional consultation. I also wish to thank my colleagues Iulia Nica and Josip Bozic for their friendly advice. I must thank Petra Pichler for always being welcoming and courteous.

I would also like to thank Austrian Science Fund (FWF) for funding part of the project Augmented Diagnosis and Testing for SOAs (Audit 4 SOAs) under grant P23313-N23.

Contents

Abstract (German) vii

Abstract v

List of Tables xv

List of Figures xvii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Thesis Statement . . . 4
- 1.4 Contributions . . . 4
- 1.5 Organization . . . 5

2 Service-Oriented Architectures 7
- 2.1 Introduction . . . 7
- 2.2 A brief history of SOAs . . . 8
  - 2.2.1 Early SOA . . . 9
  - 2.2.2 Second generation of SOA . . . 10
  - 2.2.3 Contemporary SOA . . . 11
- 2.3 SOA basics . . . 11
  - 2.3.1 First generation Web services technology . . . 12
  - 2.3.2 Second generation Web services technology . . . 19
- 2.4 SOA challenges . . . 29

3 Preliminaries and Related Work 31
- 3.1 Testing Preliminaries . . . 32

3.2 Testing trends of SOA Applications 36
3.2.1 Symbolic Execution Approach 37
3.2.2 Model Checking Approach 39
3.2.3 Petri Net Approach 40
3.2.4 Graph-based Approach 40
3.2.5 Contract-based Approach 42
3.2.6 Search-based Approach 42
3.3 Debugging Definitions 44
3.4 Diagnosis of SOA Applications 46
3.5 Conclusions 48

4 Model-based SOA Testing 49
4.1 Introduction 49
4.2 Architecture 51
4.3 Definitions 53
4.4 Test Case Generation using constraints 57
4.4.1 Sequence Structure 58
4.4.2 Flow Structure 69
4.5 Experimental Setup 76
4.6 First Results 78
4.7 Random Testing of Sequential programs 80
4.7.1 Introduction 82
4.7.2 Experimental Results 85
4.7.3 Conclusions 90
4.8 Random Testing of Concurrent programs 90
4.8.1 Introduction 90
4.8.2 Empirical Evaluation 92
4.8.3 Discussion 97
4.9 Conclusions 98

5 Test Suite Reduction 101
5.1 Introduction 101
5.2 Related research 102
5.3 Preliminaries 105
5.4 Redundancy elimination 107
5.4.1 LinMIN Algorithm 107
5.4.2 BinarySearch Algorithm 109

5.4.3 Delta-Debugging Algorithm 110
5.5 Empirical Evaluation 110
5.6 Conclusions 116

6 Model-Based SOA Debugging 121
6.1 Introduction 121
6.2 Definitions 123
6.3 The Debugging Approach 126
6.4 Experiments 133
6.5 Conclusions 138

7 Conclusions 141
7.1 Results summary 141
7.2 Open Questions and Future Work 144

List of Definitions 145

List of Theorems and Lemmas 147

Bibliography 149

List of Tables

4.1 Examples Details 79
4.2 Empirical results obtained 81
4.3 Experimental results for the AllPath TCG algorithm. 87
4.4 Experimental results for the random TCG algorithm. 89
4.5 Experimental results for the StructRuns TCG algorithm. . . 94
4.6 Experimental results for the RandomRuns TCG algorithm with len = 40. 95
4.7 Infeasible paths for the RandomRuns TCG algorithm. 97

5.1 Experimental results for Redundancy Reduction algorithm. . 113

6.1 Single Faults Diagnoses. 136
6.2 Results for programs with single and double faults. 140

List of Figures

2.1 Twitter Service 8
2.2 An early SOA Model 10
2.3 Contemporary SOA Model taken from [Leitner et al., 2013] . 12
2.4 Web Service roles 14
2.5 A short example of an XML Schema. 16
2.6 A SOAP message. 17
2.7 WSDL components taken from [WSDL, 2001]. 18
2.8 An Executable BPEL process 22
2.9 BPEL process definition 24
2.10 Receive Activity 25
2.11 Reply Activity 26
2.12 InvokeActivity 27
2.13 AssignActivity 28
2.14 IfActivity 28
2.15 WhileActivity 28

4.1 The BPEL Example 50
4.2 The BPELTesterFigure 52
4.3 A Sequence activity 58

4.4 TCG algorithm that considers all paths. 59
4.5 AllPathsSUB algorithm for computing all paths for a Flow Graph G up to a given pre-defined length MaxLen. 60
4.6 Path 1: $c(\pi)$ for low-risk and low amount loan requests. . . . 64
4.7 Path 2: $c(\pi)$ for high-risk and low amount loan requests. . . . 64
4.8 Path 3: $c(\pi)$ for high amount loan requests. 64
4.9 MINION constraints for Path 1. 65
4.10 MINION constraints for Path 2. 65
4.11 MINION constraints for Path 3. 65
4.12 Technical view of the Bank Loan Business Process. 66
4.13 Flow activity 69
4.14 Flow Example 70
4.15 Our structural TCG algorithm StructRuns 73
4.16 Run 1: $c(\pi)$ for the first guard to be active. 74
4.17 Run 2: $c(\pi)$ for the second guard to be active. 74
4.18 MINION constraints for Run 1. 75
4.19 MINION constraints for Run 2. 75
4.20 Flow Example Test Case 79
4.21 Coverage vs. path length for the ATM example. 81
4.22 TCG algorithm based on random paths. 84
4.23 AllPaths TCG alg: activity coverage and mutation score vs. path length 88
4.24 TCG algorithm RandomRuns based on random paths. 92
4.25 RandomRuns: Mutation score as function of the number of test cases for *BMI* 97

5.1 The minimum, maximum, and average mutation score for the *CALC2* example with varying subset size 108

5.2 The probability for a subset of the original test suite of *CALC2* to have a mutation score larger than 85.0 108

5.3 LINMIN - A linear search procedure for test suite minimization 111

5.4 BinSearch – A Binary search procedure for test suite minimization 112

5.5 DELTAMIN – Using delta debugging for test suite minimization 112

5.6 The test suite reduction for the *CALC2* example with varying alpha 114

5.7 The minimum, maximum, and average mutation score for the *CALC2* example with solSize=3 and varying subset size 117

5.8 The probability for a subset of the original test suite of *CALC2* with solSize=3 to have a mutation score larger than 85.0 . . . 117

5.9 AllRandomPaths – Using random subsets for test suite generation 118

5.10 The minimum, maximum, and average mutation score for the *CALC2* example with solSize=5 and varying subset size 119

5.11 The probability for a subset of the original test suite of *CALC2* with solSize=5 to have a mutation score larger than 85.0 . . . 119

5.12 The test suite reduction for the *CALC2* example for multiple tests per feasible path 120

6.1 BPEL Flow Graph of the Triangle Example Process 122

6.2 Annotated Flow Graph Representation 128

6.3 A graphical presentation of the running example's single-fault diagnoses. 132

6.4 BPEL Flow Graph of the Bank Loan Business Process 134

6.5 Comparison of the trace size and diagnoses size for faulty programs. . . . 137

1 Introduction

1.1 Motivation

Service-oriented Architectures (SOAs) have changed today's computing in an evolutionary manner. The popularity of the SOA paradigm is noteworthy by the exponential growth of social networking companies like Twitter, Facebook and Amazon cloud web services [Cloud, 2013]. According to a study from [Corporation(IDC), 2015], the world wide Big Data technology and services market growth is expected to reach $48.6 billion in year 2019. The peculiar features of SOA-based systems such as ultra-late binding, Quality of Service (QoS) aware composition, runtime discovery of services, and service level agreement (SLA) automated negotiations are considered the driving force for this emerging paradigm [Canfora and Penta, 2009a]. The wide acceptance of the service-oriented paradigm also depends on the fact that the traditional concept of software ownership is rapidly shifting to software provision, where a software is charged as a service rather than as a product. This shift provides on one hand greater flexibility in software use; but also raises many challenges for assuring reliability of software charged as a service.

The predominant way of assuring reliability of service-based software has been monitoring these services and applying repair actions on runtime [Console et al., 2007], [Friedrich et al., 2010b]. Although monitoring helps in building self-healing and self-repairing systems, it is unable to give confidence that a system would work before its deployment. Besides that, it also requires to maintain recovery actions for all possible exceptional events, which is realistically not possible [Canfora and Penta, 2009a]. This calls for cost-effective testing strategies for not only minimizing exceptional events

set; but also maintaining the same quality assurance level. There are well-established testing strategies developed for web-based systems, distributed systems, and component-based systems [Myers, 1979]. Unfortunately, these traditional testing techniques are not able to cope with the dynamic and intrinsic nature of SOA-based systems [Canfora and Penta, 2006].

There are many challenges associated with testing applications built on SOA principles: First, the services business logic is hidden, making it a "black-box" for the tester; and due to this limited observability, generating models from service descriptions is cumbersome and inefficient [Canfora and Penta, 2009a]. Second predominant issue is the limited control over the service; services change independently from each other, making integration testing harder and laborious. Third, the cost of testing SOA-based systems is much higher than the traditional software systems because services are charged on per-use basis. In addition to that, exhaustive testing might lead to denial-of-service system behavior [Canfora and Penta, 2009a]. As a result testing is ignored all together in many situations, making such applications more vulnerable to cyber attacks [Lowis and Accorsi, 2011]. This requires us to adapt current testing methodologies in order to suit the need of SOA-based systems.

Although this problem has been investigated before, but a vast majority of presented testing approaches failed to provide an experimental evaluation [Bozkurt et al., 2013]. According to [Canfora and Penta, 2009a], high cost involved in testing is a prime reason, why runtime-verification has become a norm in SOA-based applications. This thesis presents a cost efficient model-based functional testing of SOA applications with a focus on an empirical evaluation.

Once a fault, either functional or non-functional, is observed in the testing phase, the fault-localization and repair of SOA-based environments has also been an active research area in academia. The complete scenario for diagnosing SOA environments involves many stages: diagnosing faults, repairing faults and recovery stage [Friedrich et al., 2010a]. However, the cost of generating diagnostic models is quite high [Nica, 2010]. It becomes even more challenging when it comes to the service-oriented architectures as discussed by Friedrich et al. in [Mayer et al., 2012]. The most recent work on that was conducted under a european project called

WS-DIAMOND [Diamond, 2010]. The main focus of the project was to build a platform for monitoring, diagnosis and self-healing of web services [Modafferi et al., 2006]. In contrast to their work, our diagnosis approach combines trace analysis with constraint solving for fault localization in order to improve overall diagnosis performance.

The work can be of interest to different stakeholders involved in SOA such as developer, who can use models to generate unit tests with limited cost; service provider, who can generate tests from service specifications; integrator, who can use the approach to reduce the high testing cost.

The work presented in this thesis is a part of the project called Augmented Diagnosis and Testing of SOAs (Audit4SOAs) funded by the Austrian Science Fund (FWF). It was a collaboration project between Institute of Software Technology TU Graz and Distributed Systems Group at Vienna University of Technology. The target of the project was to provide model-based techniques for testing and debugging of service-oriented architectures. The objective of the thesis was to develop a testing and diagnosis strategy of SOA processes defined in Business process execution language (BPEL).

1.2 Problem Statement

The first problem area examined in this thesis is about observability and controllability for SOA business processes. These are serious issues, because SOA applications need to ensure a certain level of "Trust", and testing is one way of resolving this issue [Bozkurt et al., 2013]. The focus was on the development of an automated model-based testing solution for BPEL compositions with an emphasis on a strong empirical analysis. How can partial behavior of SOA business processes be addressed in the test suite generation process?

The second issue explored in this work is about optimizing the automated test suite generation. Since the prevalent SOA tools need high execution times due to the complex and distributive nature of business compositions, it is important to generate efficient test suites before hand. The goal is to

reduce the generated test suite size, while maintaining the same quality criterion. How can test suites be optimized?

The third issue studied is related to model-based debugging of BPEL compositions. Basically, the purpose of any testing activity is to find faults in a given system, once this is done, the diagnosis step takes over, and is responsible for figuring out possible reasons behind the observed fault. We had two targets in the context of debugging service compositions: first, the problem of debugging partial behavioral models is studied; second, a light-weight model-based debugging approach for diagnosing BPEL functional faults is presented. How can partial behavioral models be diagnosed in a cost-effective way?

1.3 Thesis Statement

The applicability of functional testing and debugging of service compositions can be increased using constraint-based approach, with the help of an optimized test suite generation and diagnostic methods.

1.4 Contributions

The contribution has been three-folds. First, we studied the problem of extracting models from the partial service behavior of BPEL compositions, with a focus on a strong empirical analysis. Second, we presented a light-weight model-based debugging approach for finding functional faults in BPEL compositions, again with a particular focus on an empirical analysis. In the end, we analyzed the issue of test suite redundancy, a reason for high cost involved in testing; and presented our solution to the problem.

The emphasis of our research was to present a light-weight testing and debugging approach for SOA applications, which can be easily adapted in the industry. Below is the list of conference and workshop publications to tackle the above stated research questions:

- An overview of different issues regarding testing of SOA applications is published in the paper titled, "Fifty shades of grey in SOA testing" [Wotawa et al., 2013].
- A constraint-based method of formal representation about BPEL compositions is presented in, "SOA grey box testing- a constraint-based approach" [Jehan et al., 2013b].
- A detailed description of the model generation using a typical SOA case study is published in the paper titled, "Functional SOA testing based on constraints" [Jehan et al., 2013a].
- The question of whether a random testing approach is better than the structured testing approach is examined in the paper, "SOA testing via random paths in BPEL models " [Jehan et al., 2014].
- The formal representation of both sequential and concurrent constructs in BPEL composition with a focus on the empirical analysis is published in the paper titled, "BPEL Integration Testing" [Jehan et al., 2015].
- The fault-localization of functional faults using a light-weight debugging approach is presented in the paper, "Functional Diagnosis of SOA BPEL Processes" [Hofer et al., 2014].
- A more generalized analysis of the debugging approach is published in the paper titled, "Focussed Diagnosis for Failing Software Tests" [Hofer et al., 2015].
- A preliminary work on a problem of reducing the redundancy in test suites is published in the paper titled, "Analyzing the reduction of test suite redundancy" [Pill et al., 2015].

1.5 Organization

The structure of the thesis is as follows: In Chapter 2, an introduction to the Service-oriented Architectures (SOAs) is presented. Chapter 3 presents an overview of related research in the field of testing and diagnosis of SOAs in general and Business Process Execution Language (BPEL) in particular. The contribution of this thesis with respect to the model-based testing approach is explained in Chapter 4. The issue of test suite redundancy and the analysis of different algorithms for cost-effective testing is discussed in Chapter 5. In the end, a light-weight debugging approach for diagnosing

BPEL functional faults is presented in Chapter 6. The summary of results obtained and open questions left in the discourse of this work are outlined in Chapter 7.

2 Service-Oriented Architectures

2.1 Introduction

"Ufone" is one of the largest GSM (Global System for Mobile communication) mobile service provider with around 24 million customers in Pakistan [Ufone SOA Integration, 2012]. It has a network coverage across 10,000 locations. Also, its services are available in more than 160 countries world wide. In order to provide their customers real-time service and reduce the churn rate[1], the company decided for moving the critical operational systems such as customer relationship management (CRM), network provisioning and billing towards SOA. With the successful implementation of SOA, Ufone can now handle around 1.5 million transactions per day.

Another successful realization of SOAs can be observed by an online social networking service, "Twitter" [Dorsey, 2006]. This social networking service facilitates 302 million users (May 2015, wikipedia) world-wide to send and receive short text messages in real time. The company started in year 2006, and has grown ten times between year 2010 and year 2013, with a record number of 400,000,000 tweets per day. This exponential growth in the number of users brought various challenges, such as poor concurrency and latency. Another major issue was that different parts of the service were developed in different languages such as Java [Gosling, 1995], Ruby [Matsumoto, 1995], Scala [Odersky, 2004] and Javascript [Eich, 1995]. These challenges were mainly because of the tight coupling between various components of the organization. Twitter testing team lead Jeremy Cloud discussed in his talk [Cloud, 2013], how SOA helped them to cope with the aforementioned challenges of scalability and concurrency. As a result, the Twitter service, which previously could handle just twenty tweets per second

[1] the annual percentage rate at which customers stop subscribing to a service.

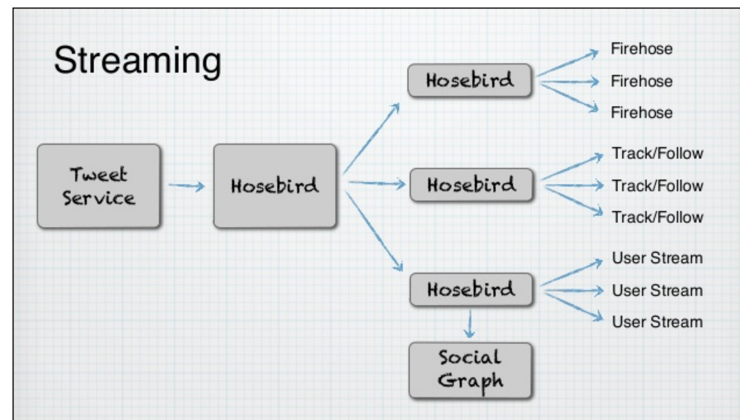


Figure 2.1: Twitter Service Decomposition

(TPS) and 400 queries per second (QPS) in year 2006, could manage 1,000 TPS and 12,000 QPS by year 2010 [Humble, 2011]. Figure 2.1 displays one of the core APIs based on SOA principles, used in the Twitter application, for providing real-time tweets to the clients.

The chapter is divided as follows: Section 2.2 presents a short history behind the SOA paradigm. A detailed description of basic SOA terms used in the scope of this work is presented in Section 2.3. The last Section 2.4 lists some of the challenges faced by SOAs.

2.2 A brief history of SOAs

With the pervasiveness of devices like smart phones, tablets, and web applications in our every day life, the object-oriented paradigm was a hinderance to solve the cross-platform problems. This need fueled the emergence of distributed computing (component technology), whereby the client and server objects need not to be executed on the same machine [Daigneau, 2011]. This gave birth to technologies like Java Remote Method Invocation (RMI) [JavaRMI, 1997], Common Object Request Broker Architecture (CORBA) [CORBA, 1991], and Microsoft Component Object Model/Distributed Common Object Model (COM/DCOM) [COM, 1993].

Although this shift from "local objects to distributed objects" solved the portability issue among different programming languages and platforms, but only worked given the client and server objects were using the same underlying platform [Englander, 2002]. Moreover, it gave birth to new issues like effective load-balancing and memory utilization by the server object. In order to solve issues arising from distributed computing, the concept of web services was introduced in year 2001, which should be independent of underlying technology and can solve interoperability issues. This new concept of loosely-coupled, distributed and heterogeneous web applications laid the basics for popularity of SOAs [Raman, 2009].

Another vital factor in the success of SOA has been the exponential growth of World Wide Web, through which businesses started expanding, connecting different parts of the world, and shortening physical distances by expanding their services around the globe. This growth with a passage of time required an architecture, which can better facilitate the continuous integration of new functionality to the business. Some typical examples for this shift are the most famous companies like Amazon, eBay, Facebook, Twitter and Google. Today their services have billions of users around the globe, and have become a part of our every day life.

SOA history can be divided into three generations; the early SOA model (2001- 2005) was the era of web service technology; the second generation of SOA (2005 -2010) focused on service composition and related standards for QoS, security and reliability; the third generation covers the time span from year 2010 onwards.

2.2.1 Early SOA

The emergence of SOA is tightly coupled with the development of Web service technologies and standards. It happened at the time of emergence of distributed computing, when there was a strong need to exchange the information stored in one computer over a network in a portable format. A Web service represents a basic unit in a SOA-based business process, where the main purpose of a web service is to perform a certain task in response to a certain request from another web service. According to [Englander, 2002],

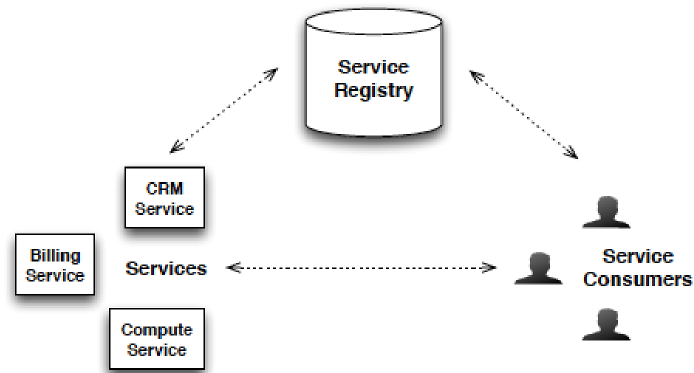


Figure 2.2: An early SOA Model

Web services represent, "server functions with published interfaces, needed to access their capabilities using standardized protocol".

The early SOA model is composed of three main components: services; service registry; and service consumers as shown in Figure 2.2. The services descriptions are expressed in Web Services Description Language (WSDL) [WSDL, 2001], a form of XML document, and are stored in a registry. The service consumers can look up the registry for available services, and should be able to connect to the desired service via message protocols like SOAP (Simple Object Access Protocol) [SOAP, 2007] or REST (REpresentational State Tranfer) [Fielding, 2000]. The bindings were static in this era. This era was taken over by the service compositions era as depicted in Figure 2.3.

2.2.2 Second generation of SOA

The second generation of SOA encompasses the time period from year 2005 to year 2010. In this period non-functional requirements of SOA processes such as service composition, message security and reliability were main research topics. In this respect, Business Process Execution Language (BPEL) came out as an industry standard. The research focussed on the orchestration

of business processes and related issues. More details about BPEL are presented in Section 2.3.2.

2.2.3 Contemporary SOA

The third generation of SOA covers the time span after 2010. In this era, business processes just became a part of the big picture presented in Figure 2.3. On the front end, the user interacts with a web interface, but in the background, there are many different components working together in synergy in order to facilitate a user. Depending on the size of the particular SOA implementation, the number of back-end components may vary. In a small infrastructure, we may have just a couple of business processes co-ordinating different web services running on the intranet. These web services could perform a variety of functions such as billing services, customer-relationship management service or computational services wrapping up the functionality of already running legacy systems. In large-scale organizations, usually enterprise service buses (ESBs) are used for enterprise integration. Some of the key features of a message bus include loose coupling among diverse web services by using XML as a communication language. It also supports synchronous and asynchronous communication with the help of standardized message routing services. A more detailed overview of ESB and its integration into business can be looked up in [Balepur Venkatanna Kumar, 2010].

2.3 SOA basics

Service-oriented Architecture (SOA) is more like an architectural style rather than a specific technology; for it is a collection of laws and policies to implement a model-driven service development [Josuttis, 2007]. In order to be self-contained, we review the relevant SOA terms and definitions within the scope of this thesis.

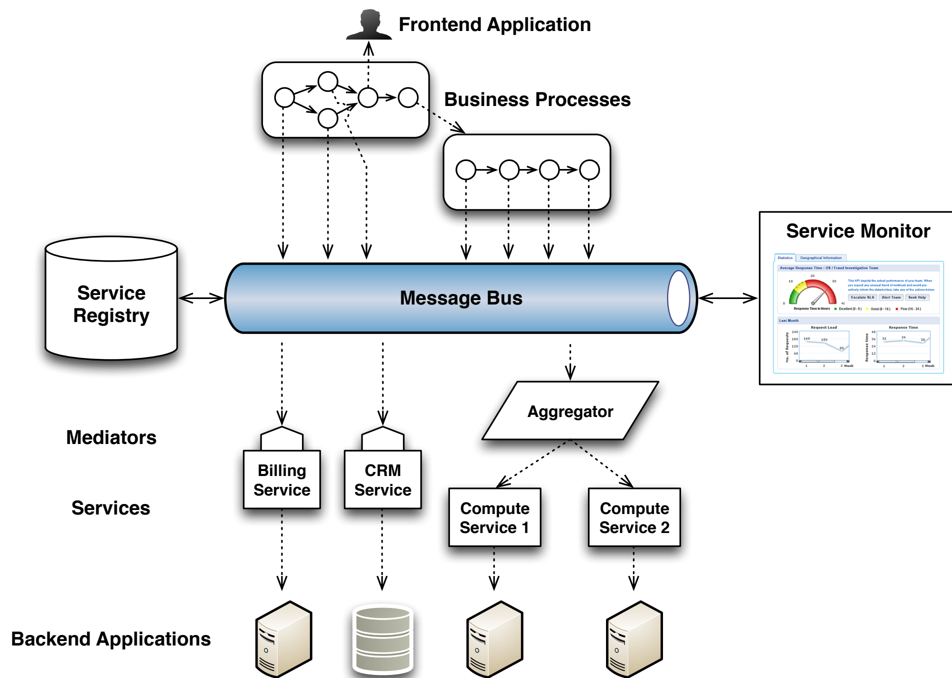


Figure 2.3: Contemporary SOA Model taken from [Leitner et al., 2013]

2.3.1 First generation Web services technology

There exist many definitions explaining the SOA paradigm in the literature. Some of them are listed in this section. According to Organization for the Advancement of Structured Information Standards Reference Model (OASIS RM): [Model, 2006]:

"Service-Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."

The "distributed capabilities" are also termed as Web services, and are considered the fundamental building blocks of any SOA. [Erl, 2007] defines SOA as follows:

"Service-Oriented Architecture represents a distinct form of technology architecture designed in support of service-oriented solution logic which is

comprised of services and service compositions shaped by and designed in accordance with service-orientation."

Since service lies at the heart of any service-oriented solution, it is important to understand first the concept of a service.

Web Services

A Web service is defined by W3C[2] as follows:

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [W3C, 2011].

According to another definition from [Erl, 2005]: "A Web service is composed of three main components: a service contract, which contains publicly available functions in the form of WSDL document; a business logic part, which encompasses the implementation details; a message processing logic, using message passing protocol like SOAP or REST".

A Web service can act as a service provider, or a service consumer or both (incase of service compositions). Figure 2.4 depicts different roles a Web service can perform based on its usage.

The basic Web service technology stack comprises of technologies: Web Services Description Language (WSDL), XML Schema Definition language (XSD), Simple Object Access Protocol (SOAP), Universal Description, Discovery, Integration (UDDI) [Erl, 2005]. All of these technologies use XML as an underlying language. The brief explanation of these Web services technologies is as follows:

[2] World Wide Web Consortium: https://www.w3.org/

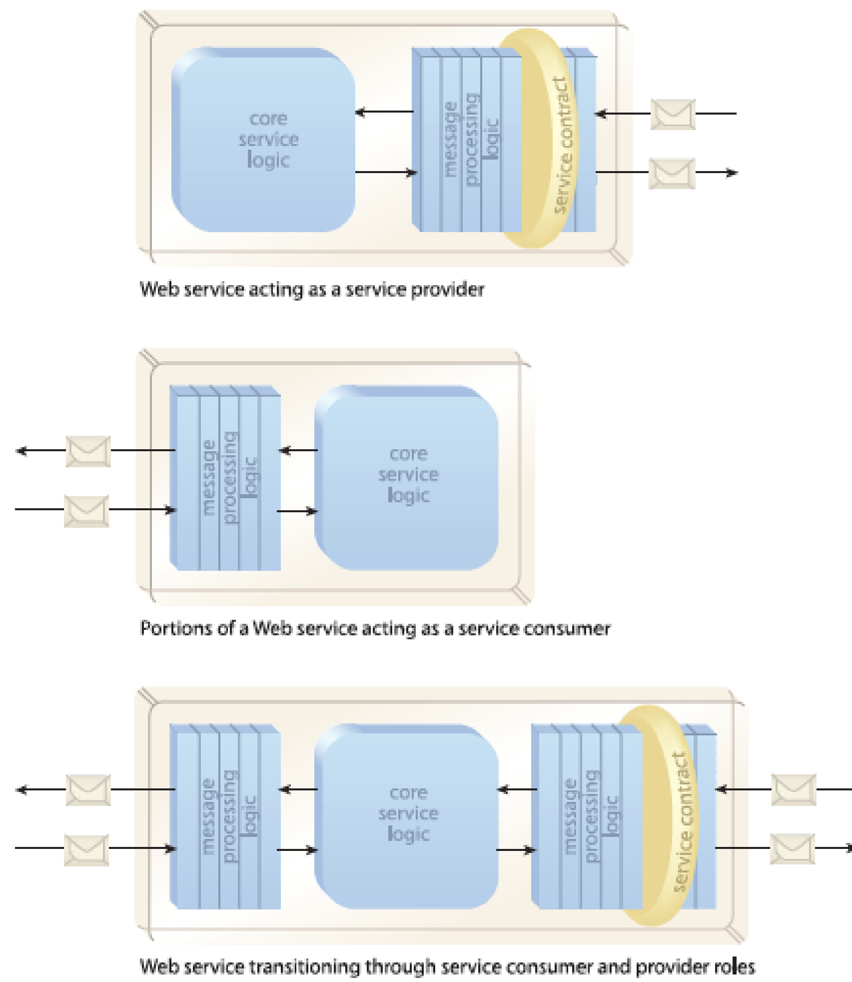


Figure 2.4: Web Service roles taken from [Erl, 2005]

XML Schema Definition

XML stands for extensible markup language[3] and is considered as a "lingua franca" for distributed systems [Hewitt, 2009]. With XML, it is possible to describe the data in a standardized way that is also portable over a network. The success of XML lies in the platform independent data model. It was designed such as to address extensibility issues in earlier markup languages such as HTML. Extensibility means that one can describe the content specific to one's application in a standard way.

The structure of an XML document can be specified in an XML schema, which is designed to express the restrictions on the elements defined in any XML document. A detailed description of a correct XML document, its constituent elements, the ordering of defined elements, possible constraints on already specified elements can be looked up on the official website of the World Wide Web consortium[4].

A simple XML document can be seen in Figure 2.5. *Schema* is the root element of every XML document. The root element may contain further elements of either simple type or complex type. A simple element is an XML element that can contain only text. The text can be of different types such as string, integer, boolean, date and time. It can not contain further elements. The complex type element, on the contrary, can contain simple types, where each element must also have a valid datatype specified in the form of an xml schema. The XML schema must not only be well-formed, but also valid against the XML Schema in order to be used by any XML tool.

SOAP protocol

SOAP is a message-based exchange protocol. It stands for Simple Object Access Protocol [SOAP, 2007]. SOAP messages are XML documents called envelopes. Every SOAP message must conform to the SOAP specification. According to the specification, a valid message must have three main

[3] www.w3schools.com/xml/
[4] http://www.w3.org/

```
1  <?xml version="1.0"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3  <xs:element name="input">
4    <xs:complexType>
5      <xs:sequence>
6        <xs:element name="to" type="xs:string"/>
7        <xs:element name="from" type="xs:string"/>
8        <xs:element name="heading" type="xs:string"/>
9        <xs:element name="body" type="xs:string"/>
10     </xs:sequence>
11   </xs:complexType>
12 </xs:element>
13 </xs:schema>
```

Figure 2.5: A short example of an XML Schema.

components *Envelop*, *Header*, and *Body*. The "Envelop" element is the root element of any SOAP message, and encapsulates the "Header" and the "Body" elements as shown in Figure 2.6. A detailed description of the SOAP message-passing protocol can be looked up in [Englander, 2002] or in [Balepur Venkatanna Kumar, 2010]. Like XML, SOAP messages are also platform-neutral and are independent of the implementation of the sender and receiver [Englander, 2002]. In principle, it is independent of the underlying transfer protocol, but is mostly used with the Hypertext Transfer Protocol (HTTP) [Berners-Lee, 1989] for communication on the web. There are also other binding protocols like Simple Mail Transfer Protocol (SMTP) [RFC, 1982], File Transfer Protocol (FTP) [Bhushan, 1985] available, but HTTP remains the most widely used protocol to date.

WSDL

WSDL stands for Web Service Description Language [WSDL, 2001]. "It is an XML-based interface definition language that is used for describing the functionality offered by a web service"(Wikipedia). A WSDL document enables the loose coupling between web services by encapsulating the internal functionality of a service. A typical WSDL document is composed of six elements; types, message, portType, binding, port and service. All

```
1  <?xml version="1.0"?>
2  <SOAP-ENV:Envelope >
3      <SOAP-ENV:Header>
4          ...
5          ...
6      </SOAP-ENV:Header>
7      <SOAP-ENV:Body>
8          ...
9          ...
10         <SOAP-ENV:Fault>
11             ...
12             ...
13         </SOAP-ENV:Fault>
14         ...
15     </SOAP-ENV:Body>
16
17 </SOAP_ENV:Envelope>
```

Figure 2.6: A SOAP message.

these elements are inter-related to each other and can be classified into two parts; an abstract part and a concrete part. An *abstract* part of a WSDL document describes the interface level operations provided by web services using *portType* definition. These operations are linked to input and output messages used by the web service for exchange of communication with other web services. The *Types* section is used to define the data types of input and output variables used in the respective messages section. In order to execute the web service implementation, the concrete part is required, which is composed of three further elements. The *Binding* section refers to the physical transport protocol used for the on-the-wire communication. The bindings can be specified using multiple protocols such as HTTP, or SOAP. Also, multiple bindings can be made available for the same *portType* operations. The services element contains the link to the specific web address to access the web service functionality.

```
1  <definitions>
2
3      <types>
4      </types>
5
6      <message>
7      </message>
8
9      <portType>
10             <operations>
11             </operations>
12       </portType>
13
14       <binding>
15       </binding>
16
17       <services>
18       </services>
19
20 </definitions>
```

Figure 2.7: WSDL components taken from [WSDL, 2001].

UDDI

Like SOAP and WSDL, Universal Discovery, Description and Integration (UDDI) [UDDI, 2001] is considered a basic block of a web service stack. As the name suggests, it is used for storing, accessing and retrieving web services. However, UDDI has not got industry acceptance like SOAP and WSDL [Erl, 2005].

REST

REST stands for "REpresentational State Transfer". It was first introduced by Roy T. Fiedling in year 2000, who claimed in his Ph.D. thesis that: " the REST architectural style has been used to guide the design and development of the architecture of the modern Web" [Fiedling, 2000]. Twitter and Amazon are two examples of REST protocol. The success of REST services depends on many factors: they are stateless, they should not store state on the server; the REST-based services don't require WSDL; the resources are used to build the RESTful architecture, each having a unique identifier URI; Last but not the least, REST protocol discourages the use of cookies, rather, stresses on using the hypermedia to store the application state. REST unlike WSDL, SOAP, and XML Schema is not a specification. A detailed discussion of RESTful architecture and REST-based services can be read in [Richardson and Ruby, 2007].

2.3.2 Second generation Web services technology

The sole purpose of building web services was to provide loose coupling, interoperability, reusability, and discoverability between heterogeneously-built autonomous services. The first generation of web service technology lack reliable messaging protocol. Therefore, the second generation of Web services focussed on extensions for security at message level, transport and network level. These extensions were offered by specifications such as Web Services Security (WS-Security), Web Services Secure Conversation (WS-SecureConversation), and Web Services Security Policy (WS-Security Policy) [Erl, 2005]. Another notable contribution was the development of

Business Process Execution Language for Web Services (BPEL4WS). Being part of WS-* extensions stack, BPEL4WS is sometimes also written as WS-BPEL, or even BPEL for short. BPEL emerged as an industry standard for the execution of long running, complex business processes spanning over a large period of time.

WS-BPEL

WS-BPEL is an XML-based language for the definition and execution of business processes. The sole purpose of BPEL is the composition of heterogeneous web services in order to achieve a business goal. Although there are also other workflow languages such as Business Process Modeling Notation (BPMN) [OMG, 2005], Yet Another Workflow Language (YAWL) [ter Hofstede, 2010], WS-BPEL is the OASIS standard. It contains constructs of both workflow languages and programming languages. According to [Juric and Krizevnik, 2010], some of the design goals of BPEL were: the description of a business logic through composition of services, handling of synchronous and asynchronous long-running processes, invocation of processes in sequential and parallel fashion, the parallel execution of activities depending on the synchronization conditions, and the correlation of requests to particular instances within a business process.

History of WS-BPEL

Before the emergence of WS-BPEL, web services were already adapted in the industry. There was lot of work published [Bozkurt et al., 2013] until 2005, when IBM and Microsoft proposed a language for the composition of individual web services to achieve a business level integration. The aim was to provide a standard language to deal with reliable messaging problems. This new language was created purposefully to remove problems in the already used workflow languages such as WSFL [Leymann, 2001] by IBM and XLANG [Microsoft, 2001] by Microsoft. WSFL (Web Services Flow Language) was meant for directed graphs, and XLANG was a block-structured language. In BPEL language, one finds both directed graphs and block-structured approaches for modeling.

WS-BPEL Preliminaries

The most important concept in understanding WS-BPEL is the term "process" itself. According to [Harvey, 2005], a process denotes "a program running in an operating system, responsible for processing a request over some interval of time". The term *process definition* defines behavior of the process, whereas the term *process instance* refers to "an occurrence of a process to a specific input". For example, each instance of a loan business process refers to a specific loan request. Moreover, an execution engine is employed to "create and run instances of a given process definition". And each single step in a process, such as approving a loan is defined as an"activity".

Types of BPEL processes

There are two main types of BPEL processes: an abstract process, and an executable process. An *abstract process* is only a protocol definition, whereby an *executable process* is actually executed in a process engine. The primary difference between a typical web service and a BPEL process is that the user only has the description of a web service, but can never access the internal functionality of a web service. However, the functional logic of a BPEL process is available as an XML document. In fact a BPEL process itself is exposed as a web service. An executable BPEL process is shown in Figure 2.8. There are two types of files in an executable BPEL process:

- WSDL files for the specification of web service interfaces, including portTypes and operations, related to the business process.
- BPEL file(s) for the specification of a process definition in an XML format.

WS-BPEL Constructs-Activities

A WS-BPEL file is divided into two basic parts, i.e., one part describes the structure of the business process, and the second part describes the binding to actual services responsible for execution of a certain functionality. The

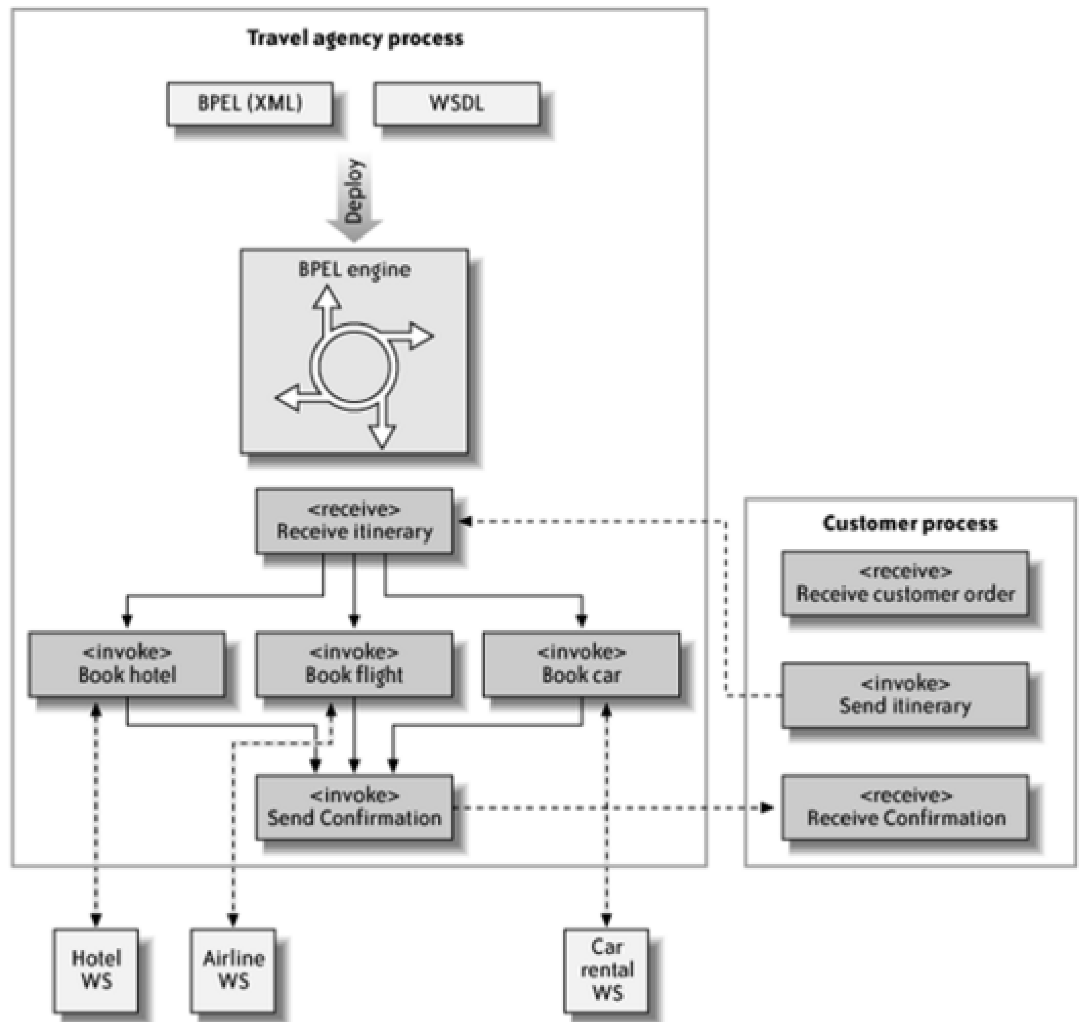


Figure 2.8: An Executable BPEL process taken from [Harvey, 2005]

document is composed of basic and structured activities. The basic activities include *receive*, *reply*, *invoke*, *throw* and *wait*. The structured activities include constructs like *while*, *sequence*, *flow* and *pick*. The basic difference between the *sequence* and the *flow* activity is the order of the execution of activities. A *flow* construct is used to define a set of activities that can be executed concurrently. The *pick* activity is responsible for handling external events such as *alarm* or *messages*. A basic skeleton of a BPEL is shown in Figure 2.9. A detailed description of all BPEL constructs and their practical usage can be looked up in [Erl, 2005].

A typical BPEL process has four main parts: *partnerLinks*, *variables*, *faultHandlers*, and any number of structured or basic *activities*. The *partnerLinks* are defined in order to make a BPEL process interact with other processes. In addition to that, the *variables* section is meant for defining data variables used by the process definition. In case of any exception, a fault handling mechanism can be defined in a *faultHandlers* section. This is generally followed by one or more structured activities such as *sequence*, *if*, *while*, *pick* or *flow*. These activities can be nested in any combination or order. Let us briefly describe the most common activities:

Receive Activity

A *Receive* activity is typically the starting activity within any business process. It is used for receiving requests from partner processes. As shown in Figure 2.10, the mandatory fields for any *Receive* activity are the name of the *partnerLink*, and the *operation* that the partner business process is calling. The incoming request message will be stored in a *variable* field.

Reply Activity

A *Reply* activity is an "activity that returns a synchronous reply to an incoming web service call triggered by a receive" [Harvey, 2005]. This activity is mandatory in a request-reply operation, in which the business process is required to send back a reply to the calling partner process. For that reason,

```
1  <process>
2
3     <partnerLinks>
4      ...
5     </partnerLinks>
6
7     <variables>
8      ...
9     </variables>
10
11     <faultHandlers>
12         ...
13     </faultHandlers>
14
15     <sequence>
16             <receive ... >
17             <invoke ... >
18             <assign ... >
19             <if>
20             <while>
21             <reply ... >
22              ...
23     </sequence>
24      ...
25  </process>
```

Figure 2.9: BPEL process definition

```
1  <receive partnerLink="NCName"
2      portType="QName"?
3      operation="NCName"
4      variable="BPELVariableName"?
5      createInstance="yes|no"?
6      messageExchange="NCName"?
7      standard-attributes>
8      standard-elements
9      <correlations>?
10         <correlation set="NCName" initiate="yes|join|no"?/>+
11     </correlations>
12     <fromParts>?
13         <fromPart part="NCName" toVariable="BPELVarName"/>+
14     </fromParts>
15 </receive>
```

Figure 2.10: Receive Activity

it is linked to the same *partnerLink* element as specified in a *Receive* activity. Figure 2.11 depicts a snippet of *Reply* activity from BPEL specification.

Invoke Activity

An *Invoke* activity is meant for calling the partner business process operation. This can be a one-way operation or a request-response scenario, in which *Invoke* must wait until a response is received. As shown in Figure 2.12, both *inputVariable* and *outputVariable* fields are optional.

Assign Activity

An *Assign* activity as illustrated in Figure 2.13 is used for updating variable values. An *assign* activity can contain one or more *copy* constructs.

```
1  <reply partnerLink="NCName"
2      portType="QName"?
3      operation="NCName"
4      variable="BPELVariableName"?
5      faultName="QName"?
6      messageExchange="NCName"?
7      standard-attributes>
8      standard-elements
9      <correlations>?
10         <correlation set="NCName" initiate="yes|join|no"?/>+
11     </correlations>
12     <toParts>?
13         <toPart part="NCName" fromVariable="BPELVarName"/>+
14     </toParts>
15 </reply>
```

Figure 2.11: Reply Activity

If Activity

An *If* activity as shown in Figure 2.14 is used for selecting one activity among a set of choices. When a *condition* attribute is set to "true", the nested activities are executed like any other programming language.

WhileActivity

A *While* activity 2.15 is similar to any imperative programming language, and is used for repetition of activities as long as the loop condition remains true.

Limitations of BPEL

It is important to note that BPEL is only useful in describing or modeling the functional aspects of a process. It is not well suited for specifying the non-functional properties associated with any process like security requirements and Service-level-agreements (SLAs).

```
1  <invoke partnerLink="NCName"
2      portType="QName"?
3      operation="NCName"
4      inputVariable="BPELVariableName"?
5      outputVariable="BPELVariableName"?
6      standard-attributes>
7      standard-elements
8      <correlations>?
9          <correlation set="NCName" initiate="yes|join|no"?
10             pattern="request|response|request-response"?/>+
11     </correlations>
12     <catch faultName="QName"?
13         faultVariable="BPELVariableName"?
14         faultMessageType="QName"?
15         faultElement="QName"?>*
16         activity
17     </catch>
18     <catchAll>?
19         activity
20     </catchAll>
21     <compensationHandler>?
22         activity
23     </compensationHandler>
24     <toParts>?
25         <toPart part="NCName" fromVariable="BPELVarName"/>+
26     </toParts>
27     <fromParts>?
28         <fromPart part="NCName" toVariable="BPELVarName"/>+
29     </fromParts>
30 </invoke>
```

Figure 2.12: InvokeActivity

```
1  <assign validate="yes|no"? standard-attributes>
2      standard-elements
3      (
4      <copy keepSrcElementName="yes|no"?ignoreMissingFromData="yes|no"?>
5          from-spec
6          to-spec
7      </copy>
8      |
9      <extensionAssignOperation>
10         assign-element-of-other-namespace
11     </extensionAssignOperation>
12     )+
13 </assign>
```

Figure 2.13: AssignActivity

```
1  <if standard-attributes>
2      standard-elements
3      <condition exprLang="anyURI"?>bool-expr</condition>
4      activity
5      <elseif>*
6          <condition exprLang="anyURI"?>bool-expr</condition>
7          activity
8      </elseif>
9      <else>?
10         activity
11     </else>
12 </if>
```

Figure 2.14: IfActivity

```
1  <while standard-attributes>
2      standard-elements
3      <condition exprLang="anyURI"?>bool-expr</condition>
4      activity
5  </while>
```

Figure 2.15: WhileActivity

Furthermore, the BPEL language received a lot of criticism from the testing community as it is described in a natural language [Lapadula et al., 2008], which makes it hard to formalize different constructs. Despite of this fact, BPEL remains an OASIS standard for composition of heterogeneous, disparate web-services. There is another issue with BPEL compositions, i.e., their tight coupling with WSDL-based web services. And, because of this limitation, BPEL can not work well with semantic web services or for that matter REST-based web services.

2.4 SOA challenges

While the decomposition of required functionality into services give different teams more autonomy, i.e, helped Twitter attain higher levels of concurrency, it opened new challenges for testing these "loosely-coupled" services. [Daigneau, 2011] describes various types of dependencies among web services such as functional, temporal and URI coupling. According to [Josuttis, 2007], "Loosely coupled distributed systems are harder to develop, maintain, and debug". He argues that the goal of introducing loose-coupling was to reduce dependencies among different parts of a large-distriubted system. For example, asynchronous communication is the most common form of loose-coupling in SOAs. But, asynchronous messages must be handled appropriately; first the reply must be associated with the original request, second the state of the original request must be stored in order to process the reply in the proper context. The situation might get worse when the order of request-reply messages is altered, or if some messages do not arrive at all. Handling all such possibilities at development level, and later on at testing and debugging would mean additional cost and complexity.

[Canfora and Penta, 2009a] listed various challenges associated with SOA applications. The first challenge is the limited observability of the services' code, for users do not possess access to the internals of the implementation. Similarly, end users have no control over consumed services, because these services are executed on the providers end. Moreover, the services' implementations can be updated without an end-user awareness, and this can pose an extra overhead regarding regression testing at the Integrator

end. Above all, the testing cost of such dynamic and hybrid large-scale applications is quite high, both at the service and composition level.

In addition to the previous mentioned challenges, SOA applications must be tested for Quality of Service (QoS) attributes, such as response time, availability, system load, cost and other attributes specified in service-level agreements. Our focus has been on the functional aspects of SOA-based processes.

3 Preliminaries and Related Work

Software testing coupled with verification and validation has been considered inevitable in any software development life cycle. *Software testing* means "the process of analyzing a product to verify that it satisfies specified requirements or to identify differences between expected and actual results" [IEEE, 1983]. *Verification* is "the process of determining whether or not the products of a given phase of a software development process fulfill the requirements established during the previous phase". *Validation*, on the other hand, means "the process of evaluating software at the end of its development process to ensure compliance with its requirements" [IEEE, 1983].

Testing is a commonly practiced verification technique, in which the system under test is executed with selective inputs, and the resulting output is compared with expected results, in order to verify the requirements stated in the specifications. Although there have been well-established testing techniques and methods to ensure software quality and reliability, testing is still considered the most difficult, laborious and error-prone part of a typical software development process [Myers et al., 2011]. With the rise of service-oriented computing, the problem of software testing has become more intricate and complex than ever. For example inherent features such as limited controllability and observability have further complicated the functional testing of SOA business processes [Hierons and Ural, 2008], [Hierons and Ural, 2009].

Once the fault is identified, naturally one is interested in finding out probable reasons behind the occurred fault. In this way, the diagnosis process complements a testing process. There are various challenges specific to diagnosing business processes. For example, the complete specification of the input/output of composed services is not available. Moreover, the diagnosis needs to consider the state of a web service in fault-localization, diagnosis

and also repair actions [Friedrich et al., 2010b]. In this Chapter, we present issues and challenges specific to the functional testing and debugging of SOA-based business processes. We present a light-weight debugging approach for detection of behavioral functional faults in BPEL compositions.

This Chapter is organized as follows: First, some general definitions from software testing are presented in Section 3.1. Later, related work available in the context of SOA testing is discussed in Section 3.2. Similarly, the relevant debugging definitions are given in Section 3.3. This is followed by the related work in the diagnosis of service-oriented processes in Section 3.4 respectively. We present conclusions in Section 3.5.

3.1 Testing Preliminaries

Software testing is ”the process of executing a program with the intent of finding errors” [Myers et al., 2011]. The term ”Error” basically means source responsible for deviation of the system under test from its expected behavior. According to [IEEE, 1983]: an *error* is “the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. There can be many different types of errors, and according to [Goodenough and Gerhart, 1975], can be broadly classified as performance and logical errors. These errors can be observed by: either a missing path; a wrongly taken path; a missing computation, and a wrongly computed value. An error can result in software containing a *fault*, which is defined as “an incorrect step, process, or data definition in a program”. Similarly, a fault is responsible for a *failure*, which denotes an incorrect behavior of a program. The exact definition of a failure is “the inability of a system or component to perform its required function within the specified performance requirements” [IEEE, 1983].

According to [Adrion et al., 1982], a complete verification of any program is only possible through ”exhaustive Testing”. As this goal is in reality not possible, determining the scope of any testing activity is of utmost importance. Some of key challenges in testing involve generating test data, measuring the test data quality, automating test execution, and prioritizing test case selection [Myers et al., 2011].

The first step is the selection of the test data which serves as an input to the system under test. As a second step, it is important to determine if the test data uncovers faults or not. This is known as the *"test oracle"* problem [Howden, 1978]. Similarly, measuring the quality of test data helps in deciding when to stop testing, because complete testing is not possible in practice. Two such methods are: statistical methods or deterministic methods [Adrion et al., 1982]. The *statistical methods* are random in nature as compared to *deterministic methods*, in which test data must reproduce same results under the given set of conditions. Since testing covers almost 50% of the software development cycle [Myers et al., 2011], automation of the testing process is useful in reducing the testing effort, time and cost. Likewise test case prioritization goal is to lower down the testing cost by discovering more errors with less number of tests.

[DeMillo et al., 1978] presented two hypotheses in their paper; *"competent programmer hypothesis"* and a *"coupling effect"*. They argued that programmers are "close to correct", and based on this assumption, they claim that the frequency of simple errors is alway higher than the complex ones. Or putting differently, simple errors are "coupled" with complex errors. So, it makes more sense to look for these simple errors, which lead to the complex one eventually. This lead to another question, that is, which errors should we test for? This question can be hard to answer, as it mainly depends on the nature of the underlying program or system. For example, for embedded systems, functional testing might be enough to test most of the functionality, but in SOA-based environments, functional testing must go hand in hand with the non-functional testing of software attributes like response-time, network-faults, and testing cost. This thesis is about functional testing of SOA-based systems, and is aimed at complementing non-functional testing of such systems.

Test data generation: There are three main types of test data generation techniques, i.e., black-box, white-box, and grey-box. *Black-box testing* is a functional analysis technique, and is useful when the internal logic of the SUT is not available. This type of testing is done in scenarios, where inputs and expected outputs are known, but not the internal program structure. This implies that test data is generated from requirements or design specifications only, because it stresses only on the external behavior of the system under test. An exhaustive *Black-box testing* would consider

all possible test inputs to test for the expected result (oracle), which is impossible for larger programs. Hence, methodologies like boundary value analysis, equivalence partitioning are employed to measure the test data adequacy [Myers et al., 2011]. The advantage of using *black-box testing* is that no prior programming knowledge is required. One drawback is that tester might not be able to test certain parts of the code due to visibility problem. It might also happen that unnecessary tests are generated to test certain software functionality.

White-box approach, on the other hand, is a structural analysis technique, and is used when the source-code of the program under test is available. It is called white-box, because the internal structure of the program is known prior to testing. The classical techniques applied in the context of white-box testing include mutation testing, API testing. The target of selecting test cases is the execution of every path in the program; which is realistically not possible especially in programs with loops. Therefore the test suite adequacy is measured with the help of some code coverage criteria such as branch coverage, statement coverage or path coverage [Myers et al., 2011]. For example 100% branch coverage implies that all branches in the control flow graph of the system under test have been executed once. This certainly provides some confidence in the testing approach, but is not complete since the same branch can be tested with different sets of inputs yielding unexpected results. The approach is mostly applied to the unit level. The disadvantage of using *white-box approach* is that missing functionality from the specification or requirement documents can not be detected. In addition to that, a thorough understanding of the SUT implementation is required to generate useful tests.

There is a third kind of testing approach, *grey-box testing*, in which partial information about the internal structure of the program is available. In the context of this thesis, the control-flow logic of a BPEL process is available, but the underlying functioning of the involved web services is a black-box. Hence we suggest grey-box testing approach for testing SOA business process.

There are also many types of testing with different testing goals: *functional testing* checks if a software conforms to its requirements specifications. *Non-functional testing* evaluates a software for non-functional requirements such

as performance, load, and scalability. *Regression testing* is to ensure if a software fulfills its design specification after changes such as configuration or software updates. *Integration testing* is performed "when individual software modules are combined and tested as a unit" (Wikipedia).

Test data adequacy: Once the test data is generated using either static analysis such as symbolic execution, model-checking, or dynamic analysis method, the next step in the testing process is to determine the effectiveness of the generated test suite. This can be done in many ways depending on the budget of the particular project. The two commonly used methods are coverage analysis and mutation analysis.

Mutation analysis: [DeMillo et al., 1978] were the first to introduce this technique. The idea is as follows: First, by applying syntax changes to the original program, many possibly faulty copies of the program under test are generated. These altered versions of the original program are called "mutants". These mutants are generated according to certain mutation operators [Boubeta-Puig et al., 2011]. Each mutation operator denotes different fault class, and the goal is to generate mutants representing different fault classes. Second, the generated mutants are tested with the test suite in order to determine if the test suite can differentiate between the original program and mutants or not. A mutant is termed "killed" if a test case can distinguish it from the original program. And the total number of killed mutants over non-equivalent mutants denote the adequacy score or the"mutation score". The higher the mutation score, the better the quality of the test suite. The mutants which the test suite can not distinguish from the original program are called "equivalent mutants". The scope of mutation testing is limited in practice primarily because of high computational cost. In addition to that, it is still undecidable whether an equivalent mutant is equivalent to the original program or not [Budd and Angluin, 1982]. The survey gives a comprehensive overview of the the work done in the field of mutation testing since its origin [Jia and Harman, 2011].

The "SUT" or "program", in the context of this thesis, means software artifacts such as BPEL source code, schema definitions of the composed web services and WSDL specifications. In general, this could mean the requirement documents, stating the functional and non-functional quality attributes of the project under consideration. The expected behavior of the

SUT is extracted from the structure of a BPEL process document and related artifacts. Once we have the model out of a BPEL specification, we derive the possible input/output relationship using the constraint-based approach discussed in detail in next Chapter 4.

3.2 Testing trends of SOA Applications

Testing approaches can be divided into many different types aiming at testing different behavior of the system under test. With respect to SOA-based or service-centric systems, one of the earliest survey was published by [Canfora and Penta, 2006]. This survey gives a glimpse of testing issues from the perspective of functional and non-functional point-of-view. Also another survey done by same authors [Canfora and Penta, 2009a] provides a good overview of testing challenges posed by such system from viewpoint of different stakeholders. A detailed account of various testing approaches used in the service-oriented architecture can be found in [Bozkurt et al., 2013]. In both of these surveys, the underlying issue discussed in SOA testing is that of "trust".

This lack in trust comes from the limited observability and controllability of SOA-based systems. The limited observability means that users or testers have no knowledge about the internal implementation of the service. Therefore a tester can not apply white-box approaches to verify that the service indeed comply to the agreed service level agreements (SLAs). The limited controllability comes from the fact that a service provider can change the service behavior without prior notification to users. As a result, users may experience unexpected changes in the service functional or non-functional behavior.

In classical testing techniques, the issue of limited observability and controllability is not that much of an issue as in service-oriented architectures [Hierons and Ural, 2008], [Hierons and Ural, 2009]. In scope of this work, functional testing approach for the BPEL compositions has been explored. We report on the approaches that are closely related to our approach in the context of test case generation, execution and diagnosis of BPEL compositions.

There is a lot of research already done in testing SOA applications; this work intersects mainly with model-based test case generation approaches. The survey [Bozkurt et al., 2013] classifies model-based testing and formal verification of web services into symbolic execution, model-checking and petri-nets. A short description of each of these testing approaches is provided in the corresponding section. This work is based mainly on symbolic execution principles, and also partially on contract-based testing of composed web services. Moreover, we make use of some work done in unit testing and fault-based testing for the test suite execution step.

3.2.1 Symbolic Execution Approach

Symbolic execution is a program analysis technique testing programs on symbolic inputs rather than concrete input values. In addition to that, a path condition is stored whenever a branch condition is executed. The purpose for storing the path condition is to maintain path constraints. Once all constraints are encoded as a constraint satisfaction problem, a constraint solver is used for the concrete test case generation.

The idea of symbolic execution for program testing was first introduced by James King [King, 1976] to test programs for infinite large classes of inputs. The approach is also called static symbolic execution, for it only analyses the source code of the program to predict the program behavior. Among the classical symbolic execution tools, SELECT [Boyer et al., 1975] can handle only sequential programs with limited number of input data types.

One drawback of using static symbolic execution is that it can not model the behavior of external system calls. The dynamic symbolic execution aims at modeling not only the program structure, but also the unknown program behavior such as interactions with the environment. While our work is based on classical symbolic execution, the usage of dynamic symbolic execution for test input generation has been explored extensively. DART was the first tool built on the idea of dynamic symbolic execution [Godefroid et al., 2005a]. DART (Directed Automated Random Testing) works on dynamic symbolic execution with concrete inputs, collects symbolic constraints on the input to generate feasible paths using a constraint solver. It searches for different

variants of the previous input to search for other possible execution paths. The goal is to find all feasible execution paths. Unlike static symbolic execution, dynamic symbolic execution has the advantage of deriving new test inputs without any test driver. Also, the behavior of environment can be modeled to test the dynamic program execution such as pointer analysis and library function calls. The tool was developed to test C implementation.

Path explosion is another problem associated with static symbolic execution. [Sen and Agha, 2006] introduced the term *concolic* testing, which is a hybrid of symbolic and concrete execution. They developed a tool called CUTE, which basically extends DART by introducing concolic testing for unit testing of C programs. Like DART, CUTE uses both concrete and symbolic test inputs to leverage limitations of the symbolic execution. The tool has two versions, CUTE supports testing of C programs, and jCUTE is used for testing multithreaded Java programs. They use partial-order reduction techniques to reduce the number of infinite generated paths.

[Tillmann and De Halleux, 2008] presented a white box test generation tool (Pex) for .NET environments. Pex was developed to generate parametrized unit tests for .NET environments, whereby pointer access and floating point arithmetic often yield to unknown program behavior, which can not be reasoned with static symbolic execution. The feasibility of execution paths is determined using the constraint solver Z3 [De Moura and Bjørner, 2008]. However, it does not work in non-deterministic environments or for concurrent programs.

One of the limitations of symbolic execution is the complexity of constraints generation in case of programs containing loops or recursion. There are many search techniques (heuristic search, partial order and symmetry reduction search) discussed in the literature to reduce the number of generated paths. The interested reader can look up the Cadar et. al [Cadar et al., 2011] which provides an extensive overview of the state-of-the-art symbolic execution techniques.

In the SOA domain, symbolic execution tools face the challenge of handling dynamic behavior of the environment. [Bentakouk et al., 2011] suggested a black-box oriented conformance testing using a SMT (Satisfiability Modulo Theories) solver for constraint-based generation out of BPEL compositions.

They use symbolic transition systems as an underlying formal model for the behavioral representation of the system.

The survey [Zakaria et al., 2009] by Zakaria et al. gives a very good comparison of different unit testing approaches applied to BPEL processes. A key issue they pointed out is the lack of an empirical evaluation. Surprisingly, only one out of 27 considered studies provides results on real-life BPEL processes.

3.2.2 Model Checking Approach

Model checking is a popular technique used nowadays for the verification of software [Clarke and Lerda, 2007]. It was initially developed to verify finite-state concurrent systems [Clarke et al., 1983]. It has been used since then, for verifying both hardware and software systems. A model-checker takes two inputs: a state transition graph of some hardware (circuit) or a software program, and the specification, represented as temporal logical formulas. The goal of a model checker is to figure out if the specified formula holds true for the given model. A *witness* is a path in the execution model where the formula is satisfied. Similarly a *counterexample* is a path provided by the model checker, in which the specified formula does not hold true. These counter-examples are typically used as test cases for testing software.

There are many model-checkers used in practice today. A widely discussed model-based technique exploited in the context of web service testing is model checking [Bozkurt et al., 2013]. The general idea behind its application is to translate BPEL specifications into a formal modeling language like PROMELA [Garcia-fanjul et al., 2006] and test criteria into a formal property language like LTL [Pnueli, 1977]. Both specifications and properties then serve as an input to the SPIN model checker. The model checker provides counterexamples for the test case generation. Zhen et al. [Zheng et al., 2007] applied the same idea to web services and BPEL processes. There is a more enhanced work by same authors, in which Zhou et al. also address the state space explosion problem inherent with model checking [Zhou et al., 2007]. Moreover, they also developed a tool for the generation of JUnit test cases for automated test execution.

There are many advantages [Clarke, 2008] of model checking: It is faster than theorem proving, and even works in case of partial specifications. And, if the formula is not verified by the model, the model checker gives a counterexample to guide the user in locating fault in the model. So, model-checkers can be very useful in debugging software. Also, verifying concurrent properties is nearly impossible using manual approaches. But, the advantages come at the price of the state space explosion. The *state space explosion problem* refers to an exponential number of states generated by model checkers, which has a negative impact on the applicability of model checking to software. The problem can be addressed using techniques like bounded model checking [Clarke et al., 2012]. Besides, there is second issue, i.e., the formal specifications are hardly available.

3.2.3 Petri Net Approach

Petri nets are another modeling means for graphical representation of communication between distributed, and concurrent systems. Some typical examples include communication protocols, distributed-database systems and concurrent programs [Murata, 1989]. Therefore it is useful in determining problems related to concurrent behavior such as reachability, liveness, and soundness.

Model-based testing techniques using Petri Nets have also been explored extensively. Petri Nets are used for modeling concurrent processes, and can be categorized into Plain Petri Nets [Ouyang et al., 2007], Colored Petri Nets [Yang et al., 2005] and High-level Petri Nets. Dong [Dong, 2009] developed a tool for test case generation of BPEL processes using High-level Petri Nets. The basic approach is to build a reachability graph from which test cases can be extracted. The approach has a very high space complexity.

3.2.4 Graph-based Approach

Graphs can be employed for coverage-based testing of programs. It is a useful structural testing technique, where the idea is to represent the program formally in a graph-like structure, and the test data is generated

so as to test possibly all branches, or paths present in the model of the software under test. Considering the fact that some paths are infeasible, the main goal of such an approach is to attain maximum coverage such as path, statement, or decision [Adrion et al., 1982]. Both model checking and Petri nets require formal specifications of the system under test to be available, which is unfortunately not always possible. On the contrary graph-based approach does not have any such restriction. It only requires some coverage criteria and an underlying structure of the system under test.

Yuan et al. [Yuan Yuan and Sun, 2006] presented a graph search-based test case generation of BPEL processes that makes use of matrix transformations of control flow graphs, path coverage, and the classification of nodes in the graph depending on incoming and outgoing edges. Yuan et al.'s approach is close to ours in two ways. First, they suggest to transform a BPEL program to a control flow graph. Second, they generate test data using the Lp constraint solver [Berkelaar,], which is later on combined with test paths to generate abstract test cases. However, our approach differs from Yuan's work. Although we use the path coverage criterion for test path generation, we further add pre- and post condition contracts to test paths, in order to handle the test oracle problem. In addition, we use the Minion constraint solver [Gent et al., 2006] to generate test data instead of relying on Lp.

Another closely related work is from Yan et al. [Yan et al., 2006], which relies on an extended Control Flow Graph (XCFG). The idea behind their work is to extract all sequential paths from the XCFG, and to combine them into concurrent test paths. From these concurrent test paths they collect constraints using backward substitution. In our approach, we transform each sequential path directly into a set of constraints, which is checked for satisfiability directly using the Minion constraint solver. If constraints are not satisfiable, we discard the corresponding path. Otherwise, the constraint solver returns values for all variables used to execute the path, which we directly register as a test case in the test suite under construction.

3.2.5 Contract-based Approach

Contract-based testing techniques are also relevant to our work. Design-by-contract is a well-known software engineering technique for more reliable testing [Meyer, 1992]. In this approach, the developer needs to provide contracts, i.e., the pre-conditions and post-conditions, that must hold true before and after any access to the developed software component. The post-conditions can be very useful in the specification of test oracles, i.e., the expected test output. However, these contracts by the developer and the provider entail high costs, and are often ignored.

In the context of SOAs, this idea has been applied mainly in web service testing. For example, [Heckel and Lohmann, 2005] argue that contracts applied at the model level are useful in the automated generation of test oracles, but can be very costly to implement. Also, assertions are easier to apply in OWL-WS technology but are difficult to implement in a WSDL-based process model [Bozkurt et al., 2013]. Dai et al. combine this approach with Petri Nets [Dai et al., 2007]. They specify contracts using an OWL-S model and transform them into Petri Nets. The test cases are generated based on a Petri Net behavioral analysis.

Although there have been many approaches in defining contracts for web-services testing, there is still no Design by Contract standard for SOA [Bozkurt et al., 2013]. We combine contract-based testing with symbolic execution in our approach. The tester can specify contracts, i.e., pre- and post-conditions on the model derived from the BPEL specification. This additional information can increase the quality of generated tests by providing the required inputs and expected outputs for the external web services.

3.2.6 Search-based Approach

The Search-based approach is also a well known software engineering technique for test data generation. A search-based approach employs a metaheuristic search techniques for generating test inputs specific to a test goal. "A metaheuristic is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may

provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity" (Wikipedia). The test goal can be configured using a fitness function. Some typical meta-heuristic search techniques include genetic algorithms, simulated annealing and tabu search [McMinn, 2004]. It is widely used in many testing types, such as functional testing, regression testing, web testing and interaction testing.

The paper from Canfore et. al. [Di Penta et al., 2007] explores search-based testing of SOA systems in order to trigger SLA violations using genetic algorithm. They discuss their approach on two case studies. The approach might produce some false negatives due to multiple invocations of a service. They use two fitness functions based on the black-box and the white-box approach to cause SLA violations. They claim that the white-box approach performs better than the black-box. McMinn et. al. have presented a novel search-based approach for generating string data type as test inputs such as dates, banking codes, identity cards, social-security numbers etc. for testing ten java projects [McMinn et al., 2012]. Harmann and Bozkurt [Bozkurt and Harman, 2011] propose a tool about generating realistic test data and compare the effectiveness with randomly generated test inputs. They exploit existing web services for generating realistic input data.

[Blanco et al., 2009] applied the Scatter search to derive test cases for BPEL compositions. Instead of a control flow graph, they use a state graph to represent the business process logic and generate test cases considering the branch coverage criterion. Basically, the search process works on randomly generated solutions for each transition, the goal is to cover all transitions, but the search can be stopped when the maximum number of test cases have been achieved.

Although the problem of testing SOAs has been investigated thoroughly, the lack of real-world case-studies is the biggest bottleneck in comparing the effectiveness of different testing approaches [Bozkurt et al., 2013].

3.3 Debugging Definitions

Debugging is an integral part of any software development lifecycle. The main focus of any debugging technique is to detect, localize and possibly correct faults revealed in software testing phase [DeMillo et al., 1996]. There are many well-known debugging approaches such as slicing-based [Korel, 1990], spectrum-based [Abreu et al., 2009], delta debugging [Cleve and Zeller, 2005], and model-based software debugging [Mayer and Stumptner, 2007].

Model-based software debugging derives its roots from model-based diagnosis. *Model-based Diagnosis* is used in diagnosing probable reasons behind the observed fault. The earliest work in the field of diagnosis can be found three decades ago, when Davis et. al. proposed a diagnosis approach for finding faults in digital circuits based on the structure and behavior of the system [Davis, 1984]. In his words: A model is an ”understanding of how a system should work”, which is used to detect the misbehavior of the system [Davis, 1993].

In other words, a model is a representation of the correct behavior of the physical system. A set of observations depicting any malfunctioning of the actual system can be used on the model to explain the cause behind the fault. The possible explanations obtained from the model are known as “diagnoses”. The description of a faulty system can be expressed in various ways such as first-order logic or temporal called first principles.

A general theory of diagnosis from first principles was first proposed by [Reiter, 1987]. He presented a generalized algorithm for diagnosis based on conflicting sets obtained from the contradiction in system descriptions and observations, which was later on corrected by [Greiner et al., 1989]. This algorithm uses the term “conflicts” to denote inconsistent parts of the model given a set of observations, i.e., inputs and expected outputs. And, uses a theorem prover to find the conflicts. There are three main components needed for computing diagnosis, i.e., system components, system descriptions and observations.

There is a common assumption in model-based diagnosis that all information regarding system description and faults is available. However, this might not be true in reality [de Kleer and Williams, 1987]. Most of these

approaches use hitting-set algorithm for computing explanations from conflicts. As the hitting-set algorithm has high space complexity, there have been approaches to compute diagnoses using tree-structured algorithms [Stumptner and Wotawa, 2001].

In the context of software debugging, *program slicing* has been widely discussed in academia in order to locate software bugs [Tip, 1995]. A *Slice*, according to [Weiser, 1982] is an executable subset of a program P, which contains all relevant statements with respect to some slicing criterion. A *slicing criterion* is specified by a set of variables at some location in a program. The purpose is to focus only on statements which actually influence the particular variable(s) at a specific program location. Hence, all other statements become irrelevant in the debugging context, thereby reducing the program size to be analyzed. There are two major types of slicing: static and dynamic. The [Weiser, 1982] approach is a static one, as it considers all possible inputs for the variable of interest v. [Korel and Laski, 1988] suggested an approach, which only considers run-time input given to the program for computing slices of a particular variable v. The added advantage of this approach is further reduction in the slice size, along with precise tracking of dynamic data structures such as arrays [Tip, 1995]. For a detailed survey of different program slicing techniques, we refer the interested reader to the survey done by [Tip, 1995].

The original scope of the model-based diagnosis was finding faults in physical systems. [Wotawa, 2002] showed the relevance of the diagnosis approach in finding faults in software programs. He argued that hitting-sets are equivalent to "slices" used in dynamic slicing for the software debugging. The program statements in a slice would represent components in a dependency-based model. A failing test case would represent the set of observations.

In this work, we use model-based debugging approach using dynamic slicing. The debugging approach presented in [Wotawa et al., 2012] is adapted to service-oriented architecture needs. And the dynamic slicing is considered because of reduced runtime costs.

3.4 Diagnosis of SOA Applications

There have been many efforts in providing a comprehensive diagnostic framework for diagnosis, monitoring and repair activities of SOA processes. The most recent work on that was conducted under a european project called WS-DIAMOND[1]. The main focus of the project had been to build a platform for monitoring, diagnosis and self-healing of web services [Modafferi et al., 2006]. They introduced a plug-in so as to overcome shortcomings of the standard BPEL engine, by augmenting its repair features. [Console et al., 2007] as a part of the DIAMOND project, describes the decentralized architecture for model-based diagnosis. All participating services are assumed to have a local Diagnoser, and to maintain loose-coupling principle, a global Diagnoser is responsible for communication between all participating components. As a part of the project, Ardagna et al. [Ardagna et al., 2007] described a framework for adaptive web service processes. In [Li et al., 2009], the authors present a decentralized diagnosis approach for BPEL processes, [Travé-Massuyès et al., 2006] considered the question of how to define diagnosability of systems and its use for diagnosing web services. Ardissono et al. [Ardissono et al., 2008] discussed the question of enhancing web service compositions using diagnosis.

[Friedrich et al., 2010b] proposed an alternative approach to diagnose processes with partial known behavior and claim to outperform dependency-based methods. They use CLP(FD) constraint solver for obtaining minimal diagnoses from the execution trace of a failed business process. To make up for the partially known behavior of web services, they introduce a set of possible behavior themselves in order to make the dependency-based method work. Our work makes use of a minion constraints solver [Gent et al., 2006], where we add the missing information using table constraints. In a more recent work, [Mayer et al., 2012] computed the complexity of their approach to be second level of the polynomial hierarchy. They argue that dependency-based models are not capable of capturing true semantics of partial models because of the highly dynamic nature of distributed systems. They use discrete-event system models as a foundation for their work, but unlike typical "diagnosis-model" determines the normal or faulty nature of an event

[1] http://wsdiamond.di.unito.it/

from execution. They believe that partial models can be better diagnosed or repaired using answer set programming [van Harmelen et al., 2007] rather than constraint satisfaction.

[Yan et al., 2009] also make use of discrete event systems to model the BPEL processes. However, they employ synchronized automata to represent process trajectories, and diagnose the faulty process using consistency-based systems. They categorize exceptions as either "time-out" (due to network fault or the remote web service) or "business logic", i.e., faults within the workflow activities. They focus in their work on "business logic" faults. We also make use of dynamic slicing [Korel, 1990] combined with diagnosis [Wotawa, 2002] in our diagnostic model. But, in contrast to their work, we use graph-based approach for modeling the service process.

In another work, [Friedrich et al., 2010a] et. al. generate repair plans for failed executed processes. They propose a model-based planning strategy for not only diagnosing faults, but also generating a repair plan for faulty processes. The approach includes both design-time information of the process structure and run-time monitoring. They argue that it is not possible to guess all faults at the design-time. For that, they suggested a heuristic-based reasoner, which determines the effect of "non repairable" activities on "repairability" of process definitions. In addition, they also include the run-time execution history for the "generic repair " plan generation.

Our work is more related to the work done by [Friedrich, 2010], that diagnoses the behavior of faulty service-based processes using logic programming. Their solution first finds diagnoses for the occurred fault, then all diagnoses are given as an input to the repair generator. The repair generator part is responsible for selecting the most suitable repair suggestion as a compensation for the failed or partially executed process. They use disjunctive logic programming (DLV)[Leone et al., 2006] for repair generation. We model the activities with in a business process using constraints.

In contrast to these papers, we describe a solution that combines dynamic slicing with model-based diagnosis for improving the overall diagnosis performance. Hence, it can be considered similar to the approach Wotawa et al. proposed for debugging Java programs [Wotawa, 2002]. We are particularly interested in debugging functional faults in BPEL compositions. The focus

was to build light-weight debugger for diagnosing faults in BPEL partial models.

3.5 Conclusions

In this chapter, we presented related work in the context of model-based testing and debugging of SOA business processes. One of the main issues in testing SOA compositions is the high cost at the integrator side. Model-based testing can greatly reduce the cost by automating the test generation process. Also the coverage achieved by model-based testing tools is much higher than randomly or manually generated tests. Furthermore, formal verification techniques such as model-checking, petri nets and symbolic execution, can be combined with model-based testing for test-case generation and test-coverage analysis. These approaches can greatly reduce the testing costs, as they can be performed offline. The oracle problem can be addressed using contract-based approaches in the testing of SOA processes. Hence, model-based testing and verification of service composition combined with contract-based approaches can better cope with the issue of "Trust" in testing. However, the contracts should be provided by the developer or the provider, so as to reduce the testing cost at the integrator end. Further details about the approach are presented in Chapter 4.

Regarding debugging of service compositions, related work concentrates on the monitoring, diagnosis and repair of web services. We present a light-weight debugging approach which can reduce the debugging cost. The approach takes execution traces of BPEL compositions, and employs a model-based debugging approach to look for probable causes of functional faults. More details about our approach are presented in Chapter 6.

4 Model-based SOA Testing

4.1 Introduction

A typical SOA-based system consists of a multitude of services, business-processes, message busses, registries and service monitors owned by multiple stakeholders [Leitner et al., 2013]. In the context of SOAs reliability, observability and controllability are the two major issues faced by industry as well as the research community [Hierons and Ural, 2009]. According to [Bozkurt et al., 2013], limited observability and controllability raise the issue of *"Trust"*, and testing is one way of building the confidence that the implementation conforms to specifications. However, due to the intricate nature of SOA-based systems, the testing process is not as straightforward as with traditional software systems [Canfora and Penta, 2006]. In a traditional testing process, the system under test is generally owned by one stakeholder, and the implementation is also available, but in case of SOAs it is not the case, rather the system under test is composed of many loosely coupled web services owned by many stakeholders.

To illustrate the problem, let us consider the Bank Loan example depicted in Figure 4.1, taken from [Bpe, 2012]. The corresponding process starts upon receiving a loan request from a client as follows: Loan requests below 10.000 credits are sent to the corresponding BPEL service *calculateRisk*. This service computes risk related to a particular client based on information like the clientID and the loan amount. The requests from low risk clients are approved immediately. Those from high risk clients or with amounts starting at 10.000 credits are sent to another service *thoroughAssessment*. This service is responsible for thorough assessment before a decision is made.

The problem lies in the fact that implementation details about these two essential services are unavailable, as a result the effect of calling web ser-

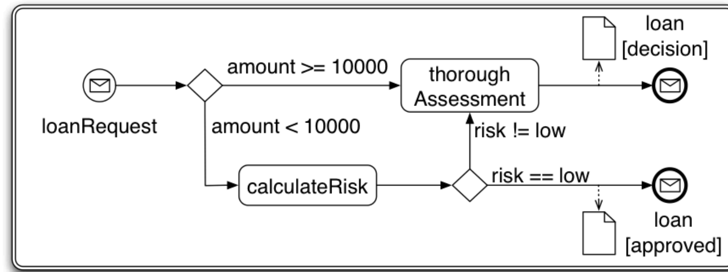


Figure 4.1: The Bank Loan BPEL Process

vices *calculateRisk* and *thoroughAssessment* is unknown. This is part of the information we aim to attach to the model via pre- and postcondition contracts. For our example, we assume `calculateRisk` to suggest a low risk for clients with excellent bank records, or whenever the amount is below 1000 credits. This can be easily captured in pre- and postconditions (see Section 4.4) attached to the corresponding process component.

The Chapter is organized as follows: In Section 4.2, we present overall architecture of our approach. Section 4.3 describes underlying definitions used in our approach. This is followed by the test case generation in Section 4.4. Section 4.5 explains the experimental setup and results obtained from first experiments are discussed in Section 4.6. We compare structured and random testing of sequential and concurrent BPEL processes in Sections 4.7, and 4.8. Section 4.9 presents conclusions.

The content of this Chapter has been published in the following papers.

- Fifty shades of grey in SOA testing [Wotawa et al., 2013].
- SOA grey box testing- a constraint-based approach [Jehan et al., 2013b].
- Functional SOA testing based on constraints [Jehan et al., 2013a].
- BPEL Integration Testing [Jehan et al., 2015].
- SOA testing via random paths in BPEL models [Jehan et al., 2014].

4.2 Architecture

Model-based testing (MBT) is a popular approach for testing complex systems, as it helps structuring the test design process and building common understanding among different stakeholders [Binder et al., 2015]. In practise "Model-based testing encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases [Utting et al., 2012]". According to this definition, MBT is a three step process: model generation, test generation and test execution.

Our model-based testing architecture comprising BPELTester along with links to external tools is illustrated in Figure 4.2. It is composed of three modules: *Analysis module*, *Test Suite generator* and *Test executor*. The Analysis module is responsible for building the model. It takes the SOA definitions comprising BPEL and WSDL files and converts them into a control flow graph representation that might be annotated using pre- and post conditions of service invocations. Each node in a control flow graph represents an activity in the BPEL process.

The Test Suite Generator module is responsible for test generation. It derives abstract test cases (paths) that represent particular traversals through the control flow graph. Each such traversal represents a particular execution of the SOA process model. In order to compute corresponding test cases, paths in the control flow graph are translated into constraint representation. Our constraint representation makes use of static single assignment form as explained in [Nica, 2010]. Afterwords, this constraint representation of paths is used as input to the MINION constraint solver [Gent et al., 2006], which computes all the necessary inputs and expected outputs that characterize such an execution.

The test execution is done by the Test2Unit module. The module derives the appropriate input for the *BPELUnit* [Lübke, 2006] tool that is used for actually executing generated tests. This tool plays an important role in the results analysis as this module is also responsible for generating the test suite for the *MuBPEL* tool that is a mutation testing tool for BPEL language. We rely on this tool to measure the quality of the generated test suite by

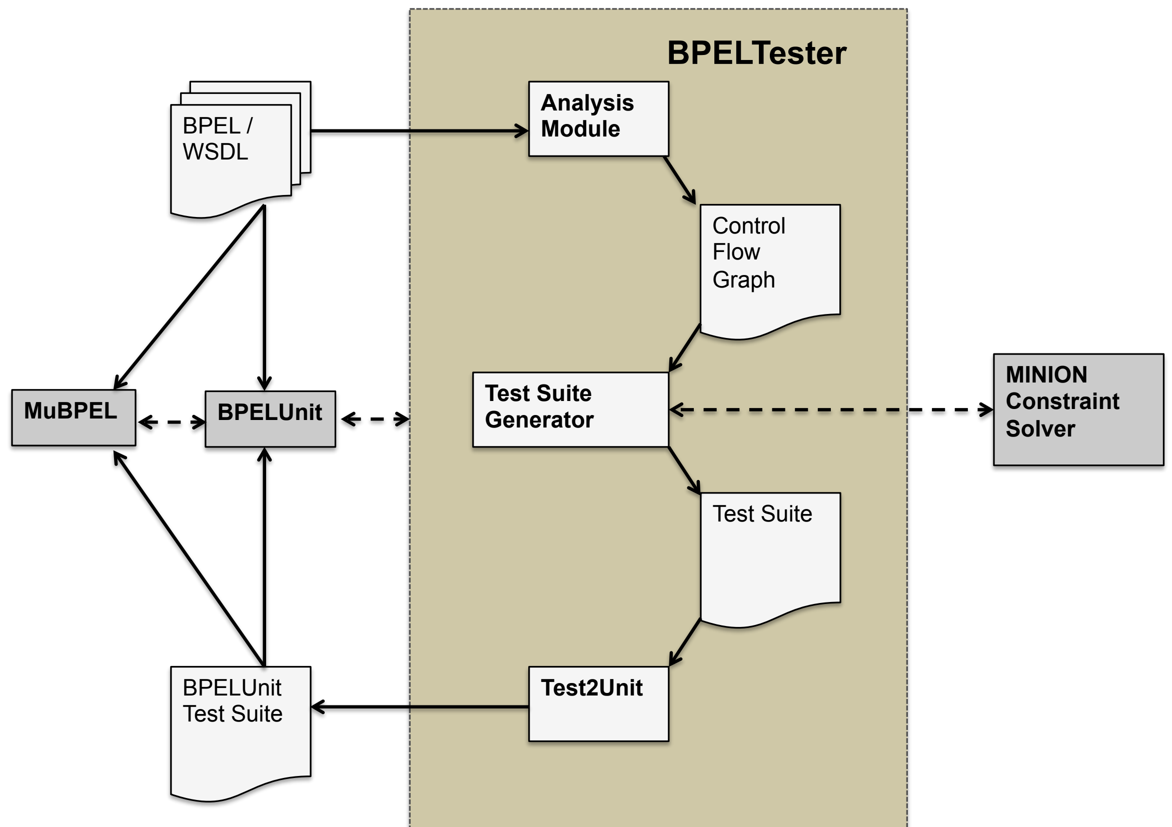


Figure 4.2: The BPELTesterarchitecture.

checking if it can distinguish between a mutant and the original program. The quality is measured through the mutation score of the generated test suite. For further details about the actual implementation and the working of the mutation tool, we refer the interested reader to the tool website [BPEL Mutation tool, 2011].

4.3 Definitions

As mentioned in the previous section, the first part in our test case generation method is the extraction of a control flow graph from the BPEL source code. The control flow graph serve as our reasoning model.

Definition 1 (BPEL Flow Graph). *A BPEL Flow Graph G is a tuple* $(V, E, v_s, F, \Gamma_A, \Gamma_C)$*, where vertices* $v \in V$ *represent BPEL process activities, E is the set of edges* $e = (v_i \in V, v_j \in V)$ *which correspond to the connections between BPEL activities,* $v_s \in V$ *is the start vertex,* $F \subseteq V$ *is the set of graph G's leaf vertices, and* Γ_A *as well as* Γ_C *are functions that map vertices to activity assignments and conditions respectively.*

A BPEL Flow Graph captures all the possible executions between the defined vertices in a given process. In our case, a vertex v stores all the information relevant to a corresponding BPEL activity. And a use-case consists of a particular sequence of vertices from the start vertex v_s to the end vertex v_F. A path π is formally defined as follows.

Definition 2 (Path). *Given a BPEL Flow Graph G, a sequence of vertices* $\pi = v_1 v_2 \dots v_n$ *is a path in G iff (1) for all* $i \in \{1, \dots, n\}$ *we have* $v_i \in V$*, (2) the sequence starts with the start vertex* v_s*, i.e.,* $v_1 = v_s$*, (3)* v_n *is a leaf vertex, i.e.,* $v_n \in F$*, and (4) for all* $i \in \{1, \dots, n-1\}$ *we have* $(v_i, v_{i+1}) \in E$ *.*

A user has to select the start and the end vertex in order to extract corresponding paths between the selected vertices. Once we have all the possible paths between the selected vertices, we need to make sure, if such a path is feasible or not [King, 1976]. For that purpose, we need to define a path condition $c(\pi)$ that comprises π's vertices' assignments and conditions.

Definition 3 (Path Condition). *Given a path* $\pi = v_1 \dots v_n$ *of some BPEL Flow Graph G, path* π*'s path condition* $c(\pi)$ *is a sequence of assignments and conditionals of* π*'s vertices defined as* $c(\pi) = \Gamma_A(v_1) \cup \Gamma_C(v_1) \dots \Gamma_A(v_n) \cup \Gamma_C(v_n)$*.*

Variables in a path condition are replaced by indexed variables in order to ensure static single assignment form (SSA) [Brandis and Mössenböck, 1994].

A static single assignment form ensures that every variable is defined only once in a program. This means whenever a variable is defined, i.e., occurs at the right side of an assignment, the corresponding index is incremented. This makes the translation of the path and its path condition $c(\pi)$ to a constraint representation much easier. Depending on the satisfiability of a path condition, we can define a feasible path formally as follows.

Definition 4 (Feasible Path)**.** *A path π in a BPEL Flow Graph G is feasible if its path condition $c(\pi)$ is satisfiable.*

The above stated definitions were used in the test suite generation of sequential BPEL programs. However, we needed to adapt these definitions in order to take care of concurrency issues between various parallel branches for concurrent BPEL processes. Such flow constructs allow for possibly concurrent branches that are activated by individual and optional guards. If such guards are not specified, we assume them to be *True* for simplifying our description.

An extended BPEL flow graph allow us to encompass both the sequence and flow structures, whereby the partial knowledge about the vertices V can be specified using $\gamma_C(v \in V)$ and $\gamma_A(v \in V)$.

Definition 5 (An Extended BPEL Flow Graph)**.** *An* Extended BPEL Flow Graph *G is a tuple $(V, B, E, v_0, F, \gamma_C(v \in V), \gamma_A(v \in V), \gamma_P(v \in V), \gamma_G(v \in B), \gamma_B(v \in V \setminus B))$, where V is a finite set of vertices representing BPEL process activities, $B \subset V$ is the finite set of fork activity vertices (where a run might branch), $E \subseteq V \times V$ is a finite set of directed edges representing the connections between BPEL activities (edge $e = (v_1, v_2) \in E$ connects v_1 to v_2), $v_0 \in V$ is the start vertex, $F \subseteq V$ is the set of leaf vertices (with no outgoing edges), and the functions $\gamma_C(v)$ and $\gamma_A(v)$ map vertices $v \in V$ to activity conditions and assignments respectively. If v is in B, $\gamma_P(v \in V)$ returns the complementing join activity vertice (and vice versa), and $\perp$ otherwise. Function $\gamma_G(v \in B)$ returns a list of tuples (e_i, TG_{e_i}) for all of a fork vertice v's outgoing edges e_i and their transition guards TG_{e_i} (if there is no guard specified, we assume True so that this branch is always enabled). For any vertice v in $V \setminus B$, the function $\gamma_B(v \in V \setminus B)$ returns the closest predecessor in B if there is such a node, and $\perp$ otherwise.*

If there are no concurrent computations, an actual execution follows a path in the flow graph as of Def. 2 in order to derive corresponding test cases

by searching for a satisfying variable assignment to the conditions and assignments encountered along a path.

Definition 6 (Finite Path). *A finite path π of length n in an Extended BPEL flow graph G as of Def. 5 is a finite sequence $\pi = \pi_1\pi_2...\pi_n$ such that (1) for any $0 < i \leq n$: $\pi_i \in V$, (2) $\pi_1 = v_0$, (3) for any $0 < i < n$, the edge $e = (\pi_i, \pi_{i+1})$ is in E, and (4) $\pi_n \in F$. $|\pi|$ denotes the length of a path π. We use $f(\pi)$ to refer to the last vertex in π.*

Definition 7 (Finite Path Segment). *A finite path segment π in an Extended BPEL flow graph G is defined like a path, but does not have to start in G's initial state v_0, and neither is $f(\pi)$ required to be in F of G.*

Unlike sequence activity, the flow construct allows parallel branches to be active simultaneously. Therefore, we introduce the following definition of a run.

Definition 8 (Finite Run). *A finite* run *r of length n in an Extended BPEL Flow Graph G as of Def. 5 is a finite sequence $r = r_1r_2...r_n$ such that (1) for any $0 < i \leq n$: $r_i \in V$, (2) $r_1 = v_0$, (3) $r_n \in F$, and (4) for any $0 < i < n$, either the edge $e = (r_i, r_{i+1})$ is in E, or if $\gamma_B(p_i) \neq \perp$ then there has to be some $i < j \leq n$ such that (a) there is no $i < k < j$ with $r_k = \gamma_P(\gamma_B(r_i))$ and (b) edge $e = (r_i, r_j)$ is in E. $|r|$ denotes the length of run r. With $f(r)$ we refer to the last vertex in r.*

A finite run in a flow activity can be considered analogous to the notion of a path in definition 2 for a sequence activity. Since the order of parallel branches is not defined, a finite run only ensures the partial order in a single branch. Like a path (see definition 2) in a sequence activity, a run r might also be infeasible. Therefore, we also need to check if a run's collected assignments and conditions are possible.

Definition 9 (Feasible Run). *A* feasible *run r is a run as of Def. 8 s.t. the conditions and assignments encountered along the run are feasible. It is* complete, *iff for all satisfied transition guards TG_{e_i} at all $v \in B$ visited by r, the corresponding branch started by edge e_i is present in r.*

A corresponding satisfying assignment for a complete run r defines a valid test case. In a sequence activity, test cases can be generated from any feasible

path 4 in a flow graph. Likewise, in a flow activity, we derive test cases from feasible paths π in G, but in order to compute π's constraints, we model all the branches within a flow activity, where only the ones belonging to a path π need to be active.

Definition 10 (Run Constraints). *For a path $\pi = \pi_1\pi_2...\pi_n$ in some Extended BPEL flow graph G as of Def. 5, we create the* run-constraints $C(\pi)$ *as follows. For each $l \in \gamma_G(\pi_i)$ of a $\pi_i \in B$, we define a branching variable b_l. Let scope be an initially empty list of these branching variables, where we can append a variable b_l via $append(scope, b_l)$, and ask for the last variable with $b_l = last(scope)$ (which will be $\perp$ if the list is empty) as well as remove the last variable via $drop(scope)$. Furthermore, let stop be an initially empty list of vertices in G which we can access with the same functions as scope. Then let $C(\pi)$ be the union of the constraints as derived by traversing π from π_1 to $f(\pi)$ (possibly recursively) as of Def. 11, where in recursive calls the original path can be referred to as π^0, and where variables are replaced by indexed variables in order to implement a static single assignment form (see [Brandis and Mössenböck, 1994]).*

In order to compute run constraints for a $flow$ activity, we need to take care of three things: First, the local scope of variables defined in each branch should be maintained. Second, if the current branch is not a part of the run, we need to propagate values of all variables defined in that branch, so that their SSA representation before the fork activity should match the representation after the join activity. This is due to the fact, that we model all branches in a $flow$ activity, irrespective of the fact if that branch is active or not. Third, in case the current branch is active in a particular run, then we need to update corresponding conditions and assignments in that branch stored in Π.

Definition 11 (Run Scope). *For a given path segment π in G, its branching variables and lists scope and stop, we do the following: Let Π be an initially empty list of tuples (v, b_m, π') such that v is a vertex, b_m is a branching variable, and π' is a path segment in G. Then, traversing π from π_1 to $f(\pi)$ do as follows.*

1. *if $\pi_i = last(stop)$, then for each (π_i, b_m, π') in Π do: First, remove (π_i, b_m, π') from Π, and then add constraints for π' as of this Definition for a local scope having b_m as it sole element, computing the local branching variables for π', and assuming a local empty stop list. When there is no more (π_i, b_m, π') in Π, call $drop(scope)$ and $drop(stop)$.*

2. *if* $\pi_i \notin B$ *then (a) add constraints* $\gamma_C(\pi_i) \cup \gamma_A(\pi_i)$ *if* $last(scope) = \perp$ *and proceed with Step 1 for* π_{i+1}*, or (b) add constraints* $(b_l \rightarrow \gamma_C(\pi_i)) \cup (b_l \rightarrow \gamma_A(\pi_i)) \cup (\neg b_l \rightarrow \gamma'_A(\pi_i))$ *for* $b_l = last(scope) \neq \perp$ *and* $\gamma'_A(\pi_i)$ *replacing every assignment of a variable in* $\gamma_A(\pi_i)$ *with an assignment of the variable's old value (so that we are always synchronized in respect of the SSA indices when arriving at the join activity, regardless of which branch was active).*
3. *if* $\pi_i \in B$ *then do as follows. For* $l = ((\pi_i, \pi_{i+1}), TG_l) \in \gamma_G(\pi_i)$*, add the constraints* $b_l \rightarrow TG_l$ *and* $TG_l \rightarrow b_l$*, and append* b_l *to scope, append* $\gamma_P(\pi_i)$ *to stop, but add the constraint* b_l *only if* $\pi = \pi^0$*. Then find for each* $m = ((\pi_i, v), TG_m) \in \gamma_G(\pi_i)$ *s.t.* $m \neq l$ *a path segment* π' *leading from* v *to* $\gamma_P(\pi_i)$*, and add the tuple* $(\gamma_P(\pi_i), b_m, \pi'')$ *s.t.* π'' *equals* π' *but with the last vertex* $(\gamma_P(\pi_i))$ *removed to* Π*, as well as add constraints* $b_m \rightarrow TG_m$ *and* $TG_m \rightarrow b_m$*.*

Once the run constraints for a path π are collected, we can check the feasibility of all the run constraints using the constraint solver [Gent et al., 2006], and store all the feasible branches as test cases. MINION is an out of the box, open source constraint solver, whose syntax requires a little more effort on modeling the constraints, e.g., it does not support different operators to be used within one constraint. Because of that, more than one constraint may be needed to model certain operators like addition and subtraction.

The test suite generation of a $flow$ activity is different from that of a *sequence* activity in a way that we need to synchronize the assignments and conditionals of all the concurrent branches in a run r.

Definition 12 (Test Case and Test Suite)**.** *A test case for a BPEL Flow Graph G is a variable assignment that makes a complete run r (or a path p) in G feasible. A test suiteTS is a set of test cases.*

4.4 Test Case Generation using constraints

The test case generation for BPEL processes has been investigated thoroughly. Most of the work on test case generation of BPEL compositions use formal verification tools such as petri-net or model checkers to derive

```
1 <sequence standard-attributes>
2     standard-elements
3     activity+
4 </sequence>
```

Figure 4.3: A Sequence activity

test cases [Bozkurt et al., 2013]. Exploiting constraints for testing has already been considered in the literature. Gotlieb and colleagues [Gotlieb et al., 1998] presented an approach for extracting test cases from programs using a constraint representation of source code. Our work is similar, as we also rely on constraints for automated test data generation. However, the application domains are different, and the extraction of constraints has to take care of partial specifications.

A BPEL process can be abstract or executable: An abstract process is partially specified and is not intended to be executed; an executable process, on the contrary, has to be fully specified. An executable process is composed of basic and structured activities. Among the structured activities, two very common structures are the sequence activity and flow activity. In a *Sequence* activity, all encompassing activities have to be performed sequentially, whereas a *Flow* activity is defined to execute more than one activity in parallel. Because of that, our test-case generation process for sequential BPEL processes is different from that of concurrent BPEL processes. Another important activity is *Invoke* activity, which is meant for communication of the the business process with its partner processes or web services. The communication could be either one-way (asynchronous) or a two-way (synchronous) process. In Sections 4.4.1, 4.4.2, we outline our constraint-based solution to derive functional tests from the control flow graph representation of an executable synchronous BPEL process.

4.4.1 Sequence Structure

A Sequence activity is a structured activity which contains other basic and structured activities as shown in Figure 4.3. The activities need to be

```
1:  procedure AllPaths(G, MaxLen, v_s)
2:      initialize test suite S ← ∅
3:      compute the set P← AllPathsSUB(G, MaxLen,π,P,v_s)
4:      for each path π ∈ P do
5:          Compute the path constraints C(π)
6:          check the satisfiability of path constraints C(π)
7:          if C(π) is satisfiable then
8:              add a satisfying assignment (a test case) to S
9:          end if
10:     end for
11:     return test suite S.
12: end procedure
```

Figure 4.4: TCG algorithm that considers all paths.

executed in a strictly defined manner. A sequence activity can contain nested sequence activities, where $+$ symbol denotes one or more activities.

Figure 4.4 comprises our test case generation algorithm used only for sequential BPEL processes. The algorithm takes the BPEL Flow Graph G and the maximum path length *MaxLen* as inputs and computes feasible test cases. Note that *MaxLen* has to be equal or larger than the length of the smallest path in G from the start to an end vertex. The algorithm is search-based and traverses the flow graph using a depth-first search strategy for extracting paths. Afterwards, path conditions are computed and checked for consistency. If the path condition is feasible, the variable assignments that result from such a check are saved as a test case. Finally, the algorithm returns a test suite.

In the following, we discuss the different steps of the algorithm and those activities that have to be carried out in more detail. The approach is similar to symbolic execution, already discussed in Section 3.2.1. Similar to symbolic execution, conditions are computed which belong to a particular execution path. In our case, we convert each path condition into a constraint satisfaction problem (CSP). The conversion takes place in two steps: first, the BPEL flow graph constructs are converted to an intermediate representation, called static single assignment (SSA). The detailed explanation of SSA representation of Java programs into MINION can be found in [Nica, 2010].

```
1:  procedure ALLPATHSSUB(G, MaxLen, π, P, v_curr)
2:      if |π| ≥ MaxLen then
3:          return
4:      end if
5:      append v_curr to path π
6:      if v_curr matches v_F  then
7:          add path π to set P
8:          return
9:      end if
10:     for each OutEdge of v_curr in G  do
11:         v_curr ← v_outEdge
12:         call AllPathsSUB(G, MaxLen, π, P, v_curr)
13:     end for
14: end procedure
```

Figure 4.5: AllPathsSUB algorithm for computing all paths for a Flow Graph G up to a given pre-defined length MaxLen.

Since BPEL is a mixture of a workflow and a programing language. Therefore, the programming language constructs can be handled much like the [Nica, 2010] approach with few subtle differences. For example, we do not do the loop unrolling, rather the loop execution is defined by the path length provided by the tester. Our conversion algorithm works as follows.

- SSA conversion: The static single assignment (SSA) form is an intermediate representation of a program such that no two left-side variables share the same name. This intermediate representation enables an easier conversion into a CSP. The basic rules used for the conversion of a BPEL path into its SSA representation are listed below:
  - We convert an *Assign* activity by adding an index to a *To* variable each time the variable is defined, i.e., declared as the *To* variable. If a variable is redefined, the index is incremented so as to satisfy the SSA property. The index of a *From* variable, i.e., referenced variable is equal to the last definition of the variable.
  - We convert *Receive* and *Reply* activities into assignments.
  - *Invoke* is easily converted into assignments, where the right hand side variable is the "input variable" and the left side variable is

the "output variable".
  - We convert the structured activity *If* in two steps: 1) the condition is saved in an auxiliary variable. 2) each assign or invoke activity is converted according to the above rule. 3) The condition variable is set to true in case of an "if " or "else if" branch, but to false in case of an "else" branch.
  - The *while* structure is converted similarly to *If*, with the exception that the condition is always set to true. The loop is repeated up till the maximum length specified by the tester.
  - The *Flow* activity construct is modeled different from that of other constraints. First, the transition guards are converted into *If* structure. Since there can be more than one branch in a flow construct. The algorithm makes sure that each branch is active once, e.g., the condition variable is set to true, in order to model the concurrent behavior into constraints. However, note that, our implementation assumes that parallel branches do not have shared variables. Under this assumption, the concurrent behavior can be modeled much like a sequence activity with the exception that all branches have to be modeled for each run.

- Constraint conversion: The second step involves the conversion of SSA statements into their corresponding constraints. The representation of the conditions and assignments resulting from the SSA conversion as constraints is simple and requires basically nothing else than a direct mapping from variables to constraint variables and from the conditions to assignments to their respective representation. In order to illustrate the constraint conversion we show the constraint representation for the path from the Bank Loan example using MINION constraints. The constraint `ineq(x, y, k)` ensures that $x \leq y + k$, and `eq` states that both variables used as parameters have to have the same value.

The conversion steps outlined above can be best understood with the help of the loan example introduced in the start. The flow graph of the loan example is given in Figure 4.12.

For this example there are three possible paths: (1) `loanRequest`, `amount >= 10000`, `thoroughAssessment`, `loan[decision]`, (2)`loanRequest`, `amount < 10000`,

calculateRisk, risk == low, loan[approve], and (3) loanRequest, amount < 10000, calculateRisk, risk != low, thoroughAssessment, loan[decision].

Let us consider path (2), making the following assumptions regarding the BPEL components' behavior: Component loanRequest has an empty pre-condition and $amount > 0$ as post-condition. Component calculateRisk's behavior is given only partially: up to 1000 credits, the risk is assumed to be low. This partial specification can be formalized using the post-condition $amount < 1000 \rightarrow risk == low$. For calculateRisk the pre-condition is assumed to be empty. Taking into consideration the pre- and post-condition as well as the conditions related to other BPEL components, we obtain the following conditions for path (2):

```
1: amount_0 > 0
2: amount_1 = amount_0
3: amount_1 < 10000
4: amount_1 < 1000 → risk_0 == low
5: risk_0 == low
6: loan_0 == approved
```

We use MINION to check for a variable evaluation that satisfies all constraints. For path (2), the assignment amount_0=1, amount_1=1, risk_0=low, loan_0=approved is such an evaluation. Obviously, amount_0 $= 1$ with the expected output loan_0 $=$ approved is a valid test case and exactly ensures executing its corresponding path.

The flow graph for our running example is quite similar to the graphical representation of the original process in Figure 4.1. The vertices for this process are defined by the following activities: *receiveInput* (v_{rI}), *AssignLoan* (v_{AL}), *IfLoan* (v_{IL}) and (v_{Else_IL}), *AssignInRisk* (v_{AIR}), *InvokeRiskService* (v_{IRS}), *AssignOutRisk* (v_{AOR}), *IfLowRisk* (v_{ILR}) and (v_{Else_ILR}), *AssignApproved* (v_{AA}), *AssignInAssess* (v_{AIA}), *InvokeAssessRisk* (v_{IAR}), *AssignOutAssess* (v_{AOA}), and *replyOutput* (v_{rO}).

We store all data related to a particular activity with the corresponding vertex, i.e., input variables, output variables, assignments, as well as conditions for structured activities *if*, *else if*, and *while*. Essential are also the pre-

and post conditions related to an activity. The conditions for vertex v_{rI} (the *receiveInput* activity) are defined, for example, as follows:

```
Pre-condition:  $input_loan > 0
Post-condition: $loan = $input_loan
```

While due to the pre-condition only loan requests with positive amounts are allowed, the post condition ensures that the local variable "loan" is assigned the actual loan amount. For these pre- and post conditions we use the same language as is used for BPEL expressions, i.e., XPATH [Xpa, 2011], where we currently support the usual Boolean operations. In addition, auxiliary information about Invoke activities is added in the form of pre- and post conditions. That is, for the `calculateRisk` service, the expected behavior is defined via the (partial) postcondition $(loan \geq 1000) \vee (risk == 0)$.

In our running example a low-risk low-amount request is treated via the following path (Path 1): $\pi = v_{rI} v_{AL} v_{IL} v_{AIR} v_{IRS} v_{AOR} v_{ILR} v_{AA} v_{rO}$.

We will use such paths to derive our test cases. That is, as a first step, a search-based algorithm traverses the graph in a depth-first search manner in order to extract the paths from v_s to the leaf vertices. For our running example there are three such possible paths, i.e, the first for low-amount low-risk requests leading to an immediate response approving the request, the second and third paths requiring a more thorough assessment.

For the three paths in our exemplary business process, the corresponding path conditions are given in Figures 4.6, 4.7, and 4.8 respectively.

A corresponding variable assignment satisfying $c(\pi)$ for Path 3, and thus a test case, is the following one:

```
input = 10001
loan = 10001
AssessRiskPLRequest = 10001
output = AssessRiskPLResponse
```

1 $v_{rI} : input_0 > 0$
2 $v_{AL} : loan_1 = input_0$
3 $v_{IL} : loan_1 < 10000$
4 $v_{AIR} : CalculateRiskPLRequest_1 = clientID_0$
5 $v_{IRS} : loan_1 < 1000$
6 $v_{AOR} : risk_1 = CalculateRiskPLResponse_0$
7 $v_{ILR} : risk_2 = 0$
8 $v_{AA} : approvalResult_1 = 1$
9 $v_{rO} : output_1 = approvalResult_1$

Figure 4.6: Path 1: $c(\pi)$ for low-risk and low amount loan requests.

1 $v_{rI} : input_0 > 0$
2 $v_{AL} : loan_1 = input_0$
3 $v_{IL} : loan_1 < 10000$
4 $v_{AIR} : CalculateRiskPLRequest_1 = clientID_0$
5 $v_{IRS} : loan_1 < 1000$
6 $v_{AOR} : risk_1 = CalculateRiskPLResponse_0$
7 $v_{Else_ILR} :!(risk_2 = 0)$
8 $v_{AIA} : AssessRiskPLRequest_1 = loan_1$
9 $v_{IAR} : loan_1 \geq 1000$
10 $v_{AOA} : AssessRiskPLResponse_1 = 1$
11 $v_{rO} : output_1 = AssessRiskPLResponse_1$

Figure 4.7: Path 2: $c(\pi)$ for high-risk and low amount loan requests.

1 $v_{rI} : input_0 > 0$
2 $v_{AL} : loan_1 = input_0$
3 $v_{Else_IL} :!(loan_1 < 10000)$
4 $v_{AIA} : AssessRiskPLRequest_1 = loan_1$
5 $v_{AOA} : AssessRiskPLResponse_1 = 1$
6 $v_{rO} : output_1 = AssessRiskPLResponse_1$

Figure 4.8: Path 3: $c(\pi)$ for high amount loan requests.

```
1  ineq(0, input_0, -1)
2  eq(loan_1, input_0)
3  ineq(loan_1, 10000, -1)
4  eq(CalculateRiskPLRequest_1, clientID_0)
5  ineq(loan_1, 1000, -1)
6  eq(risk_1, CalculateRiskPLResponse_0)
7  eq(risk_2, 0)
8  eq(approvalResult_1, 1)
9  eq(output_1, approvalResult_1)
```

Figure 4.9: MINION constraints for Path 1.

```
1  ineq(0, input_0, -1)
2  eq(loan_1, input_0)
3  ineq(loan_1, 10000, -1)
4  eq(CalculateRiskPLRequest_1, clientID_0)
5  ineq(loan_1, 1000, -1)
6  eq(risk_1, CalculateRiskPLResponse_0)
7  diseq(risk_2, 0)
8  eq(AssessRiskPLRequest_1, loan_1)
9  ineq(1000, loan_1, -1)
10 eq(AssessRiskPLResponse_1, 1)
11 eq(output_1, AssessRiskPLResponse_1)
```

Figure 4.10: MINION constraints for Path 2.

```
1  ineq(0, input_0, -1)
2  eq(loan_1, input_0)
3  ineq(10000, loan_1, 0)
4  eq(AssessRiskPLRequest_1, loan_1)
5  eq(AssessRiskPLResponse_1, 1)
6  eq(output_1, AssessRiskPLResponse_1)
```

Figure 4.11: MINION constraints for Path 3.

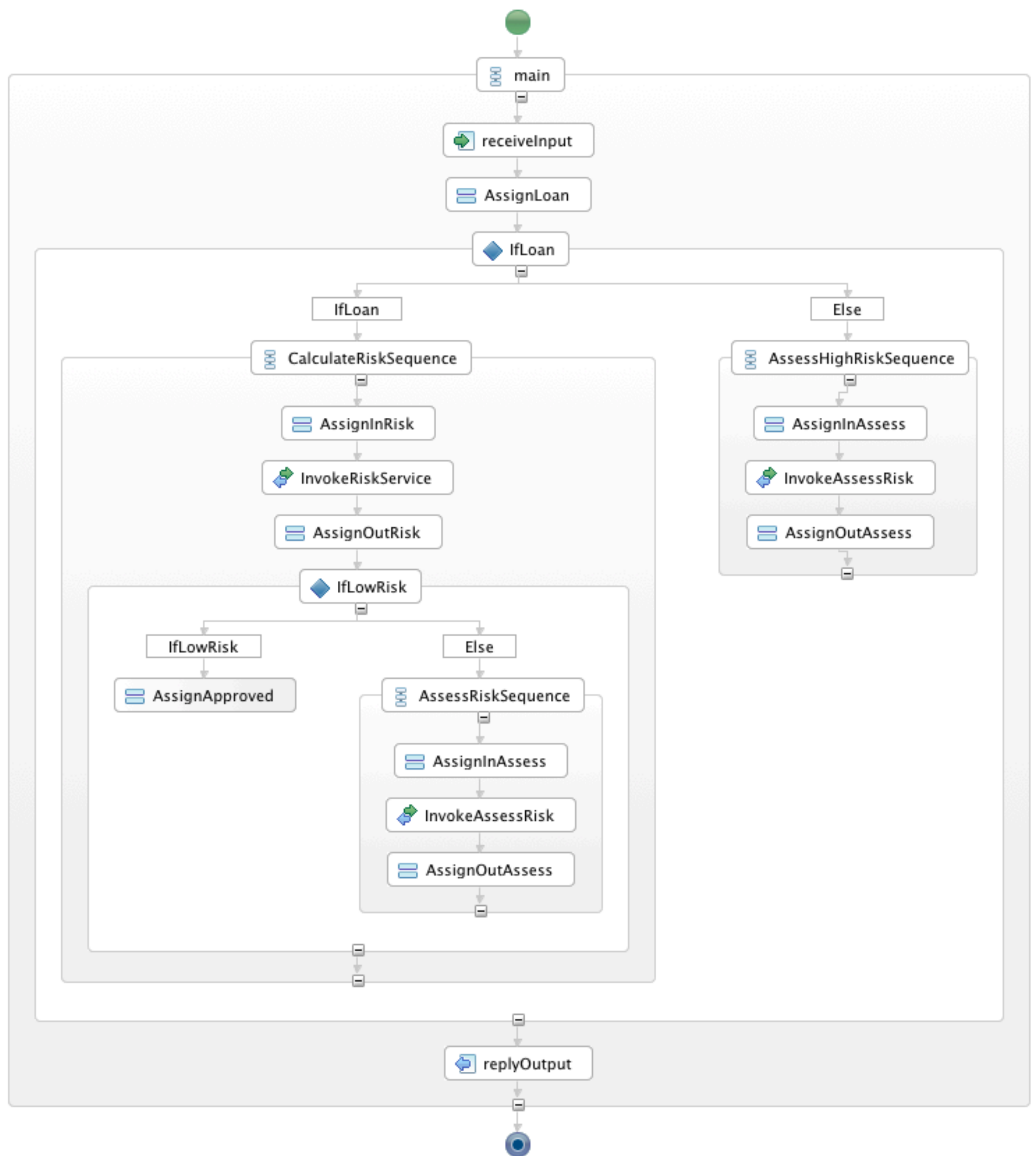


Figure 4.12: Technical view of the Bank Loan Business Process.

Analysis

Lemma 1 (ComputingAllpaths)**.** *The set P in Algorithm 4.4 contains all paths* π *s.t.* $|\pi| \leq MaxLen$*.*

Proof. In Line 3 of Algorithm 4.4 we call the AllPathsSUB algorithm given in Figure 4.5. This algorithm only returns sets of $size <= MaxLen$. Hence, any path in P can never exceed MaxLen and the lemma must hold.

□

Lemma 2 (CheckSatisfiability)**.** *Each path* π *included in a test suite S is a feasible path in a flow graph G.*

Proof. For lemma 2 we can argue that only paths where the feasibility check has been done can pass (lines 4-10). Hence, the lemma has to hold. □

Lemma 3 (Valid Solution)**.** *A test in a test suite S created via algorithm AllPaths for some BPEL flow graph G is a satisfiable variable assignment of a feasible path in G.*

Proof. Line 6 and 7 in Algorithm 4.4 make sure that only satisfiable assignments are test cases.

□

Theorem 4 (AllPathsSoundness)**.** *The AllPaths algorithm is sound such that only feasible paths of length less than MaxLen are added to the test suite S.*

Proof. The proof follows from lemma 1 to 3 directly. We compute all paths. They are feasible and lead to test cases as of definition 5. □

Theorem 5 (AllPathsCompleteness)**.** *The AllPaths algorithm 4.4 is complete. That is, no feasible path is excluded from the result.*

Proof. Similarly using Lemma 1, it can be seen that step 3 of the algorithm performs exhaustive depth-first search to find all paths. Since some of paths might be infeasible, based on Lemma 2, step 5 ensures that all feasible paths are included in the result. Hence, there is no missing path. Figure 4.5 illustrates the AllPathsSUB algorithm, where we take the last vertex of the current path π and recursively call the algorithm on a copy of this path by adding a successor vertex at the end. □

Regarding the termination of *AllPaths* algorithm shown in Figure 4.4, we can see that step 3 in the algorithm makes sure that all paths up till $MaxLen$ are generated, where the $MaxLen$ is larger than the shortest path in a flow graph G. Due to depth-first search and the fact that search is bounded by $MaxLen$, it can be argued that the search terminates in finite time. For loop in step 4 checks the feasibility of all paths included in a set P, which is bounded by $MaxLen$ parameter, hence is finite and terminates. Step 5 concerns whether path constraints are satisfiable or not. This involves converting a path π into corresponding SSA representation π_{SSA}, and translating π_{SSA} into constraint representation π_{CO}. Both of these steps terminate because of finite path length. The last part is about calling constraint solver to find a solution for π_{CO} of a path π. If there is no solution possible, loop will check for the feasibility of the next path. The termination of the for loop is dependent on the termination of constraint solver call. In the end, all feasible paths are added in a test suite S.

The complexity of the algorithm can be computed by summing up the complexities of steps 3 to 6. Let us assume that there are $|V|$ number of vertices and $|E|$ number of edges in a flow graph G, then the size of flow graph becomes $|G| = |V| + |E|$. Step 3 computes all paths up till certain $MaxLen$ which in worst-case can lead to exponential paths when $MaxLen$ is larger than the longest path in a flow graph G. Step 5 in the algorithm can be performed in polynomial time since it involves simple conversion from conditions and assignments (path conditions) to SSA form, and later from SSA form to Constraints representation. Step 6 involves call to constraint solver for finding a solution. Therefore the overall complexity would be $O(|V|^{MaxLen}) + O(|C(\pi)|) + ConstraintSolver(M)$, where $O(|V|^{MaxLen})$ represents an upper bound of the set P, $O(|C(\pi)|)$ denotes the number of path conditions for all paths included in the set P, and $ConstraintSolver(M)$

```
1 <flow standard-attributes>
2     standard-elements
3     <links>?
4         <link name="NCName"/>+
5     </links>
6     activity+
7 </flow>
```

Figure 4.13: Flow activity

denotes the time required to solve the MINION model M comprising of variables, their domains and corresponding constraints(VAR, DOM, CON).

The test case generation algorithm as illustrated in Figure 4.4 can model only sequential processes. Therefore in order to cater for Flow structure, we adapted the modeling process as explained in the next Section 4.4.2.

4.4.2 Flow Structure

According to the BPEL specification document, a *Flow* activity is meant to execute more than one activity in parallel as shown in Figure 4.13. In order to define control dependencies among child activities, *Links* construct can be used. Each such *Link* can be activated when the corresponding *transition condition* is active. If no condition is specified, that *link* is assumed to be *true*.

The semantics of a BPEL Flow activity are depicted in Figure 4.14. The flow activity defines two branches that are triggered if their respective guards ($x < 10$ and $y < 10$) are activated. Both variables x and y are assigned new values in each branch, which are reused after join activity. An execution then follows a *run* in the graph, where, in contrast to a path as we have been using in our earlier work for sequential programs [Jehan et al., 2014, Wotawa et al., 2013], more than one branch may be active simultaneously.

Deriving a flow graph from the BPEL process, and annotating it with our partial knowledge about called web services (and other available knowledge)

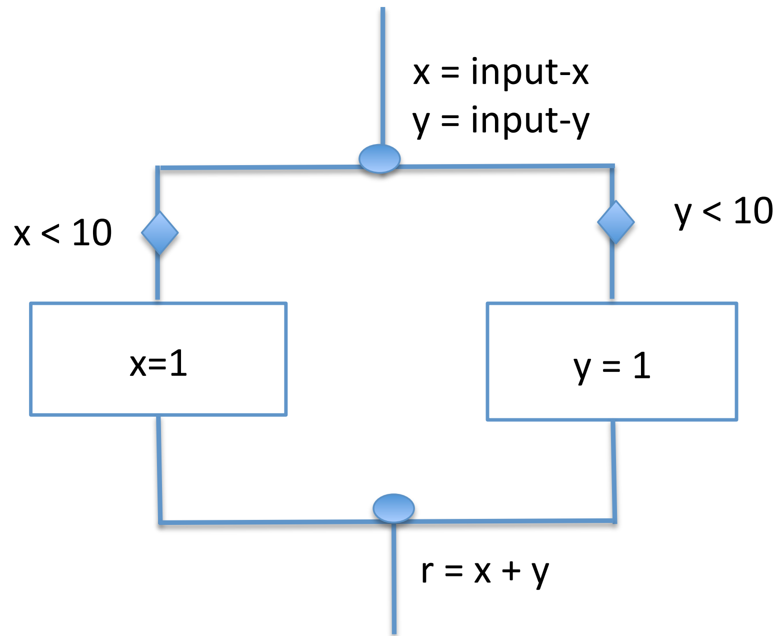


Figure 4.14: Flow Example

in the form of pre- and postconditions (to be added as conditions and assignments), our test case generation algorithms still select paths, but to a corresponding run's model we add also all the parallel branches that might be traversed as well (depending on the actual assignment and the corresponding evaluation of the guards). Deriving a satisfying variable assignment for a constraint representation of this model, we can derive a corresponding test case, and in turn, following different strategies for choosing paths, test suites.

Assuming there is no interaction between the parallel branches, we can derive the variant given in Figure 4.15, supporting also parallel computations. The StructRuns algorithm is thus also search-based, and the only difference is that we derive *run-constraints* as of Def. 10 instead of collecting only the assignments and conditions along the path itself (path-constraints) as in AllPaths algorithm shown in Figure 4.4. If such a run-constraints model is satisfiable, the relevant corresponding variable assignments are saved as a test case.

Lemma 6 (ComputingAllruns)**.** *The set P in the algorithm contains all runs π s.t. $|\pi| \leq MaxLen$.*

Proof. Line 3 in StructRuns as illustrated in Figure 4.15 computes the set P in an extended BPEL flow graph G which as of def. 5 and 8 contains all runs π s.t. $|\pi| \leq MaxLen$. Because of the fact that $|\pi|$ can not exceed $MaxLen$, such a run can not exist. It is *complete*, iff for all satisfied transition guards TG_{e_i} at all $v \in B$ visited by r, the corresponding branch started by edge e_i is present in r. Also according to def. 11 all parallel branches are modeled and synchronized.

□

Lemma 7 (CheckRunSatisfiability)**.** *Each path π included in a test suite S is a feasible run in an extended flow graph G.*

Proof. Only paths where the feasibility check as of def. 9 has been done can included in the test suite S(lines 4-9). Hence, the lemma has to hold.

□

Theorem 8 (StructRunsSoundness)**.** *The StructRuns algorithm is sound such that only feasible runs of length less than MaxLen are added to the test suite S.*

Proof. The proof follows from lemma 6 and 7 directly. We compute all runs. They are feasible and lead to test cases as per definition 12.

□

Theorem 9 (StructRunsCompleteness)**.** *The StructRuns algorithm is complete. That is, no feasible run is excluded from the result.*

Proof. Similarly using Lemma 6, it can be seen that step 3 of the algorithm performs exhaustive depth-first search to find all runs. Since some of runs might be infeasible, based on Lemma 7, steps (4-9) ensure that all feasible runs are included in the result. □

Regarding the termination of StructRuns algorithm in Figure 4.15, we can see that step 3 in the algorithm generates all paths up till $MaxLen$, where the $MaxLen$ is larger than the shortest path in an extended flow graph G. Hence, it can be argued that the search terminates in finite time. For loop in step 4 checks the feasibility of all runs included in the set P, which is bounded by $MaxLen$ parameter, hence is finite and terminates. Step 5 concerns whether run-constraints are satisfiable or not. Both of these steps terminate because of finite run length. The last part is about calling constraint solver to find a solution for run-constraints as of def. 10 and all feasible runs are added in the test suite S.

The complexity of the algorithm can be computed by summing up the complexities of steps 3 to 5. Let us assume that there are $|V|$ number of vertices and $|E|$ number of edges in a flow graph G, then the size of flow graph becomes $|G| = |V| + |E|$. Step 3 computes all runs up to certain $MaxLen$ which in worst-case can lead to exponential paths when $MaxLen$ is larger than the longest path in an extended flow graph G. Step 5 involves call to constraint solver for finding a solution for the run-constraints. Therefore the overall complexity would be $O(|V|^{MaxLen}) + O(|C(\pi)|) + ConstraintSolver(M)$, where $O(|V|^{MaxLen})$ represents an upper bound of the set P, $O(|C(\pi)|)$ denotes the sum of path conditions for all runs included in the set P, and $ConstraintSolver(M)$ denotes the time required to solve the MINION model M.

With flow activity it is possible to execute activities in parallel. However, the execution is not completely parallel. The reason for that is the branches do not execute in concurrent threads. That means one thread starts whenever a fork activity is observed until it reaches some blocking activity like invoke. At this point another thread starts executing the other branch. This restriction is to ensure thread safety of BPEL process variable. In addition to that every fork activity is followed by a join activity, which provides synchronization between parallel branches. The flow activity completes when all branches in a flow have finished processing. The algorithm in Figure 4.15 makes sure that all outgoing branches between a forking node and its corresponding join node in a run along with guards is modeled. Since we do not know which guard is active in the current run, we model all guards in any run computed by the StructRuns algorithm in Figure 4.15.

1: **procedure** StructRuns(G, $MaxLen$)
2: initialize test suite $S \leftarrow \emptyset$
3: compute the set P of all paths π s.t. $|\pi| \leq MaxLen$, where for vertices $v \in B$, we create for each (e_i, TG_{e_i}) in $\gamma_G(v)$, a path s.t. TG_{e_i} is enabled.
4: **for** each path $\pi \in P$ **do**
5: check the satisfiability of run-constraints $C(\pi)$ as of Def. 10
6: **if** $C(\pi)$ is satisfiable **then**
7: add a corresponding test case to S
8: **end if**
9: **end for**
10: return test suite S.
11: **end procedure**

Figure 4.15: Our structural TCG algorithm StructRuns

For example in Figure 4.14, the algorithm computes three possible runs depending on the guard condition. That is, in the first run, we say that the first guard $x < 10$ must be true and the second guard $y < 10$ may or may not be true. Similarly, in the second run, the first guard $x < 10$ may be active, but the second guard $y < 10$ must be true. Also, there is another run, in which both guards may or may not be active. Under this assumption, we get two feasible runs shown in Figures 4.16, and 4.17 respectively.

The vertices for this process are defined by following activities: *receiveInput* (v_{rI}), *IfG* (v_{IG}), *AssignX* (v_{AX}), *IfG2* (v_{IG2}), *AssignY* (v_{AY}), *replyOutput* (v_{rO}).

In our flow example the first branch is triggered via the following run (Run1): $\pi = v_{rI} v_{IG} v_{AX} v_{?IG2} v_{AY} v_{rO}$.

We will use such runs to derive our test cases. That is, as a first step, a search-based algorithm traverses the graph in a depth-first search manner in order to extract the runs from the fork node v_f to the join node v_j. For the possible runs in our flow process, the corresponding run conditions are given in Figures 4.16, and 4.17 respectively. The constraint representation of both runs is detailed in Figures 4.18 and 4.19. Note that, in case any guard is not triggered, we have to propagate the variable values in the respective branch, in order to synchronize the indices at the join node. A

1 $v_{rI} : x_0 = input_x$
2 $v_{rI} : y_0 = input_y$
3 $v_{IG} : x_0 < 10$
4 $v_{AX} : x_1 = 1$
5 $v_{IG2} :?(y_0 < 10)$
6 $v_{AY} : y_1 = 1$
7 $v_{rO} : r_1 = x_1 + y_1$

Figure 4.16: Run 1: $c(\pi)$ for the first guard to be active.

1 $v_{rI} : x_0 = input_x$
2 $v_{rI} : y_0 = input_y$
3 $v_{IG} :?(x_0 < 10)$
4 $v_{AX} : x_1 = 1$
5 $v_{IG2} : y_0 < 10$
6 $v_{AY} : y_1 = 1$
7 $v_{rO} : r_1 = x_1 + y_1$

Figure 4.17: Run 2: $c(\pi)$ for the second guard to be active.

corresponding variable assignment satisfying $c(\pi)$ for Run 1, and thus a test case, is the following one:

```
input_x = 0
input_y = 10
r = 11
```

The basic assumption that must hold in order to use our approach for testing BPEL *flow* activity is that there are no shared variables; i.e., if a variable x is redefined in a branch A, then the variable x should not be redefined in any other branch within a current *flow* activity.

```
1  eq(x_1, input_x)
2  eq(y_1, input_y)
3  reify(ineq(x_1, 10, -1), cond_1)
4  eq(cond_1, 1)
5  reifyimply(eq(x_2, 1), cond_1)
6  reify(ineq(y_1, 10, -1), cond_2)
7  reify(eq(cond_2, 0), cond_3)
8  reifyimply(eq(y_2, 1), cond_2)
9  reify(eq(y_2, y_1), cond_3)
10 weightedsumgeq([1,1], [x_2, y_2], r_1)
11 weightedsumgeq([1,1], [x_2, y_2], r_1)
```

Figure 4.18: MINION constraints for Run 1.

```
1  eq(x_1, input_x)
2  eq(y_1, input_y)
3  reify(ineq(x_1, 10, -1), cond_1)
4  reify(eq(cond_1, 0), cond_2)
5  reifyimply(eq(x_2, 1), cond_1)
6  reify(eq(x_2, x_1), cond_2)
7  reify(ineq(y_1, 10, -1), cond_3)
8  eq(cond_3, 1)
9  reifyimply(eq(y_2, 1), cond_3)
10 weightedsumgeq([1,1], [x_2, y_2], r_1)
11 weightedsumleq([1,1], [x_2, y_2], r_1)
```

Figure 4.19: MINION constraints for Run 2.

4.5 Experimental Setup

WS-BPEL is a product of many XML specifications: WSDL 1.1. is used to represent the BPEL process as well as partner processes and services; XML Schema 1.0 represents the BPEL data model; XPATH is an expression language; and XSLT is meant to provide data manipulation. The BPEL artifacts need execution engine for deployment. There are many proprietary and open-source BPEL tools available. Among these proprietary tools, the most notable ones are *IBM WebSphere process server* [WebSphere, 2006], *Oracle BPEL process manager* [Oracle Process Server, 2010], and *ActiveEndpoints* [Active Endpoints, 2010]. Among open-source tools, we have *Apache ODE* [ODE, 2006] and *Orchestra* [OW2, 2012].

We make use of *Eclipse BPEL designer* [BPELDesigner, 2006] which provides support for defining and editing of BPEL processes. Since the designer comes as an Eclipse plug-in, it can be deployed on an open-source engine like Apache ODE [ODE, 2006], which itself executes as a web application under the application server like Tomcat [Tomcat, 2006]. The setup requires Java 6 or higher for execution. In addition to that we used *Glass fish server* [GlassFish, 2006] for executing the partner web services. For running experiments, we used following versions of the aforementioned software.

- Eclipse BPEL Designer 1.0.3
- Apache ODE 1.3.5
- Apache TOMCAT 7
- GlassFish 3.1 Application Server
- BPELUnit 1.6.0
- MuBPEL 1.2.2
- Minion Constraint Solver 1.6.1

Once the BPEL process is up and running, we can perform testing using our test suite. The quality of the generated test suite can be measured by the unit coverage tool for BPEL process [Lubke et al., 2009], [Lübke, 2006]. The tool can be downloaded as an Eclipse plug-in or as a command-line tool. The tool supports two testing modes, real-time and simulated testing mode. The tool requires user to input manual test cases, which can be executed either on actually running business process or on mock processes. We make

use of the command-line version of the tool for assessing the activity and branch coverage of our generated test suite. The good thing about the tool is that one can choose the deployer, i.e., *Apache ODE*, *ActiveBPEL*, *Oracle*, or *Fixed* in case the user takes the responsibility of the deployment itself.

There can be some problem regarding testing of URLS, which may cause a test suite to fail, that is, if the port on which the process is deployed is busy, or if any web service is down for some reason. Make sure that all web services are up and running, or if you are using mock services, the service endpoints must be specified according to the tutorial guidelines. For our test suite to be executed, it is important that the unit testing tool passes the test suite. Other wise, the coverage results might be different. Alternatively, the plug-in version of the tool can be installed in order to verify the results from our tool (which includes the command-line version). As mentioned in a survey by [Bozkurt et al., 2013], the tool has a problem of oracle. As we already have expected output from *test suite generator module*, our approach can overcome the oracle specification problem.

In addition to the unit testing tool, we also use mutation testing tool for BPEL processes [BPEL Mutation tool, 2011]. The tool comes as a command-line version with an embedded instance of ActiveBPEL engine. The download and installation on a Mac machine can be tricky, if one wishes to use different execution engine than *ActiveBPEL*. Since *ActiveBPEL* is acquired by ActiveEndpoints [Active Endpoints, 2010], and is sold only as a commercial tool to organizations migrating to SOA, we had to make *MuBPEL* work with Apache ODE. It is to be noted that MuBPEL instance runs on port 8080, the other BPEL engine must be executed on some other port. Also the execution takes a long time. It might be a bottleneck, if the test suite size is larger, or if the number of mutants for the subject process exceeds certain number. Another problem encountered was that some mutant might block the port, which could bring the test execution process to halt. Although, we tried to set the maximum time limit of 20 ms for each mutant, but sometimes the MuBPEL tool would not continue the execution. The only solution would then be to stop both *MuBPEL* and the *Tomcat* instances running the *ApacheODE* web application. The tool also has a repository of some small-sized BPEL processes from different vendors. We have included few of those examples in our test experiments. In addition, only synchronous

BPEL processes were found to be stable. This is why our experiments do not include any asynchronous process.

There were many challenges faced during the experimental setup. First, there is a lack of benchmark BPEL examples to be used as an underlying subject data [Bozkurt et al., 2013]. Each execution engine comes with a couple of small-sized business processes, but fail to provide some large-scale examples to measure the effectiveness of different approaches. Second, there were many ambiguities observed in BPEL specifications document and open source BPEL engines [Lapadula et al., 2008, Hallwyl et al., 2010], which hampered the integration of the mutation tool with our approach. Last but not the least, the string datatype, which is quite often used in the message exchange between web services and BPEL processes further complicated the test case generation step.

The test suite execution requires inputs and expected outputs from the BPEL process. The *BPELUnit* tool requires user to specify these inputs and expected outputs for each test case via eclipse plug-in. We however generate test suite automatically using the command line version of the tool. Basically the test suite execution module employs the conversion algorithm, which traverses all feasible paths and converts each *receive* activity into a *send* activity, and each *reply* activity into a corresponding *receive* condition as required by the BPEL Unit Test Suite. Furthermore, we add for any *invoke* activity the corresponding *partner track* information to the XML test suite.

An example test case for run 2 described in the Flow example is shown in Figure 4.20. The *data* tag stores test inputs sent to the BPEL process such as test inputs x and y. The expected value is defined using the *receive* tag.

4.6 First Results

In this section, we report first empirical results obtained using the Java implementation of our BPELTester tool. We considered three examples in our evaluation of sequential processes: *LoanApproval*, *ATM*[1], and a simple hand-crafted example comprising a while statement within the process

[1] http://docs.jboss.com/jbpm/bpel/v1.1/userguide/tutorial.atm.html

```
1   <tes:data>
2    <flo:FlowRequest>
3      <flo:Input_x>10</flo:Input_x>
4      <flo:input_y>0</flo:input_y>
5    </flo:FlowRequest>
6   </tes:data>
7  </tes:send>
8   <tes:receive fault="false">
9    <tes:condition>
10    <tes:expression>//flo:FlowResponse</tes:expression>
11  <tes:value>'11'</tes:value>
12 </tes:condition>
13 </tes:receive>
```

Figure 4.20: Flow Example Test Case

Table 4.1: Examples Details

Prog	BPEL Activites Used
Loan	Receive, Reply, Assign, If, Else, Invoke, Sequence
Atm	Receive, Reply, Assign, If, Else if, While, Invoke, Sequence
While	Receive, Reply, Assign, While, Sequence
Flow	Receive, Reply, Assign, Flow

definition. Table 4.1 list the activities used in these examples. The results from the flow activity are taken from our paper [Jehan et al., 2015].

In Table 4.2 we summarize the obtained test generation results, i.e., the number of BPEL activities in any process **n**, the number of paths **p**, the maximum path length **MaxLen** varying from 10 to 50 (in case of examples with while loops), the minimum and maximum path lengths of the BPEL process **minP** and **maxP**, and the minimum and maximum numbers of MINION constraints **minC** and **maxC**. **totalT** represents the total time in milliseconds it took to generate the executable test cases. The time for checking the paths' feasibility via constraint solving was always very small ranging from 11 to 26 milliseconds and is thus omitted in the table.

Cardoso [Cardoso, 2006] explained the complexity of BPEL processes us-

ing the control flow complexity (CFC) metric. The test execution tool described in [Lubke et al., 2009] supports test coverage metrics like activity, branch, link and handler coverage. Since having a large number of tests is undesirable for testing web services due to the related costs [Canfora and Penta, 2009b], we investigated our test suites' quality with respect to coverage via the tool described in [Lubke et al., 2009]. For all considered examples, we attained 100% activity and branch coverage by just the minimum set of paths. In particular, activity and branch coverage for the Loan and While examples reach 100% for the smallest path length with only 3 paths. For the more complex ATM example, we obtain 100% coverage for a minimum path length of 19 with 13 generated paths (see Table 4.2). The coverage progression as a function of the path length is given for the ATM example in Figure 4.21.

The obtained empirical results are promising and indicate the usefulness of our approach. Even for smaller path lengths, we obtained coverage of 100%, where it took less than 1 second for computing the test suite. It is worth noting that executing the tests took twice the time for generating the test suite.

We used *AllPaths* algorithm to generate tests from sequential activities. The results show the number of generated tests is huge. And we need to perform some analysis to cut down the number of tests and improve the coverage of the generated test suite. For parallel activities, we make use of *StructRuns* algorithm, so as to cater for the simultaneous execution of parallel branches. In next Section, we analyze our random testing approach for sequential programs.

4.7 Random Testing of Sequential programs

A big challenge in testing SOA-based system is the testing time required to ensure the desired functionality of the overall system. Because of the fact that most service-oriented systems are meant to be accessed through web interfaces, the runtime performance becomes crucial. Be it social networking services such as *Twitter*, *Facebook* or *Linked-In* or commerce companies such as *Amazon*, users need to be assured that they receive service within given

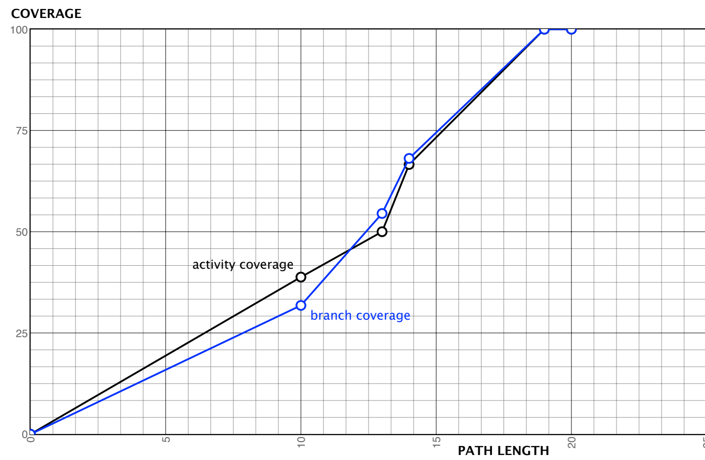


Figure 4.21: Coverage vs. path length for the ATM example.

Table 4.2: Empirical results obtained

Prog	n	MaxLen	p	minP	maxP	minC	maxC	totalT
Loan	16	10	3	8	10	8	11	160
Atm	27	10	1	9	9	10	10	57
		20	17	9	20	10	22	1,312
		30	132	9	30	10	33	9,150
		40	1,367	9	40	10	46	108,983
		50	12,950	9	50	10	57	1,372,059
While	8	10	3	8	10	13	16	169
		20	28	8	20	13	28	2,028
		30	78	8	30	13	45	6,194
		40	153	8	40	13	62	13,703
		50	253	8	50	13	75	23,849
Flow3	11	15	2	11	11	11	11	243

time. This means, the back-end testing of the whole system needs to meet strict time constraints. There are numerous approaches discussed in the literature to meet-up this challenge. We compare random testing with structure-based testing and discuss results from this comparative analysis in Section 4.7.2.

4.7.1 Introduction

Random testing has been effectively used in practice to aid automated testing of functional programs. [Claessen and Hughes, 2000] introduced a tool (QuickCheck) for an automated random testing of Haskell programs. The tool is able to verify formal properties specified as Haskell functions. These properties can be used as test oracles (e.g., if a test has passed or not). The authors have included a number of case-studies to show the effectiveness of their tool. However, they believe that the effectiveness of random testing is largely dependent on the distribution of the test data.

[Godefroid et al., 2005b] developed a tool called *DART* for directed automated random testing of C programs, thereby eliminating the need to write test drivers. They argue that test drivers are difficult to write, therefore unit testing is ignored causing serious reliability issues. Their approach is also able to perform dynamic program analysis, and if necessary generate new test inputs along alternative program paths, thereby alleviating the common issue of "low coverage" with the random testing approach. It has been shown to uncover errors like assertion violations and non-termination. They claim to discover 65 % of about 600 available functions in the oSIP library. In contrast to these papers, we apply random testing on a model obtained from the source code directly.

Although mutation testing is effective in measuring the test suite quality, the required time for computing the mutation score of the test suite generated by our *AllPaths* algorithm described in Figure 4.4 is quite high. Therefore, the random approach might help to attain a reasonable mutation score within less time.

Our random TCG algorithm shown in Figure 4.22 requires three parameters: a BPEL flow graph G, the maximum path length $MaxLen$, and the required

randomTests $numTC$ for the resulting test suite S. The first two parameters, G and maximum path length $MaxLen$ are same as that in $AllPaths$ algorithm (see Figure 4.4). The random Algorithm only generates specific number of tests defined by $numTC$, thereby reducing the test suite generation time. It is important to understand the role of $MaxLen$, because theoretically any length can be chosen, we have set a value higher than the longest path for examples. Due to random selection of paths, a single path might get selected many times. Like $AllPaths$ algorithm, we make use of the minion constraint solver to find satisfying statements about the feasible paths in G, which are stored in a test suite.

Lemma 10 (RandomTestSuiteSize). *The test suite S contains only random test cases s.t. $|S| < numTC$.*

Proof. Line 4 in algorithm ensures that test suite size does not exceed $numTC$.

□

Lemma 11 (ComputingFeasibleRandompaths). *The algorithm illustrated in Figure 4.22 contains all feasible random paths π s.t. $|\pi| < MaxLen$.*

Proof. Lines 6 makes sure that $|\pi|$ does not exceed $MaxLen$. In Line 7, we select randomly a successor vertex v out of $V\ \exists e = (f(\pi), v) \in E$. The vertex v is appended at the end of π as long as $|\pi| < MaxLen$. Lines 9 to 13 checks for the feasibility of the random path whenever a leaf vertex $f(\pi)$ is reached. Hence, only feasible randomly selected paths are included in the test suite S.

□

Theorem 12 (RandomPathsCorrectness). *$RandomPaths$ algorithm 4.22 is correct. That is, all feasible random paths of length less than MaxLen are included in the result S s.t. $|S| < numTC$.*

Proof. The proof follows directly from Lemma 10, where only random test cases s.t. $|S| < numTC$ are added to the test Suite S. Hence, the algorithm is sound. Similarly, making use of Lemma 11, it can be argued that the

```
1:  procedure RandomPaths(G, Len, numTC)
2:      initialize test suite S ← ∅
3:      initialize MaxLen ← random(0, Len)
4:      while |S| < numTC do
5:          initialize π ← v_0
6:          while |π| < MaxLen do
7:              pick random v ∈ V s.t. ∃e = (f(π), v) ∈ E
8:              add v to π: π ← πv
9:              if f(π) ∈ F then
10:                 if C(π) is satisfiable then
11:                     add a satisfying assignment to S
12:                 end if
13:             end if
14:         end while
15:     end while
16:     return test suite S
17: end procedure
```

Figure 4.22: TCG algorithm based on random paths.

algorithm is also complete, because it only all feasible random paths s.t. $|\pi| \leq MaxLen$.

□

The termination of the *RandomPaths* algorithm 4.22 is ensured by while conditions in line 4 and 6, e.g., only paths upto *numTC* are generated, where the length of each random path can not exceed *MaxLen*. The *MaxLen* is some random number between 0 and the specified *Len*. The complexity of the algorithm would be $O(|V|^{MaxLen}) + O(|C(\pi)|) + ConstraintSolver(M)$, where $O(|V|^{MaxLen})$ represents an upper bound on the number of random paths, $|C(\pi)| = \bigcup_{(0<i<numTC)} \Gamma_A(\pi_i) \cup \Gamma_C(\pi_i)$ denotes the sum of path conditions for random paths bounded by *numTC*, and $ConstraintSolver(M)$ represents the time required to solve the MINION model M.

4.7.2 Experimental Results

The main focus of the comparative analysis of structured and the random approach was to analyze the performance of both approaches in terms of unit-coverage and the mutation score.

For our experiments, we used several SOA examples and converted also a software example to fit our environment. The processes were developed using Eclipse BPEL process editor and were deployed on Apache ODE engine. For computing activity and mutation coverage, we used *BpelUnit* [Lübke, 2006] and *MuBPEL*[BPEL Mutation tool, 2011] tools.

The three SOA examples *Loan*, *LoanCov* and *SquaresS* are available from the mutation tool repository [BPEL Mutation tool, 2011], where we used the *Loan* example also in [Jehan et al., 2013a]. While *LoanCov* is a slight variation of the *Loan* example, *SquaresS* computes the obvious arithmetic function. The fourth SOA example *ATM* is a simplified version of the process discussed in [ATM,]. With *Triangle*, we implemented also a typical example from software engineering studies like [Langdon et al., 2010]. This process decides for a given triangle whether it is equilateral, isosceles or scalene. The activity types involved in the different examples include Receive, Reply, Assign, If, Else if, While, Invoke, and Sequence. For the computation of reported numbers we used a 13" MacBook Pro (Late 2011) with a 2.4 GHz Intel Core i5, 4 GB 1333 MHz DDR3, running OS X 10.7.2.

The results obtained from *AllPaths* algorithm are shown in Table 4.3. The example size is denoted by n, whereas the given maximum path length is labeled as *MaxLen*. The number of paths for the chosen maximum length is represented by p, *miP* and *maP* stand for the minimum and maximum lengths of derived paths respectively. Similarly, the minimum and maximum numbers of constraints derived for any path are reported as *miC* and *maC* respectively, and *TotalT* defines the total time in milliseconds used to derive test suite S. The most vital numbers added are for *Cov* and *Mut* that give us the percentage of activities covered as well as the number of killed mutants, respectively.

The results indicate that by increasing *MaxLen*, both coverage and mutation score values increase, but, at the cost of increasing computation times and a

larger test suite. Figure 4.23 describes the relationship between these two measures. That is, by increasing path length, both coverage and mutation scores increase. It can be also be observed, that unit coverage of 100% is achieved, but mutation score never exceeded 90 percent. This is because of the surviving mutants, which could not be killed by the test suite. Considering that detection of equivalent mutants is a non-decidable problem, a mutation score of 100% is practically not possible. The only way to check for equivalent mutants is through manual inspection.

The detailed results of the random TCG algorithm are shown in Table 4.4. Due to random selection of the feasible paths, we took an average of ten executions per row. The needed number of random tests is defined by *rP*, where *miP* and *maP* represent the minimum and the maximum path in the obtained ten executions. Likewise, the *minCov* and *maxCov* means the coverage obtained for the minimum and the maximum random path. *avgCov* is the average value of the minimum and maximum coverage. The standard deviation in the obtained mutation score is represented by *stdev*.

The experiments were carried out to observe the effect of a combined increase of the maximum path length and the number of random tests. It was observed that the random approach is good at reducing the computation time, but at the same time the mutation score is also declined. For example, in case of *LoanCov* example, the *avgMut* was dropped at 48,35% as compared to 71% for the "structural" AllPath approach. The same trend was observed in case of *Atm* example, where an average mutation score of 46,05%, in contrast to 87,87% with the *AllPaths* algorithm was observed. In addition to this, the *minMut* and *maxMut* of 23,63% and 68,48%, indicated high standard deviation for the computed mutation score. Only in case of *SquareS* example, the results were more or less same as that of *AllPaths* algorithm. The results show that the structural approach performs better than the random one in the context of our experimental setup. In the next chapter 5, we focus at identifying interesting combinations of both approaches.

Table 4.3: Experimental results for the AllPath TCG algorithm.

Prog	n	MaxLen	p	miP	maP	miC	maC	totalT	Cov	Mut
Loan	16	10	3	8	10	8	11	160	100	88.76
Atm	27	10	1	9	9	12	12	57	40	23.63
		15	5	9	15	12	20	322	75	67.87
		19	13	9	19	12	28	878	100	87.27
SquareS	7	10	3	7	10	11	19	312	100	89.65
		15	5	7	15	11	26	368	100	89.65
		20	8	7	20	11	39	641	100	89.65
LoanCov	27	10	3	8	10	8	10	166	64	44.26
		15	5	8	13	8	13	290	100	71.03
		20	5	8	13	8	13	290	100	71.03
Triangle	22	10	1	7	7	7	7	366	38	11.97
		15	4	7	15	7	15	801	92	66.46
		20	5	7	16	7	16	722	100	74.85

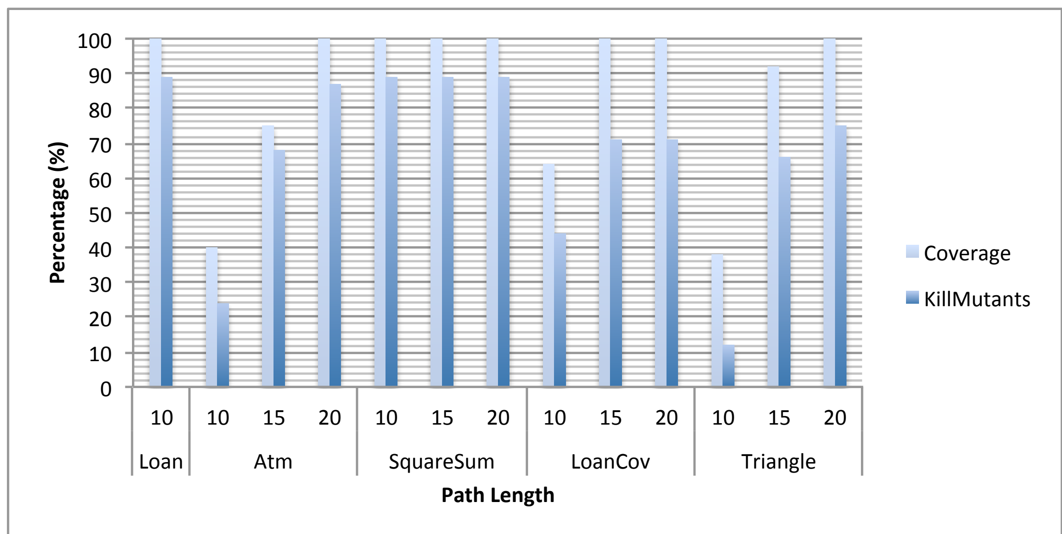


Figure 4.23: AllPaths TCG alg: activity coverage and mutation score vs. path length

Table 4.4: Experimental results for the random TCG algorithm.

Prog	n	**Len**	**rP**	**miP**	**maP**	**miC**	**maC**	**totalT**	**minCov**	**maxCov**	**avgCov**	**minMut**	**maxMut**	**avgMut**	**stdev**
Loan	16	20	1	8	12	8	12	102	36	71	53.5	24.71	55.05	39.88	21.45
		30	2	8	12	8	12	277	36	92	53.5	24.71	74.15	49.43	34.95
		40	3	8	12	8	12	200	37	100	53.5	24.71	88.76	56.73	45.29
Atm	27	20	1	9	22	13	31	68	37	66	51.5	18.18	63.63	40.91	32.13
		30	2	9	22	13	30	104	37	70	53.5	23.63	46.06	34.84	15.86
		40	3	9	28	13	42	169	37	77	57	23.63	68.48	46.05	31.71
SquareS	7	20	1	7	13	11	19	70	100	100	100	82.75	88.50	85.62	4.06
		30	2	7	13	11	19	142	100	100	100	82.75	89.65	86.2	4.87
		40	3	7	21	11	31	241	100	100	100	82.75	89.65	86.2	4.87
LoanCov	27	20	1	8	13	8	13	58	26	43	34.5	20.21	28.41	24.31	5.79
		30	2	8	13	8	13	120	42	77	59.5	23.49	47.54	35.51	17.01
		40	3	8	13	8	13	166	26	91	58.5	26.77	55.73	41.25	20.47
		50	5	8	13	9	13	385	47	84	65.5	34.97	61.74	48.35	18.92
Triangle	22	20	1	7	16	7	16	574	38	69	53.5	11.97	50	30.98	26.89
		30	2	7	15	7	15	402	40	77	58.5	35.92	52.09	44.01	11.43
		40	3	7	12	7	16	1032	69	85	77	49.40	63.47	56.43	9.94

4.7.3 Conclusions

In this Section, we compared our test-case generation process considering all paths in a BPEL control-flow graph with a random path selection. The aim was to reduce the high time required for *AllPaths* algorithm. According to the results obtained from the comparison, the random did require less run-time, but the mutation coverage was comparatively quite low as compared to the allpath selection. Second, high fluctuation in the standard deviation was observed, which shows that the test suite size plays an important role in a totally random approach. Not only this, the experimental setup was found to be quite complex and time consuming. The next Section considers the percentage of infeasible paths and apply the random concept also on the flow constructs by extending the empirical results.

4.8 Random Testing of Concurrent programs

4.8.1 Introduction

In previous Section 4.7, we presented a random search-based algorithm for sequential BPEL processes and compared its performance with the structural approach 4.4 of exploring all paths in a control flow graph. In this Section, we extend our work on parallel BPEL processes and compare the performance of different test suite generation approaches. The general idea of the algorithm remains the same, that is to traverse a BPEL process' flow graph in order to extract the necessary inputs and expected outputs for actual executions. However, regarding testing the *Flow* processes, we are interested in *runs* in the control flow graph, where more than one branch may be active at a given time.

In order to apply random testing of BPEL flow structure, we tailored the algorithm in Figure 4.15. The RandomRuns algorithm like StructRuns is also search-based. Also, the *run-constraints* are derived using the same Def. 10 instead of collecting only the assignments and conditions along the path itself as of Def. 2. Similarly, only the satisfiable run-constraints along with the corresponding variable assignments are saved as test cases. The

only difference between RandomRuns and StructRuns is that instead of considering all of feasible runs in Figure 4.15, we evaluate the effectiveness of random approach. Specifically, do we attain better coverage and mutation score with the random approach or not? The RandomRuns algorithm in Figure 4.22 like the RandomPaths algorithm extracts a desired number of random test cases, limited in length by a given parameter.

Lemma 13 (ComputingFeasibleRandomruns)**.** *The algorithm illustrated in Figure 4.24 contains all feasible random runs* π *s.t.* $|\pi| < MaxLen$.

Proof. Lines 5 makes sure that $|\pi|$ does not exceed $MaxLen$. In Line 6, we select randomly a successor vertex v out of V such that $\exists e = (f(\pi), v) \in E$. The vertex v is appended at the end of π as long as $|\pi| < MaxLen$. Lines 8 to 14 checks for the feasibility of the random path whenever a leaf vertex $f(\pi)$ is reached. Hence, only feasible randomly selected runs are included in the test suite S.

□

Theorem 14 (RandomRunsCorrectness)**.** *RandomRuns algorithm 4.24 is correct. That is, all feasible random runs of length less than MaxLen are included in the result* S *s.t.* $|S| < numTC$.

Proof. The proof follows directly from Lemma 10, where only random test cases are added to the test Suite S s.t. $|S| < numTC$. Hence, the algorithm is sound. Similarly, making use of Lemma 13, it can be argued that the algorithm is also complete, because it includes all feasible runs s.t. $|\pi| < MaxLen$.

□

The termination of *RandomRuns* algorithm 4.24 is ensured by while conditions in line 3 and 5, e.g., only runs up to *numTC* are generated, where the length of each random run can not exceed *MaxLen*. The *MaxLen* is some random number between 0 and the specified *Len*. The complexity of the algorithm would be $O(|V|^{MaxLen}) + O(|C(\pi)|) + ConstraintSolver(M)$, where $O(|V|^{MaxLen})$ represents an upper bound on the number of random runs, $|C(\pi)| = \bigcup_{(0<i<numTC)} \Gamma_A(\pi_i) \cup \Gamma_C(\pi_i)$ denotes the sum of path conditions

```
1:  procedure RandomRuns(G, Len, numTC)
2:      initialize test suite S ← ∅
3:      while |S| < numTC do
4:          initialize path π ← v_0
5:          while |π| < MaxLen do
6:              pick random v ∈ V s.t. ∃e = (f(π), v) ∈ E
7:              add v to π: π ← πv
8:              if f(π) ∈ F then
9:                  if run-constraints C(π) (see Def. 10) are satisfiable then
10:                     add a corresponding test case to S
11:                 end if
12:             else
13:                 increment infeasible paths
14:             end if
15:         end while
16:     end while
17:     return test suite S
18: end procedure
```

Figure 4.24: TCG algorithm RandomRuns based on random paths.

for random runs bounded by *numTC*, and $ConstraintSolver(M)$ represents the time required to solve the MINION model M.

4.8.2 Empirical Evaluation

The main objective of empirical evaluation was to compare the performance of StructRuns and RandomRuns algorithms. We evaluated the approach by varying the *rP* parameter while keeping the length parameter fixed. Moreover the example set is extended by including BPEL processes with Flow constructs.

For our empirical evaluation, we considered ten examples of synchronous BPEL processes. These examples include activities like Receive, Reply, Assign, If, Else if, While, Invoke, Sequence and Flow. In addition to *Loan*, *LoanCov* and *SquaresS*, *SquaresS*, and *ATM*, we have two more sequential

BPEL process, i.e., *Bmi* and *Calc*. *Bmi* is a famous example taken from software testing papers, whereas, *Calc* implements basic calculator functionalities, i.e., addition, subtraction, multiplication, and division for given input values. The examples *Flow* and *Flow3* are simple hand-crafted examples using Flow construct, whereas *Order* is a variant of an Ordering Service from the BPEL specification document.

Tables 4.5 and 4.6 offer the experiments' details when using the StructRuns and RandomRuns algorithms respectively. The number of a BPEL process' activities is given by n, the desired maximum path length by mL, the number of derived paths is labeled p, the minimum and maximum lengths of derived paths are given in columns labeled *miP* and *maP* respectively, and the minimum and maximum numbers of constraints derived for any path are reported as *miC* and *maC* respectively. *GenT* defines the total time in milliseconds it took us to derive a corresponding test suite S. The most interesting values, however, are those for *Cov* and *Mut* that give us the percentage of covered activities and killed mutants, respectively.

The overall test execution time the mutation tool took to compute mutation coverage (in milliseconds) is given in the columns labeled $ExecT$. That is, for the RandomRuns algorithm, we computed 10 samples per row and report the minimum, maximum, and average values for coverage and mutation scores respectively, with also *GenT* referring to the average value over these 10 samples. For the mutation score, we also report on the standard deviation *stdev*. In Table 4.6, *rP* defines the desired amount of test cases.

The results depicted in Tables 4.5 and 4.6 are useful in drawing following conclusions: First, the test suite generation time can be considered negligible as compared to the time required by the mutation tool. For example, the test suite generation time for nearly all examples is always less than a second, but the test execution is on average never less than half an hour. Therefore, it would not be wrong to say that SOA tools seem to be bottleneck when it comes to test execution. This holds true even in case of smaller examples like *SquareSandBMI*. Second, in case of both the algorithms, StructRuns and RandomRuns, test case generation time is almost the same. But, on average the StructRuns algorithm shows better coverage and mutation scores as compared to the RandomRuns algorithm nearly for all examples.

Table 4.5: Experimental results for the StructRuns TCG algorithm.

Prog	n	mL	p	miP	maP	miC	maC	GenT	Cov	Mut	ExecT
Loan	16	10	2	8	9	9	11	176	76.9	68.50	629,271
		15	3	8	12	9	13	227	100.0	87.64	966,228
		20	3	8	12	9	13	247	100.0	87.64	962,418
Atm	27	10	1	9	9	12	12	68	35.2	21.77	468,613
		15	5	9	15	12	20	569	100.0	80.64	2,411,368
		20	5	9	15	12	20	985	100.0	80.64	2,442,709
SquareS	7	10	3	7	10	12	17	200	100.0	88.51	726,955
		15	5	7	15	12	32	566	100.0	89.65	1,279,305
		20	8	7	19	12	42	830	100.0	89.65	2,421,234
LoanCov	27	10	3	8	10	11	11	293	64.0	52.54	1,346,072
		15	5	8	13	11	15	433	100.0	71.03	1,971,916
		20	5	8	13	11	15	467	100.0	71.03	1,971,476
Triangle	22	10	1	7	7	9	9	354	38.0	12.34	870,823
		15	4	7	15	9	25	477	92.0	66.04	2,289,147
		20	5	7	16	9	26	718	100.0	71.03	2,651,180
Bmi	15	10	5	7	9	9	9	485	100.0	90.00	1,081,270
Calc	30	10	4	5	10	6	20	248	40.0	38.63	1,633,480
		15	9	5	15	6	32	591	100.0	98.37	3,496,140
Flow	11	15	1	11	11	14	14	156	83.3	54.00	450,463
Flow3	11	15	2	11	11	11	11	243	100.0	83.30	326,788
OrderFlow	24	25	2	24	24	41	41	378	100.0	61.00	366,954

Table 4.6: Experimental results for the RandomRuns TCG algorithm with len = 40.

Prog	n	rP	miP	maP	miC	maC	GenT	miCov	maCov	avgCov	miMut	maMut	avgMut	stdev	ExecT
Loan	16	1	8	12	9	13	451.0	46.1	69.2	51.49	24.71	52.80	39.21	15.28	526,905
		2	8	12	9	13	445.0	46.1	76.9	66.89	24.71	68.53	55.16	17.51	696,221
		3	8	12	9	13	641.0	46.1	100.0	86.14	24.71	87.64	67.41	17.13	884,560
Atm	27	1	9	14	11	20	479.0	41.1	52.9	50.54	21.77	45.96	33.70	12.58	447,316
		3	9	15	11	20	644.4	35.2	76.4	54.66	21.77	66.12	48.22	15.03	1,111,243
		5	9	15	11	20	897.5	47.0	100.0	65.85	48.38	80.64	59.59	9.74	1,821,139
SquareS	7	3	7	13	12	27	405.3	100.0	100.0	100.00	83.90	89.65	88.50	1.71	808,907
		5	7	25	12	57	728.2	100.0	100.0	100.00	83.90	89.65	88.85	1.80	1,232,068
		8	7	21	12	47	872.9	100.0	100.0	100.00	88.50	89.65	89.54	0.36	1,930,244
LoanCov	16	3	8	13	11	15	926.3	47.3	89.4	64.15	33.33	55.73	43.98	11.77	1,325,958
		5	8	13	11	15	811.3	47.3	94.7	75.74	34.97	61.74	52.89	7.82	1,884,014
Triangle	22	1	7	12	9	15	475.0	33.3	66.6	50.00	12.96	45.37	26.47	15.04	1,002,234
		4	7	15	9	26	654.0	33.3	83.3	74.20	12.96	58.95	44.41	14.39	1,818,729
		5	7	15	9	26	646.0	75.0	91.6	85.00	37.96	69.75	59.35	9.86	2,722,881
		7	7	15	9	26	915.4	75.0	91.6	84.10	37.34	65.74	57.75	7.78	2,800,813
Bmi	15	1	7	9	9	9	158.7	60.0	60.0	60.00	29.09	52.72	42.09	9.99	406,481
		3	7	9	9	9	737.7	60.0	80.0	74.00	45.45	72.72	64.27	10.61	733,752
		5	7	9	9	9	700.0	70.0	90.0	83.00	60.90	81.81	73.18	7.68	1,049,588
		7	7	9	9	9	797.4	80.0	100.0	87.00	62.72	90.00	77.72	7.59	1,436,189
		10	7	9	9	9	816.8	90.0	100.0	92.00	80.90	90.00	83.00	3.71	2,240,604
		12	7	9	9	9	678.5	90.0	100.0	97.00	80.90	90.00	87.27	4.39	2,645,961
		15	7	9	9	9	852.7	90.0	100.0	98.00	80.90	90.00	88.18	3.83	2,737,402
		17	7	9	9	9	877.1	90.0	100.0	97.00	80.90	90.00	87.27	4.39	3,726,332
Calc	30	4	5	25	6	38	703.6	35.0	70.0	46.00	29.73	57.18	39.90	10.28	1,529,496
		9	5	24	6	56	822.7	40.0	100.0	68.00	38.23	84.64	59.31	15.72	2,930,746
Flow	11	1	11	11	13	23	768.4	83.3	83.3	83.30	54.54	54.54	54.54	1.17	374,245
Flow3	11	1	11	11	11	17	545.5	50.0	83.3	66.65	42.66	52.00	43.60	6.56	283,808
		2	11	11	13	19	884.7	83.3	100.0	91.65	52.00	70.66	67.73	5.65	333,727
Order	24	1	23	23	42	42	826.6	47.0	94.1	65.84	40.84	52.11	45.35	5.81	264,348
		2	23	23	42	42	1059	47.0	100.0	82.33	42.25	60.56	52.53	7.99	348,207

During the experiments, we encountered issue of infeasible paths, particularly for lengths longer than 20. The results computed for the *Calc* example are shown in Table 4.7. Here, we observed no infeasible paths in case of the StructRuns algorithm put to a path length of 20. But, for theRandomRuns algorithm, the number of infeasible paths increase if the maximum path length is increased. The issue appears more often in case of longer test case. For instance, with $|S| = 4$, we have 3 infeasible paths out of 10 random samples. The number increases to 8 infeasible paths for a test suite of size 9. Furthermore, the achieved mutation score is just 80%, which led us to an interesting question, that is how many more random paths do we need in order to attain the same mutation score as with the StructRuns algorithm.

In order to investigate that question, we considered the *BMI* BPEL process, and test suite sizes rP of 1, 3, 5 7, 10, 12 ,15, and 17 (see Table 4.6). The coverage attained for 12 to 17 paths was roughly the same, with the execution time increasing from approximately 45 minutes to 62 minutes on average.

In Figure 4.25 we summarize our findings using a box plot diagram, where the grey box indicates the bounds given by the average value and the standard deviation. The obtained results show that, for 7 test cases or more, the random algorithm provided the same 100% activity coverage and maximum mutation score of around 90% as with the StructRuns algorithm. For 12 test cases and above, the average mutation score also starts converging to the values obtained by StructRuns algorithm. However, the RandomRuns algorithm took almost double time for generating a test suite with similar average performance. For instance, the random approach took 700 milliseconds on average for computing five paths, where as the StructRuns took (smaller) 485 milliseconds to compute 5 paths for a given maximum length of 10. Not only this, the the StructRuns attained a higher mutation score of 90% percent as compared to an average mutation score of 73.18% (ranging between 60.9 and 81.81 percent) by the RandomRuns algorithm.

It is also worth mentioning that there were many surviving mutants for each of the examples used in our empirical evaluation. The highest achieved mutation score was 90%. In order to investigate the non killed mutants, we inspected the surviving mutants manually for the *BMI* example. We

Table 4.7: Infeasible paths for the RandomRuns TCG algorithm.

Prog	rP	min InP	max InP	avg InP
Calc	4	0	3	0.4
Calc	9	0	8	1.4

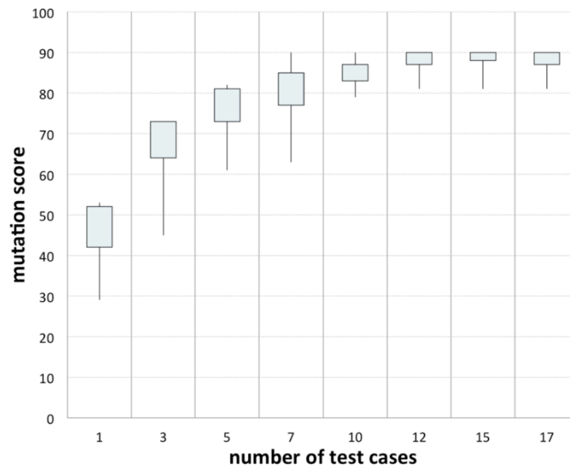


Figure 4.25: RandomRuns: Mutation score as function of the number of test cases for *BMI*

found out that by adding five additional test cases, we were able to reach a mutation score of 95%. The remaining mutants were equivalent ones. Hence, we conclude that there is still room for improving the test case generation process, in order to deliver an algorithm with a better performance.

4.8.3 Discussion

The focus of our experiments was the comparison of two test suite generation algorithms for synchronous executable BPEL processes. The first algorithm, StructRuns is used to derive test suite comprising of all feasible paths up to certain length, while the second algorithm, the RandomRuns aimed at deriving specific number of test cases considering only a random selection of feasible paths of certain length. The StructRuns algorithm performed better both in terms of test-case generation and execution time. For example,

the BMI example, the test-case generation time of 3 random paths is almost same as that of 5 random paths, but the test execution time has almost doubled. Not only does the random approach longer to construct test cases, the attained coverage and mutation score was also considerable low as compared to the structural approach. That is, using StructRuns algorithm, we attained mutation score of 90% with just 5 tests, while the RandomRuns failed to achieve the same average mutation score with 17 tests. Hence, we can conclude from the experiments that the high execution times should be reduced by employing smarter test-case generation strategies. Second, our approach does not perform well using longer paths as allowed in the random approach, rather the shorter paths favored in the structural approach gave better results. Therefore, a hybrid approach combining both the structural and random approach should be explored for an optimized performance.

4.9 Conclusions

The model-based testing is employed to derive test cases from the SOA process. Model-based testing and verification using symbolic execution is a widely investigated area in testing of business workflows. This is because of three main reasons: first, with formal verification, the integrator, a middleman between the service provider and the consumer, can reduce the testing cost by verifying the unreachable parts of the code offline. This is important, because the malfunctioning of any part of the workflow, may lead to severe problems in the monitoring phase. Second, the verification technique is used to derive test inputs, thereby reducing the testing cost. Third, the approach has a high coverage with small number of paths.

We showed that the BPEL process definitions can be represented as set of constraints, which served as a model for our approach. The model along with the test case represented the constraint-satisfaction problem, and solutions can be extracted using a constraint solver. Furthermore, we judge the quality of the generated test-suite using the unit-coverage analysis and mutation testing SOA tools. The approach was tested on a number of sequential and parallel executable BPEL processes. The test case generation time was

negligible as compared to the test execution time. In order to overcome high test execution cost, we suggest a number of test suite reduction techniques in the next chapter. The goal would be to reduce the size of the test suite, while maintaining the same coverage and mutation scores. Our approach requires multiple positive test cases for the mutation tool to be applied for judging the quality of the generated test suite.

There are, however, few limitations observed during the experiments. That is, due to non-availability of real-world case studies, the experiments are performed on the synthetic examples only, so it became difficult to compare our approach with other approaches suggested by different research groups. Second, the experiments were performed using open-source SOA tools, which might be not be as stable as the proprietary tools available from Oracle and IBM. Third, like most of the test generation approaches, we derive model from the executable composition code rather than from the requirements document. However, the assertions can be added to any activity in the workflow, which can serve as test oracles.

5 Test Suite Reduction

5.1 Introduction

Test suite minimization is the task of finding a smaller test suite that still fulfills the properties of the original test suite but which requires less test cases. Minimizing test suites is important in practical applications of testing for reducing the time required for carrying out all the tests. This holds especially in cases with strict time requirements for example when carrying out nightly-builds that should be deployed next day to the customers. In such a case not only time requirements are important. It has also to be ensured that the reduced test suite has more or less equally good failure detection capabilities than the original one. Because of the fact that we would only be able to compare the failure detection rates of two test suite when executing both, which would not be possible in the mentioned scenario, test suite minimization usually considers properties of test suites like coverage or mutation score. Here the underlying assumption is that these properties can be used as a quality measure of test suites.

Any test suite that we are able to reduce via removing tests while keeping its degree of failure detection capabilities obviously stores redundant information. As a consequence test suite minimization based on test case deletion aims at eliminating all redundancies from the test suite. Therefore, we formally introduce redundancy for test suite relying on quality measures and discuss some algorithms that can be used for redundancy elimination while still fulfilling certain properties like coverage or mutation score. Some parts of the content in this Chapter has been published in the following paper.

- "Analyzing the reduction of test suite redundancy" [Pill et al., 2015].

5.2 Related research

Test suite minimization is the task of identifying and later on eliminating redundant test cases from the original test suite. This is required because as software is constantly evolving the corresponding test suite grows accordingly, making re-testing an expensive task. This is why test suite minimization has been a subject of wide interest as part of regression testing, which aims at re-testing a software after modifications.

Regression testing faces new challenges in SOAs for many reasons: First, traditional regression testing is performed in a white-box manner, which due to observability issues in SOA domain can only be performed at the developer side. That means *integrator* or *consumer* do not have realistic test suites. Second, due to controllability problem, it is even not possible to decide at the consumer side when is it appropriate to perform regression teasing in the first place [Canfora and Penta, 2006]. Third, the cost of service invocation is the main bottleneck in dynamic service composition testing, especially if services are charged on per-use basis [Bozkurt, 2013]. Concurrency issues in service-based compositions is another important area of interest.

Regression testing can be classified as test suite minimization (reduction), test suite selection, or test suite prioritization. Test suite minimization purely focusses on elimination of redundant test cases, test suite selection, on the other hand, does not remove any test case, rather selects only appropriate tests for testing the modified or added functionality. Test suite prioritization focusses on test ordering to maximize certain properties like coverage or fault-detection.

[Rothermel et al., 2002] formally defined *test suite reduction* as follows: Given a test suite T, a set of test case requirements $r_1, r_2, ..., r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of T, $T_1, T_2..., T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to test r_i. The problem is to find a representative set of test cases from T that satisfies all r_is. The optimal test suite reduction is the one, which contains at least one test case requirement t_j from each subset T_i. The problem is considered analogous to the finding of minimal hitting-set problem. Considering the fact, that this itself is a

NP- complete problem, different heuristics have been suggested to reduce software maintenance costs [Yoo and Harman, 2012].

Due to limited observability, most of regression testing approaches are based on Graph Walk Approach (GWA)[Rothermel and Harrold, 1997]. This approach was first suggested for regression test selection, and does not require access to the source-code. [Ruth and Tu, 2007] applied a slight variation of GWA for regression testing of web services. Their approach require CFGs from all participating parties. [Wang et al., 2008] suggested an approach for regression testing of BPEL compositions based on breadth-first search. Their work is also based on [Rothermel and Harrold, 1997] approach. [Lin et al., 2006] discussed the safe regression testing of Java-based web services employing an end-to-end approach.

Test suite prioritization focusses on the permutation of test cases in order to maximize testing objective. This prioritization could be coverage-based, history-based, or probability-based. Some prominent test suite prioritization algorithms include *greedy algorithms*, *meta heuristics* and *evolutionary search approaches* [Yoo and Harman, 2012]. *Greedy algorithms* follow the greedy principle of incrementally adding test cases to an empty space so as to maximize the desired metric. But, they might fail to come up with an optimal test case ordering always. *Meta heuristic techniques* find solution to combinatorial problems at an economical cost. The target of *evolutionary search algorithms* is to follow the survival of the fittest strategy for test suite prioritization.

[Heimdahl and George, 2004] derived model-based tests from formal specifications of a "Flight guidance system". They claim that although test suite reduction is helpful in reducing the test suite size while maintaining the same coverage. However, the fault detection capabilities of the reduced test suite is also effected. [Hou et al., 2008] is one of the earliest work done on test case prioritization in the context of service compositions. Since there is an upper bound on the number of requests a web service can be called in a certain period of time, their approach does the test case selection using the Integer Linear Programming (ILP) using the given time slots. An enhanced work from the same authors is presented in [Zhang et al., 2009], the focus remains the same, that is, how to select a subset of the original test suite in order to reduce regression testing costs within the time constraint. They

model the problem as a constraint system with the objective of increasing the statement coverage of a test suite satisfying the time constraint. [Mei et al., 2009] proposed two black-box test case prioritization techniques making use of tag information present in the WSDL document of invoked web services on a set of BPEL programs.

In the context of SOAs, [Ruth et al., 2007] discussed the use of *call graphs* to tackle the concurrency issue arising from service compositions. Many approaches for test suite reduction, selection and prioritization have been proposed, but none of them presents a complete solution [Bozkurt et al., 2013]. The tester needs to do cost-benefit analysis considering several parameters, such as test suite coverage, cost of test suite execution, and also the cost of service invocations.

Program Slicing is a debugging technique to reduce the faulty program only to relevant statements for a particular variable v at a program location l. The slice for a variable x only contains the control and data dependencies of that variable. Although slice needs to be computed only once, but the program dependencies in real programs are too large in size. In case of programs with pointers, the situation becomes worse [Zeller, 2002]. It is important to note that all regression testing approaches which require access to the source-code such as dynamic program slicing are not applicable in SOAs regression testing [Yoo and Harman, 2012].

Zeller introduced a divide-and-conquer debugging technique, which focusses on reducing the test input in order to locate faults in a program [Zeller, 2002]. Interestingly enough, the approach needs no knowledge of the program source-code and data and control dependencies like traditional slicing techniques. All it needs is a passing run and a failing run, and systematic study of program states in both runs. The approach proves to be more useful in practice than program slicing. But, it requires more test runs for a single program, and is slower than program slicing techniques.

In on-line testing, there is a cost of every call to the invoked web service, which can be avoided using offline testing, whereby the tests are executed using mock services. In offline testing, the cost can be reduced at the expense of precision. Our work mainly intersects with test suite reduction techniques, where the focus is to apply mutation-score-based test suite minimization.

We present three different algorithms to reduce the test suite size while keeping the failure detection capabilities with in some pre-defined range.

5.3 Preliminaries

We assume a program under test (PUT) Π written in a certain programming language. For the *execution* of a program Π on a certain *input* I we introduce the function $[\![]\!]$, which returns an *output* O, i.e., $[\![\Pi]\!](I) = O$. The inputs and outputs of a program are set of pairs (x, v) where x is a variable and v its value. Every input and output is only allowed to specify exactly one value for each variable. A *test case* TC for a program Π is a pair (I, O_E) where I is an input and O_E is a set of pairs (x, v) where v is the expected value for a variable x. Note O_E might not specify a value for all variables. However, it is not allowed to specify two values for one variable. A test case $TC = (I, O_E)$ is said to be a *passing test case* for a program Π if the computed output based on input I is not in contradiction with the expected output O_E, i.e., $[\![\Pi]\!](I) \supseteq O_E$. A test case that is not a passing test case is said to be a *failing test case*. A *test suite* TS for a program Π is a set of test cases. In our work we do not make use of distinguishing test cases accordingly to failing or passing. We assume that all test cases of a test suite are equally important for quality assurance.

As already briefly discussed in the introduction, we are interested in eliminating all redundancies from a given test suite. Hence, we have to formally define what redundancy mean. Obviously, redundancy has to do with elements of a test suite that are not relevant for its purpose. Because of the fact that the purpose of a test suite is its capability for detecting a fault in a program, we have to first define a measure for the failure detection capability of a test suite TS for a given program Π. We do this by introducing a function m that maps a test suite and a program to a number that ideally corresponds to the failure detection capability. In practice m might be a certain coverage metric like statement of condition coverage or the mutation score. The redundancy of a test suite corresponds to the degree of test cases that can be eliminated from the test suite without substantially changing the value returned when from function m. The degree of reduction can be

defined as the number of test cases that can be removed from the original test suite. The following definition of reduction makes use of a boundary value α, which is for stating the allowed deviation from the computed value of m for the original test suite.

Definition 13 (Reduction)**.** *Given a test suite TS for a program Π, a function m measuring the failure detection capability of TS, and a boundary value $\alpha \geq 0$. The reduction ρ of TS is defined as follows:*

$$\rho(TS)_\alpha = |TS| - \min\{|TS'| | TS' \subseteq TS \wedge m(TS', \Pi) + \alpha \geq m(TS, \Pi)\}$$

Using reduction we can easily define redundancy as relative value.

Definition 14 (Redundancy)**.** *The redundancy rr of a test suite TS for a program Π, a function m and a boundary value α is defined as follows:*

$$rr(TS)_\alpha = \frac{\rho(TS)_\alpha}{|TS|}$$

Obviously rr is a value between 0 and 1 where 0 indicates that there is no redundancy in a test suite and 1 that all test cases can be removed. Hence, a value of 1 can never be reached in practice.

Before investigating on algorithms for test suite redundancy elimination we did some experiments using BPEL programs and our test suite generation algorithms. In particular we had been interested in having a look at the degree of redundancy of generated test suites. Moreover, we randomly selected subsets of the original test suites in order to have a look at the decline of the failure detection capabilities. In our initial experiments we used mutation score for estimating the failure detection capability. The results of the experiments for program *CALC2* are given in Figure 5.1 and Figure 5.2. In Figure 5.1 the minimum, maximum, average mutation score for subsets of size 1 to 12 are given. The size of the original test suite is 13. We see that there is a decline of the mutation score starting with a subset size of 6 to 7. A similar behavior can be observed from Figure 5.2 where the probability of having a mutation score greater than 0.85 for the different subsets are depicted. We computed the probabilities from randomly selected

10 subsets of a particular size. We see that even for larger subsets there is no guarantee to select the right ones that do not reduce mutation score. However, starting from subsets of size 7 the probability becomes lower than 0.5. The reduction for this test suite and *CALC2* is 7 (=13 - 6) and the redundancy 0.538 ($= \frac{7}{13}$).

5.4 Redundancy elimination

There are many different ways for eliminating redundancies from test suite. In this section, we introduce three different algorithms. Two of them are search-based and employ random selection. The third is the modified version of Delta-debugging algorithm by [Zeller, 2002].

5.4.1 LinMIN Algorithm

In Figure 5.3 we have a search algorithm for randomly selecting a subset that is smaller by one element. There are four inputs to algorithm: the original test suite TS, of program Π, mutation score $\mathbf{m}$, and margin value α.

The algorithm first unmarks all test cases in a test suite. The while loop in step 4 iteratively checks all elements in a test suite until the influence on m is larger than given. Since the test suite size is finite, so it can be argued that the loop is also finite and terminates. Steps 5 and 6 randomly marks one test case and remove it from the test suite. Step 7 checks the influence of the mutation score m of the resulting test suite, that is if no negative influence on m can be observed take this subset and randomly remove one element. Otherwise, the marked test case is added back to the test suite. Since the algorithm randomly removes one element from the test suite, the resulting ordering might not be the optimal, therefore we computed an average of 10 execution for the empirical evaluation of different examples.

The worst-case time complexity of while loop is O($|N|$), where $|N|$ denotes the number of test cases in a test suite TS. The complexity of the function call $\mathbf{m}(TS, \Pi)$ is $O(|S| * |M|)$, where $|S|$ represents the size of the test suite, and $|M|$ is the total number of mutants for a program Π.

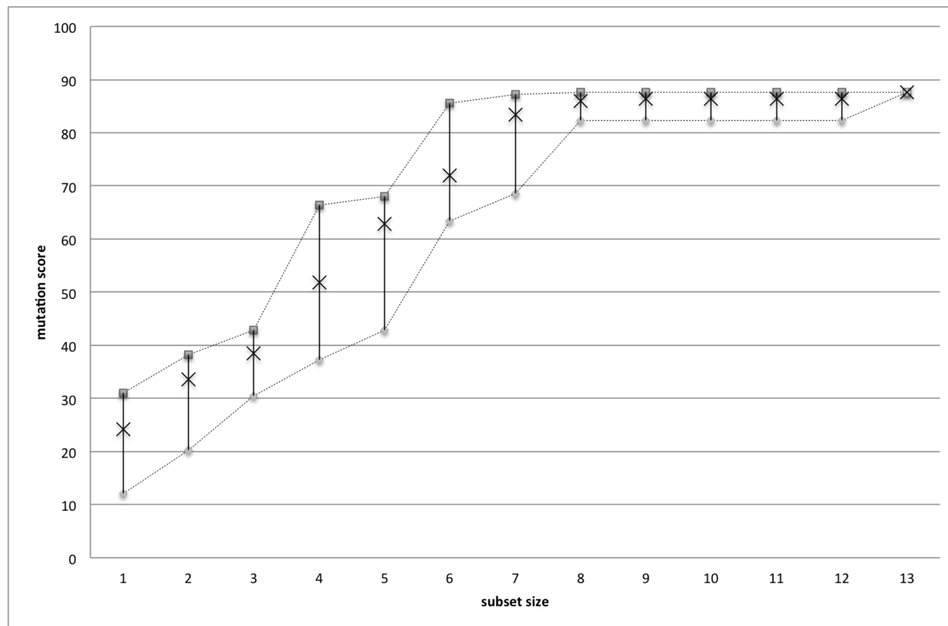


Figure 5.1: The minimum, maximum, and average mutation score for the *CALC2* example with varying subset size

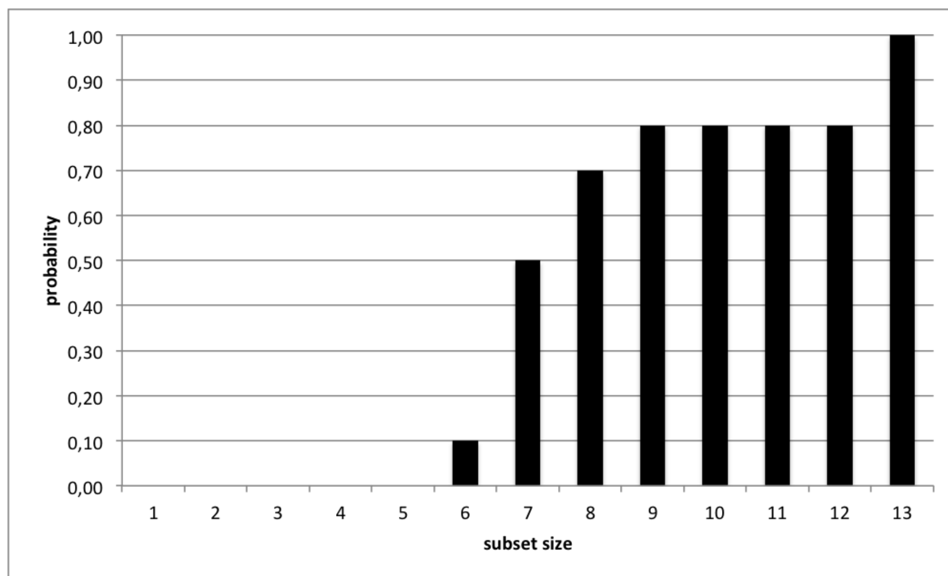


Figure 5.2: The probability for a subset of the original test suite of *CALC2* to have a mutation score larger than 85.0

Theorem 15 (LinMINCorrectness). *LinMIN algorithm illustrated in Figure 5.3 is complete and sound.*

Proof. The while loop in line 4 marks each test case $t_i \in TS$ iteratively and computes the effect of removing each marked test case from TS. The size of TS is finite, hence the loop terminates. The algorithm is sound, because the if condition in line 7 makes sure that the removed test case t_i is added back to TS in case the $(\mathbf{m}(TS, \Pi) + \alpha < maxRR)$. That means the algorithm always produces a subset with a mutation score $<=$ the original test suite TS. Due to random-selection of unmarked elements in a test suite, it might not produce optimal reduced test suite every time. Therefore, the algorithm needs to be executed multiple times to achieve better average reduction. The algorithm ensures a minimized test suite that still provides an acceptable failure detection rate according to the function **m**.

□

5.4.2 BinarySearch Algorithm

BinarySearch algorithm shown in Figure 5.4 implements binary search over the subset size. First, a subset of size $|TS|/2$ is selected randomly from TS. If the resulting subset does not influence m substantially, select a subset of size $|TS|/4$. Otherwise, select a subset of size $|TS| * 3/4$, etc. The objective is the same, that is to ensure a minimized test suite that still provides an acceptable failure detection rate according to the function **m**. However, in contrast to *LinMIN* algorithm, the execution time is much smaller, as the search space is reduced to half in every step. The downside of the approach is that it might not always find the subset $set(t)$ with $maxRR$. As a result, the test suite returned might not alway have the desired mutation score. However, the chance reduces by increasing the subset size as discussed in the empirical evaluation section 5.5. The worst-case time complexity of binary search is O(logN), where it takes logN time to find the required subset in worst-case.

5.4.3 Delta-Debugging Algorithm

Figure 5.5 illustrates the tailored version of Delta-Debugging. The original algorithm [Zeller, 2002] automates the “scientific” way of debugging. It takes a fairly large input that causes failure and aims at finding a smaller input that still produces the fault. The target is to find software changes that lead to failures.

The original delta-debugging algorithm takes a program and a faulty test case as an input. The faulty test case includes all possible changes, and aims at finding the minimal change set (configuration) causing the fault. It marks each test one of the three statuses: passing, failing or unresolved. It starts by dividing the input into two parts, and continues to increase the granularity of the search space until the failure causing input is identified. The approach is based on *divide and conquer* principle, and helps in locating faults in a software. The worst-case complexity of original minimization algorithm is $O(|c|^2)$, where c represents number of changes to the program. In the best case it has same complexity as that of a binary search algorithm.

In our case, we customize the delta-debugging algorithm for testing purposes. Using the delta-debugging minimization algorithm, we compute function m of each subset ts of a test suite TS. We mark the selected subset as either failing or passing. That is, a test is *failing*, if its mutation score combined with the deviation margin α is equal or greater than the $maxRR$ of the test suite TS. Otherwise the subset is marked as *passing*. The search continues with the failing subset of test cases until a minimum subset of test cases is found, which still provides the required mutation score.

5.5 Empirical Evaluation

The target of an empirical evaluation was to investigate the degree of redundancy of generated test suites. Moreover, we randomly selected subsets of the original test suites in order to have a look at the decline of the failure detection capabilities. As discussed in Section 5.3, a redundancy of 0.538 was observed for CALC2 example. As a second step, we applied algorithms discussed in Section 5.4 to compare their reduction capabilities. There are

```
1:  procedure LinMIN(TS = {t_1, ..., t_n}, a program Π, a function m, and
    a boundary value α)
2:      Let maxRR ← m(TS, Π).
3:      Unmark all test cases in TS.
4:      while There exists an unmarked test case t_i ∈ TS do
5:          Mark t_i.
6:          Remove t_i from TS.
7:          if (m(TS, Π) + α < maxRR) then
8:              Add t_i to TS.
9:          end if
10:     end while
11:     return TS
12: end procedure
```

Figure 5.3: LINMIN - A linear search procedure for test suite minimization

two important parameters, that is, *alpha* and the *subset size*. The parameter $alpha$ shows the boundary value for desired mutation score. That is $alpha = 0$ for $maxRR = 85$ would mean, that only subsets of mutation score $> 85\%$ would be accepted. By relaxing the value to, lets say 5, we make the algorithm include sets with mutation score$>$ 80% in the solution. Note that initially we only considered *subset size*, e.g. 10 permutations of the original test suite in order to keep the execution time minimum.

Table 5.1 states the test suite reduction achieved by all three redundancy algorithms. The boundary value is denoted by α, and the varying subset size is labeled $|Set|$. The number of test values generated by Minion constraint solver for each test case is represented by $|Sol|$. Similarly, the original test suite size is denoted by $|TS|$. The reduced test suite size obtained using redundancy algorithms is labeled by $|TS|_{LM}$, $|TS|_{BS}$, and $|TS|_{DD}$ respectively. Likewise the time taken in milliseconds by these algorithms is represented by $T_{LM}(ms)$, $T_{BS}(ms)$, and $T_{DD}(ms)$ respectively.

1: **procedure** BinarySearch($TS = \{t_1, \ldots, t_n\}$, a program Π, a function **m**, and a boundary value α)
2: Let $maxRR \leftarrow \mathbf{m}(TS, \Pi)$.
3: Set $low = 0$
4: Set $high = maxSetSize - 1$
5: **while** $low <= high$ **do**
6: mid = $low + (high - low)/2$
7: $set(ts) \leftarrow rand(set(TS, \Pi))$
8: **if** $(\mathbf{m}(\mathbf{set}(ts), \Pi) + \alpha < maxRR)$ **then**
9: $low = mid + 1$
10: **end if**
11: **if** $(\mathbf{m}(\mathbf{set}(ts), \Pi) + \alpha > maxRR)$ **then**
12: $high = mid - 1$
13: **end if**
14: **if** $(\mathbf{m}(\mathbf{set}(ts), \Pi) + \alpha == maxRR)$ **then**
15: **return** $set(ts)$
16: **end if**
17: **end while**
18: **end procedure**

Figure 5.4: BinSearch – A Binary search procedure for test suite minimization

1: **procedure** Delta-Debugging($TS = \{t_1, \ldots, t_n\}$, a program Π, a function **m**, and a boundary value α)
2: Let $maxRR \leftarrow \mathbf{m}(TS, \Pi)$.
3: Define the testing function **test** as follows:

$$\mathbf{test}(ts) = \begin{cases} \times & \textit{if } (\mathbf{m}(\mathbf{set}(ts), \Pi) + \alpha \geq maxRR) \\ \checkmark & \textit{otherwise} \end{cases}$$

4: **return** $\mathbf{ddmin}(\langle t_1, \ldots, t_n \rangle)$
5: **end procedure**

Figure 5.5: DELTAMIN – Using delta debugging for test suite minimization

Table 5.1: Experimental results for Redundancy Reduction algorithm.

Prog	α	$\|Set\|$	$\|Sol\|$	$\|TS\|_{MAX}$	$\|TS\|_{LM}$	$T_{LM}(ms)$	$\|TS\|_{BS}$	$T_{BS}(ms)$	$\|TS\|_{DD}$	$T_{DD}(ms)$
Calc2	0	10	1	13	11	4271	10	3561	6	3112
	5	10	1	13	11	3957	10	3699	4	4975
	10	10	1	13	11	4642	7	3829	4	4237
Calc2	0	10	3	33	24	27417	20	10047	6	8418
	5	10	3	33	17	20276	13	6962	5	9033
	10	10	3	33	9	19115	12	7754	4	7731
Calc2	0	10	3	33	23	22362	18	6918	6	6193
	0	20	3	33	22	29752	19	6813	6	7305
	0	30	3	33	23	38704	17	7623	6	7439
Calc2	0	10	5	53	33	91921	39	6346	6	10237
	5	10	5	53	33	90832	23	6575	5	8082
	10	10	5	53	19	90352	23	8116	4	7983
Calc2	0	10	5	53	37	83582	39	5877	6	9074
	0	20	5	53	37	150439	39	9160	6	10431
	0	30	5	53	33	246889	39	10064	6	10825

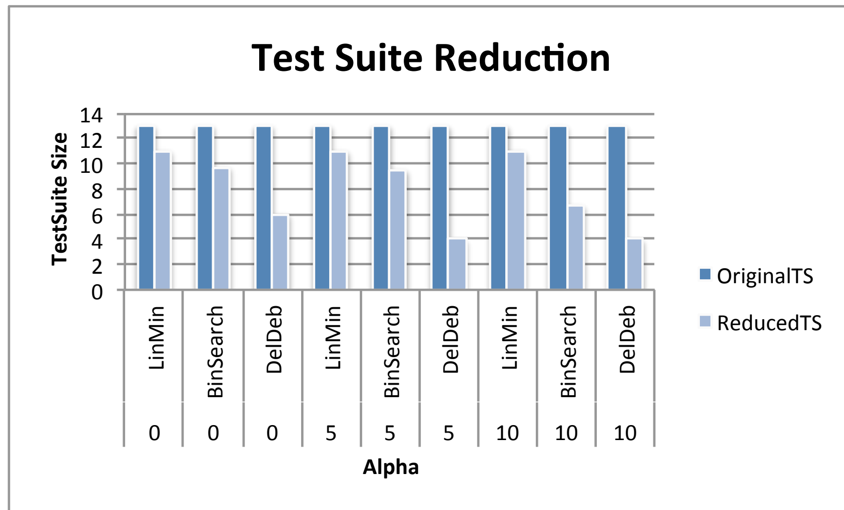


Figure 5.6: The test suite reduction for the *CALC2* example with varying alpha

Figure 5.6 illustrates the reduction attained for single solution. It can be seen that the LinMIN algorithm 5.3 does not reduce the test suite by varying *alpha*, whereas the BinarySearch algorithm 5.4 reduces the test suite size considerably. However, the Binary Search might not always find a subset with desired mutation score. The Delta-Debugging algorithm 5.5 outperforms the other two with maximum reduction of size 4. The execution times are reported in first three rows of Table 5.1.

This led us to another question of increasing the solution size (number of variable assignments for a feasible path) in order to investigate the effect on the maximum mutation score of the generated test suite, and the performance of redundancy algorithms in case of larger subsets. Therefore in third step, we changed our test-case generation algorithm by allowing multiple tests per path. Prior to that, we always had one satisfying assignment per feasible path. For experiments, we tailored the algorithm to allow for more than one number of solutions per feasible path. The purpose was to compare the mutation score of different test suites.

The AllRandomPaths algorithm 5.9 takes three parameters, i.e., *MaxLen*, *setSize*, and *solSize*. The *MaxLen* parameter defines the length of the maximum path that could be searched by the algorithm. The *setSize* determines

the number of subsets to be generated randomly. The generated subsets represents the number of test suites. Note that, we only execute the test suite of max. subset size for the analysis. The third parameter *solSize* defines the number of test values generated by the constraint solver. For our experiments, we chose solutions of size 1, 3 and 5. That means, for one test case, we asked constraint solver for different test values, and study the impact on the attained coverage. The algorithm 5.9 is a sort of an amalgam of previous two algorithms, i.e., AllPaths 4.4 and RandomPaths 4.22.

It is observed that by allowing maximum solutions of size 3 per feasible path, the original AllRandomPaths algorithm 5.9 generated in total 322 test suites, with a maximum subset size of 33 for the same maximum length 20. A slight increase in the mutation coverage (88%) is observed. Figure 5.7 illustrates the minimum, maximum, and average mutation score for the *CALC2* example larger than (85%). Starting from $|TS| = 13$, there is noticeable decline in the mutation score to 85.39 avg. Similarly in Figure 5.8 it can be seen that for larger subsets there is no guarantee of selecting subsets that do not reduce mutation score. However, starting from $|TS| = 13$, the probability becomes lower than (80%). The reduction for this test suite and *CALC2* is 20 (=33 - 13) and the redundancy 0.606 ($= \frac{20}{33}$).

The results from *LinMIN*, *BinarySearch* and *Delta-Debugging* algorithms can be looked up in Table 5.1. Note that the alpha is initially set to zero and the subset size is varied from 10 to 30. Afterwards, the parameter alpha is varied between 0 and 10 while keeping the subset size to 10. Figures 5.12a and 5.12b show the reduction attained by redundancy elimination algorithms with varying boundary value alpha and subset size.

It is observed that upon changing alpha, we get further reduction in DeltaDebugging algorithm, since we need less number of tests to attain the same maxRR. Similarly, *LinMIN* also performs better on changing alpha, but shows no improvement on varying the subset size. Also, this algorithm takes longer to compute than the other two algorithms. *BinarySearch* algorithm, on the other hand, always finds a subset with "maxRR" on increasing the subset size, but it does not always find the required set upon changing alpha. Although the addition of multiples tests per path increases the redundancy in the test suite, but *LinMIN* and *BinarySearch* algorithms converge better with larger test suite. Interestingly, the *Delta-Debugging* algorithm is

not effected by increasing the test suite size. The execution time is however doubled for all algorithms by increasing the subset size.

Similarly, upon increasing the number of test inputs to 5 for each feasible test path (originally 13), the mutation coverage does not increase, however, the maximum number of tests suites is increased to 521 with a maximum subset of size 53. The minimum, maximum, average mutation score for subsets of size 1 to 52 are reported in Figure 5.10. From $|TS| = 23$, there is noticeable decline in the mutation score to 85.79 avg. Also, the probability of having a mutation score greater than 0.85 for different subsets is shown in Figure 5.11. The reduction for this test suite $|TS| = 23$ and *CALC2* is 30 (=53 - 23) and the redundancy 0.566 ($= \frac{30}{53}$).

One issue was observed, i.e., the number of maximum killed mutants with 3 solutions per path is 270. The number reduces to 269 with 5 solutions per path. Originally, in the test suite with single solution we had 268 maximum killed mutants. So, the mutation score does not really increase by adding more tests to the suite. Figures 5.12c and 5.12d show the reduction attained by redundancy elimination algorithms by varying boundary value alpha and subset size. Both *LinMIN*, *BinarySearch* perform better on increasing the alpha size, but *Delta-Debugging* is insensitive to changes in the solution size, alpha and subset size. However, the execution times increases exponentially on increasing subsetSize (see Table 5.1).

5.6 Conclusions

In this Chapter we discussed the regression testing limitations in terms of SOA-based programs along with the related work done in this domain. Most of the regression testing techniques are based on the structure of a program, which might not be possible for all SOA-based programs. Similarly, there are other concerns like traditional approaches assume deterministic inputs, which is impossible for dynamic SOAs implementations. We evaluated test suite redundancy of different test suites and applied redundancy elimination algorithms to remove redundant test cases while maintaining the failure detection capabilities within specified limit.

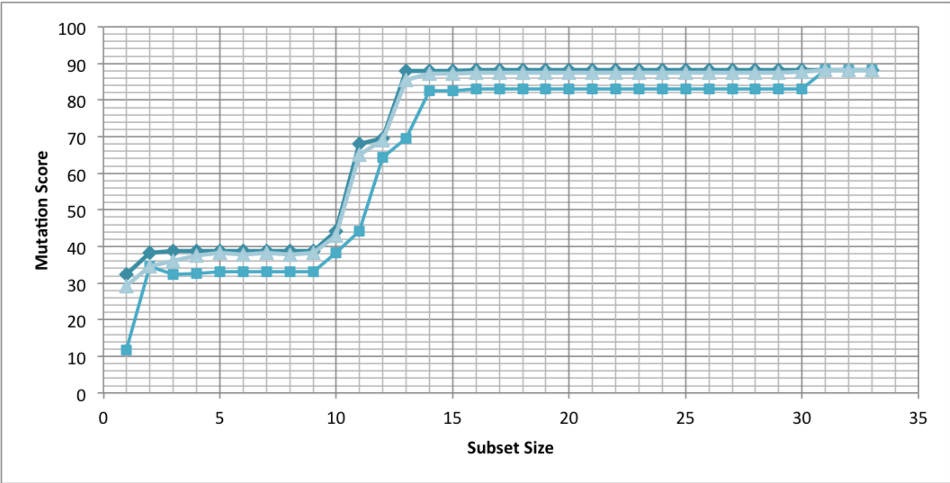


Figure 5.7: The minimum, maximum, and average mutation score for the *CALC2* example with solSize=3 and varying subset size

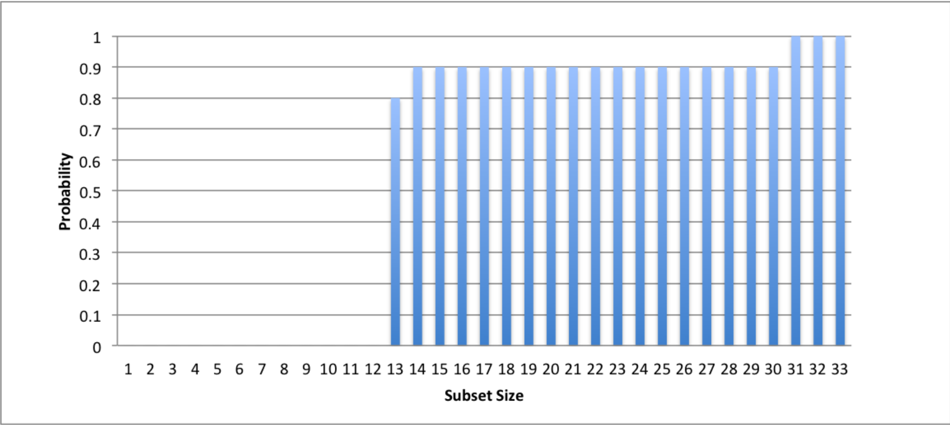


Figure 5.8: The probability for a subset of the original test suite of *CALC2* with solSize=3 to have a mutation score larger than 85.0

1: **procedure** ALLRANDOMPATHS(G, $MaxLen$, $setSize$, $solSize$)
2: initialize test suite $S \leftarrow \emptyset$
3: compute the set P of all paths π s.t. $|\pi| \leq MaxLen$.
4: **for** each path $\pi \in P$ **do**
5: check the satisfiability of path constraints $C(\pi)$
6: **if** $C(\pi)$ is satisfiable **then**
7: **for** call Constraint Solver for solSize **do**
8: add each satisfying assignment (a test case) to S
9: **end for**
10: **end if**
11: **end for**
12: store test suite S.
13: generate random subsets $sets$ of the test suite S s.t. $|set| \leq setSize$
14: **return** subsets $sets$ of the original test suite S
15: **end procedure**

Figure 5.9: AllRandomPaths – Using random subsets for test suite generation

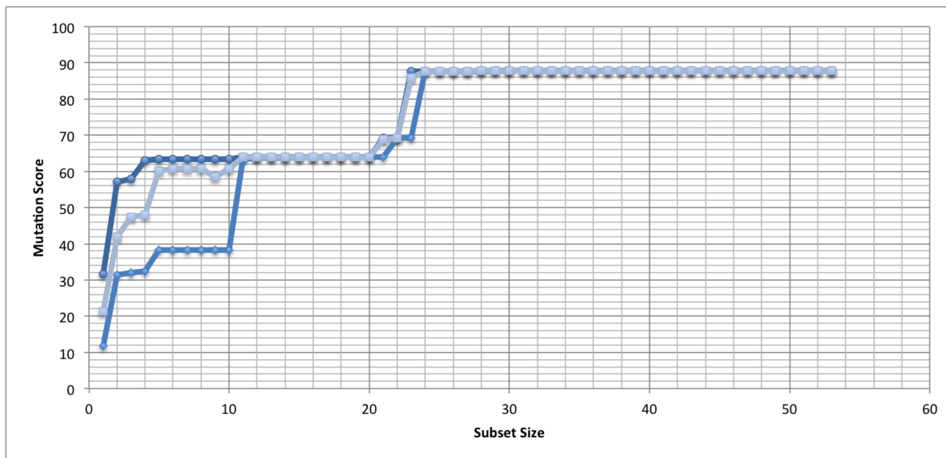


Figure 5.10: The minimum, maximum, and average mutation score for the *CALC2* example with solSize=5 and varying subset size

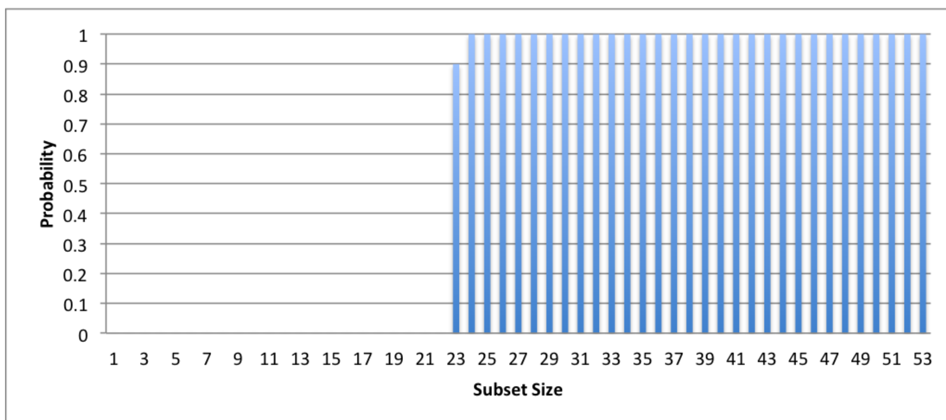
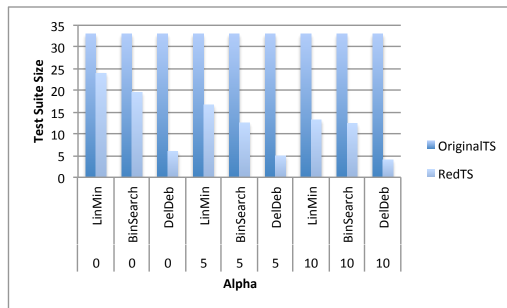
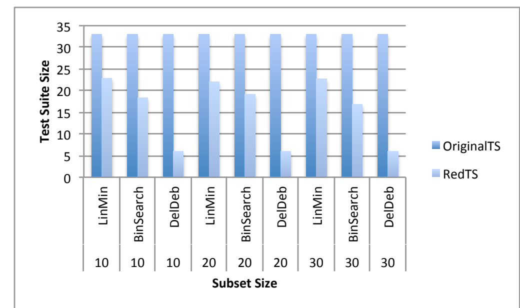


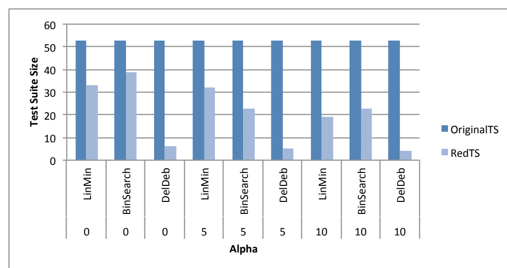
Figure 5.11: The probability for a subset of the original test suite of *CALC2* with solSize=5 to have a mutation score larger than 85.0



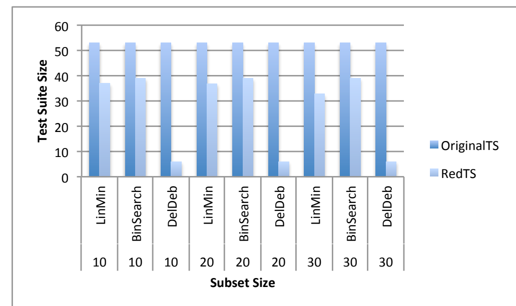
(a)*CALC2* example with varying alpha for SolSize=3



(b)*CALC2* example with varying subset Size for SolSize=3



(c)*CALC2* example with varying alpha for SolSize=5



(d)*CALC2* example with varying subset Size for SolSize=5

Figure 5.12: The test suite reduction for the *CALC2* example for multiple tests per feasible path

6 Model-Based SOA Debugging

6.1 Introduction

While Testing is focussed on finding faults in a given system, *diagnosis* is about figuring out possible reasons behind observed faults. Once the fault has been localized, an ideal diagnostic engine should also provide repair possibilities for occurred faults. Unfortunately, due to ever increasing system maintenance costs, this three step process is neglected altogether [Nica, 2010]. It becomes even more challenging when it comes to service-oriented architectures as discussed by Friedrich et al. in [Mayer et al., 2012]. This is mainly because of the fact that SOA-based processes are designed as service-providers rather than service-owners. This particular feature of any SOA-based process, although on one hand provides loose-coupling between service-components, complicates the diagnostic activity on the other hand. Hence, the diagnostic approach must deal with the problem of partial behavior.

Figure 6.1 depicts the flow graph of a well-known software engineering problem modeled as a sequential BPEL process. The corresponding program takes the triangle's sides' lengths as input, and classifies the triangle to be equilateral, isosceles, or scalene. The example makes use of an external web service, called *InvokeValid*. The program takes three inputs, i.e., the lengths of the triangle's sides a, b, and c. Let us assume that we want to classify the isosceles triangle such that a=2, b=2, and c=1. Unfortunately and unexpectedly, however, our "faulty" program returns an error message instead ("=NEG", respectively "No triangle"), due to a programming error. That is, for *condA* we wrote $(a_0 < 0) \wedge (b_0 > 0) \wedge (c_0 > 0)$ instead of $(a_0 > 0) \wedge (b_0 > 0) \wedge (c_0 > 0)$, such that the program takes the corresponding else branch for *condA* due to the condition evaluating to *false* instead of *true*

for our example. Our fault detection approach employs dynamic slicing on the constraint-representation of the BPEL flow graph. The target is to find contradicting variable assignments responsible for the fault.

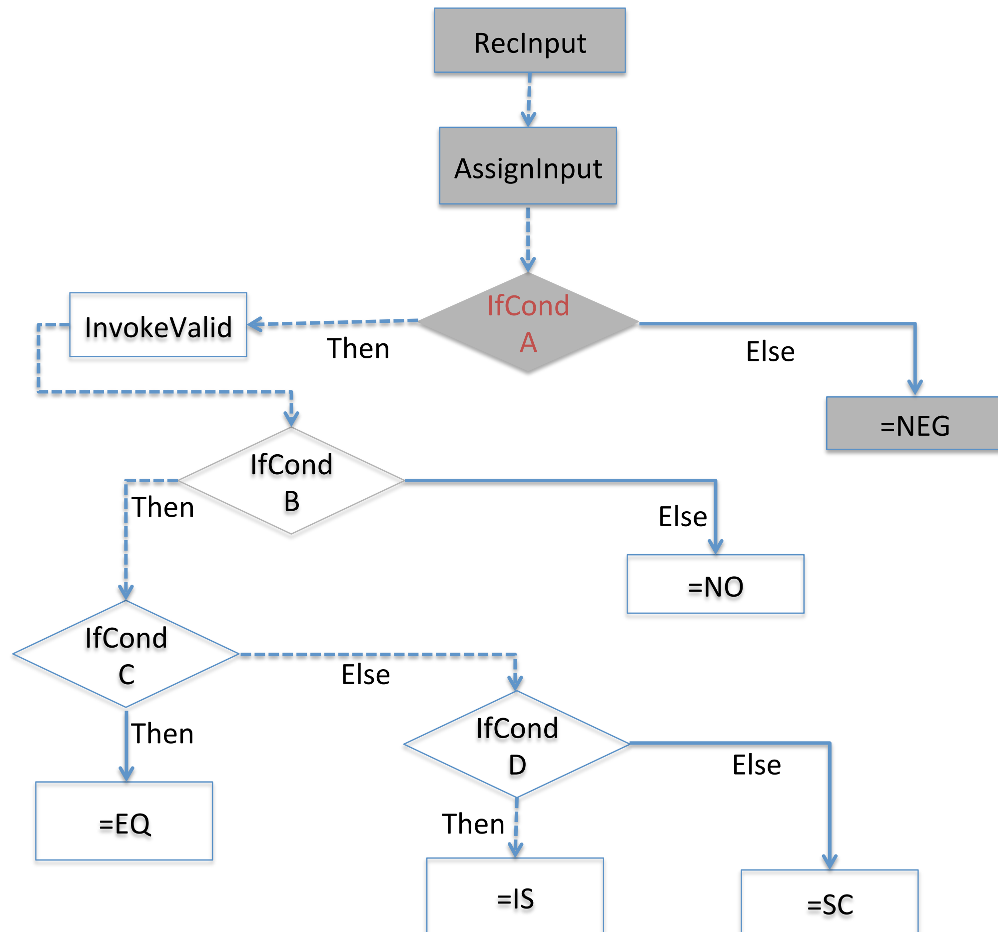


Figure 6.1: BPEL Flow Graph of the Triangle Example Process

The Chapter is organized as follows: The basic definitions in the context of our approach are revisited in Section 6.2. Section 6.3 describes the underlying architecture of our approach. Section 6.4 explains the experimental setup and results obtained from the empirical evaluation. Conclusions are presented in Section 6.5.

The content of this chapter has been published in following papers:

- Functional Diagnosis of SOA BPEL Processes [Hofer et al., 2014]. Here, the focus was to derive a debugging approach for finding functional faults in BPEL processes. The key issue addressed was to diagnose partial behavior models of BPEL compositions.
- Focussed Diagnosis for Failing Software Tests [Hofer et al., 2015]. In this paper, we generalize our debugging approach using a control flow graph model of a faulty program.

6.2 Definitions

Unlike testing, our diagnostic model represents the sequential business processes only. The BPEL model is represented by a *Control Flow Graph* G, where further information can be easily added to functions $\gamma_C(v)$ and $\gamma_A(v)$ mapping vertices $v \in V$ to a statement's conditions and assignments respectively. With G capturing the control flow structure of a program, a path π in G defines a valid scenario.

The path constraints and the feasible path for our example are shown in example 1 and example 2 respectively. With the given inputs, as shown in the example 2, the path constraints highlighted in Figure 6.1 are satisfiable. Since a wrong path π is executed, the debugging approach focusses on finding the constraints responsible for the fault.

Example 1.
$a_0 = in1$
$b_0 = in2$
$c_0 = in3$
$(a_0 < 0) \wedge (b_0 > 0) \wedge (c_0 > 0) = false$
$output_0 = \text{"No triangle"}$

Example 2. *Our running example's input* $I = \{a = 2, b = 2, c = 1\}$ *results in the grey-shaded path as of Figure 6.1.*

For the debugging problem, we require additional information about the failed scenario, such as, the trace of the faulty program execution and the test case, which triggered the particular path in a flow graph G. We can formalize both trace and a test case as follows:

Definition 15 (Trace)**.** *The trace T in G for a program execution Π is defined as the path in G allowed for the input part of Π.*

Definition 16 (Test Case)**.** *A* test case *is a tuple $\tau = (I, O)$ where I (sometimes we will write also $I(\tau)$) is the set of value assignments to input variables, and O is the set of expected values for the output variables. A test case fails iff the observed output O' deviates from O, s.t. $O' \neq O$.*

There are three main components needed to model the debugging problem: a set of components *COMP*; a set of system descriptions *SD* representing the faulty behavior of the system; and observations *OBS* about the expected system behavior. In this way, the model-based debugging swaps the roles of a model and observations as compared to model-based diagnosis originally proposed by [Reiter, 1987]. In MBSD, the model, i.e., the execution trace T is assumed to be incorrect, and the observation *OBS* denote the correct behavior of the system. In this regard, the activities included in the execution trace T define the system components *COMP*, and the path constraints $C(\pi)$ as shown in the example 1, would represent the system descriptions *SD*. Similarly, the predicate $\neg AB(c_i) \Rightarrow NominalBehavior(c_i)$, defines the correct behavior of the constraints, and *OBS* denote the expected system behavior. The system is considered to be at fault iff $SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP\}$ is inconsistent.

Definition 17 (Diagnosis)**.** *A* diagnosis *for (SD, COMP, OBS) is a subset-minimal set $\Delta \subseteq COMP$ such that $SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \setminus \Delta\}$ is consistent.*

In the context of software debugging, the program slicing has been widely discussed in academia in order to locate software bugs [Tip, 1995]. *Program Slicing* refers to the computation of all possible values of a variable v at any specific program location. [Weiser, 1982] first introduced the term "slices" as a programmer practice to debugging. The idea was simple, yet powerful, as it allows to concentrate only on program statements, which actually influence the particular variable at a specific program location. Hence, all other statements become irrelevant in the debugging context, thereby reducing the program size to be analyzed.

There are two major types of slicing: static and dynamic. The approach suggested by [Weiser, 1982] is a static one, as it considers all possible inputs for the variable of interest v. [Korel and Laski, 1988] suggested an approach, which only takes into account the run-time input given to the program, while computing the slices for a particular variable v. The added advantage of this approach is further reduction in the slice size, along with the precise tracking of dynamic data structures such as arrays[Tip, 1995].

In another work, [Korel and Rilling, 1998] compared different dynamic slicing approaches in practice. However, it may happen, that a dynamic slice is not complete because of non-executable paths in a given control-flow graph. There are some extensions, like relevant slicing [Zhang et al., 2005] to cater for this problem. Also, computing a minimal slice is an undecidable problem[Korel and Rilling, 1998]. For a detailed survey of different program slicing techniques, we refer the interested reader to the survey done by [Tip, 1995]. In the following, we explain our model-based debugging approach for locating functional faults in (BPEL) flow graph.

Although model-based diagnosis was initially introduced to diagnose faults in hardware systems, it can also be used effectively to diagnose software faults [Wotawa, 2002]. In his work, he showed how the model-based diagnosis [Reiter, 1987]and program slicing[Weiser, 1982] can be merged together to avail the benefits of both techniques. He argued that slices for particular variables can represent conflicts in the hitting-set algorithm.

In this way, we can reduce the time needed to compute conflicts out of a model. Since slices are not minimal, model-based diagnosis can help in finding single fault diagnosis. The approach proves useful in case of dependency-based model. Our approach adapts the dependency-based model approach in two major ways: first we have partial model of the BPEL composition. Second, instead of a hitting-set algorithm, we model the debugging problem as a constraint-satisfaction problem.

6.3 The Debugging Approach

Similar to Testing, our diagnosis approach is also constraint-based. The constraints systems have been very popular in artificial intelligence community. They are, nowadays, widely used in verification, testing, configuration of phone networks, recommended systems, as well as in the diagnosis of the hardware and software systems. There are various constraints solvers (Minion, Choco, Yices) available in the market, to meet the needs of a variety of diverse application domains [Nica, 2010].

Our central reasoning concept is based on the constraint representation of a BPEL flow graph as of definition 1. However, we tailor the approach already discussed for the testing of SOA applications as discussed in Chapter 4. Our approach is similar to the debugging approach presented by [Hofer and Wotawa, 2012] and [Wotawa et al., 2012]. Although both approaches employ dynamic slicing: the former approach make use of execution traces to compute slices using hitting-set algorithm [Reiter, 1987], and a constraint solver to compute the minimal diagnosis; the latter, on the other hand, derives constraints from the source-code of a program. Both of these approaches employ the dependency-graph analysis for computing constraints. Our work is an amalgam of the two in a way that, we derive constraints from the BPEL source-code, but instead of modeling all possible paths in a flow graph, we rely on the particular execution trace. In addition to that our work employs modeling of the partial information of calls to external web services. The previously discussed approaches work on the dependency-based model of the underlying program. However, the variable-dependency analysis is not enough for our model [Yan et al., 2009]. For that purpose, we tailor the constraint solving approach proposed by these approaches as follows.

The control flow graph contains both basic and structured activities, which in our case can be annotated using pre- and post conditions. These contracts are particularly helpful in reasoning about the partial functionality of invoked web services. Since the BPEL process logic is available, one might consider our approach to be a white-box, but, as a matter of fact a large portion of a system's functionality is hidden in external web services. Because of that, we prefer to name our approach as grey-box. A BPEL process only knows

about inputs/outputs of external web services, hence the diagnosis of these hidden/grey portions of the process becomes harder. [Mayer et al., 2012].

Before going into the definitions, we present an overview of the suggested approach in Figure 6.2. Consider this to be a control flow graph representation of a BPEL process, where the highlighted path is faulty but a feasible path π, as per definition 2, which meets all path constraints defined in 3 along π. We diagnose single paths rather than a complete model. And, everything aside π is a black box. In the first step, the path π is translated to constraints, which become components c_i in our model. The behavior of these components is represented by the predicate $AB(c_i)$. Moreover, the trace T is divided into segments based on branching points. For that, we introduce the intrace variable $intrace_i$, which basically determines if a segment s_i is part of the diagnosis or not.

Our debugging approach assumes that we have an execution trace T which contradicts the expected behavior of a failing test case τ. In order to diagnose the root cause of any deviation from the expected execution path, we segment the execution trace T using a boolean variable $intrace_i$ in order to keep track of the branching conditions. Basically, the purpose is to identify those segments in the trace T, which fall out of the desired path. The first segment has necessarily to be a part of each and every scenario.

Definition 18 (Segmented Path). *A segmented path (π, S) in G is a tuple such that π is a path, and S is the set of branching points s_i of π in G that divide the path into enumerated segments as follows. The branching points s_i are numbered according to their distance from π_0, the enumeration starting with 1. The first segment, numbered 0 starts at π_0 and ends with s_1. Starting with segment 1, a segment i starts right after s_i, and ends with either s_{i+1} or the path's end.*

Example 3. *The path constraints from Example 1 are divided into 2 segments. The first segments contains:*

1 $a_0 = in1$
2 $b_0 = in2$
3 $c_0 = in3$
4 $(a_0 < 0) \wedge (b_0 > 0) \wedge (c_0 > 0) = false$

The second segment contains:

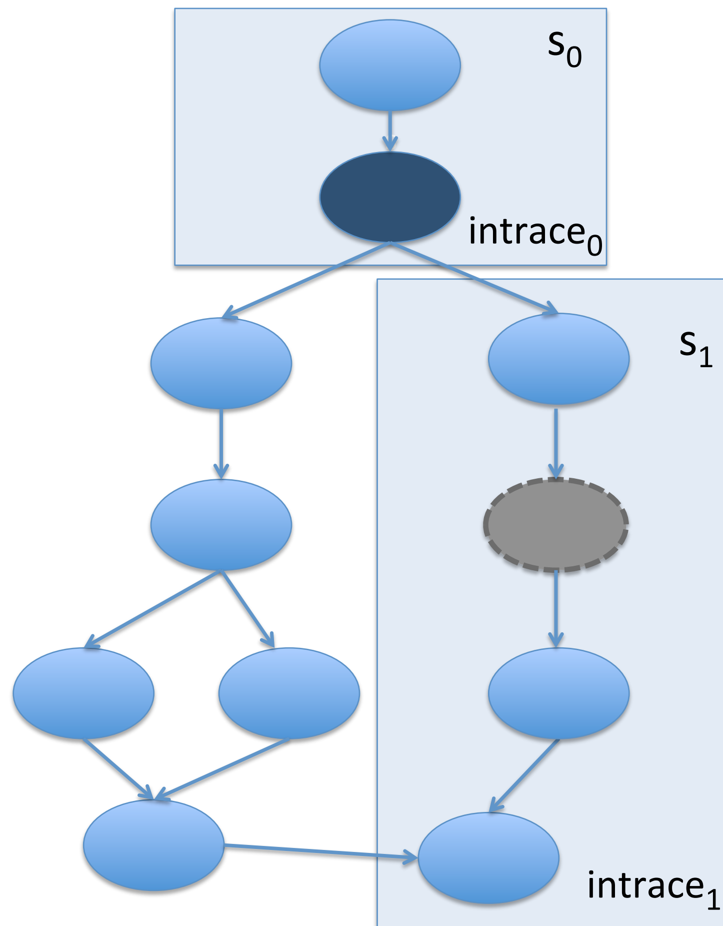


Figure 6.2: Annotated Flow Graph Representation

5 $output_0 =$ *"No triangle"*

Now let us define the specific diagnosis problem and our corresponding technical implementation.

Definition 19 (DiagnosisProblem). *A* control flow graph diagnosis problem *is a tuple* (G, τ)*, where* G *is a CFG, and* T *is a trace from a failing test case* τ*.*

The debugging problem can be modeled using the following constraint satisfaction encoding. The main difference of our approach with the one discussed in [Wotawa et al., 2012] is the introduction of *intrace* variable. The target is to differentiate between different segments in the execution trace of an unexpected output.

Definition 20 (DiagnosisCSP). *A constraint satisfaction encoding* $CSP(G, \tau)$ *for a control flow graph diagnosis problem* (G, τ) *is defined as follows:*

1. *let* (T, S) *be a segmented path for the trace* T *for* τ *in* G
2. *let the set of variables* V *contain all the variables in* $C(T)$*, as well as Boolean variables* $intrace_i$ *for all of* (T, S)*'s segments as of Definition 18.*
3. *let* $C'(T)$ *be the path constraints* $C(T)$ *altered s.t. for any individual constraint* $c \in C(T)$ *of segment* i*, we add a predicate* AB_c *and do as follows:*
   a) *if* c *is not a branching constraint from some* $s \in S$*, then* c *gets replaced by* $\neg intrace_i \vee AB_c \vee c$*.*
   b) *if* c *is the branching constraint from* $s_i \in S$*, then* c *gets replaced by the following set of constraints:* $\neg intrace_i \rightarrow \neg intrace_{i+1}$*,* $AB_c \rightarrow \neg intrace_{i+1}$ *and* $\neg intrace_i \vee AB_c \vee (c \leftrightarrow \circ intrace_{i+1})$ *- with* $\circ$ *being the expected polarity of* $intrace_{i+1}$ *when constraint* c *is satisfied.*
4. *then let* $CSP(G, \tau)$ *be the combined constraints of* $C'(\pi)$ *and* τ *as well as the constraint* $intrace_0$*.*

Now let us have a look at the corresponding CSP for our running example.

Example 4. *The constraints as of Def.* 20 *for our running example are as follows:*

$$\neg intrace_0 \vee AB_1 \vee a_0 = in1$$
$$\neg intrace_0 \vee AB_2 \vee b_0 = in2$$
$$\neg intrace_0 \vee AB_3 \vee c_0 = in3$$
$$\neg intrace_0 \rightarrow \neg intrace_1$$
$$AB_4 \rightarrow \neg intrace_1$$
$$\neg intrace_0 \vee AB_4 \vee (a_0 < 0) \wedge (b_0 > 0) \wedge (c_0 > 0) \leftrightarrow \neg intrace_1$$
$$\neg intrace_1 \vee AB_5 \vee output_0 = \text{“No triangle”}$$
$$in1 = 2$$
$$in2 = 2$$
$$in3 = 3$$
$$output_0 = \text{“isosceles”}$$
$$intrace_0$$

As discussed earlier, we make use of the constraint-based representation [Wotawa et al., 2012] for computing diagnoses. However, we adapt the constraint satisfaction problem according to definition 20, where constraints are directly derived from the BPEL source code. [Nica et al., 2013] showed that such direct approaches might perform better than the diagnosis approach based on conflict-based computation. Algorithm shown in Figure 1 illustrates the corresponding computation of diagnoses.

Theorem 16 (GetDiagnosesCorrectness). *$GetDiagnoses(M, n)$ Algorithm illustrated in Figure 1 is correct.*

Proof. Algorithm starts with $CSP(G, \tau)$ and let the solver determine satisfying assignments that are limited in active abnormal predicates.The line 3 of the algorithm is used to effectively limit the sum of active abnormal predicates to the desired cardinality i. A single query to the solver delivers all the solutions for a specified cardinality. Thus, starting with a cardinality of one, we increment the cardinality limit until we reach the given upper bound for the diagnosis cardinality. All solutions are stored (Line 5) and the constraints for the corresponding blocking clause (i.e. basically a logic or over the negated abnormal predicates in a diagnosis) are attached to the model, in order to ensure the diagnoses' subset-minimality (Line 6). Hence, Lines 5 and 6 ensure the completeness and soundness of Algorithm. Due to for loop in Line 2, Algorithm terminates in finite time. The worst-case complexity would be exponential. □

Algorithm 1 *GetDiagnoses*(M, n)

Input: A constraint system M and the upper bound of the diagnosis cardinality n
Output: All minimal diagnoses up to the predefined cardinality n

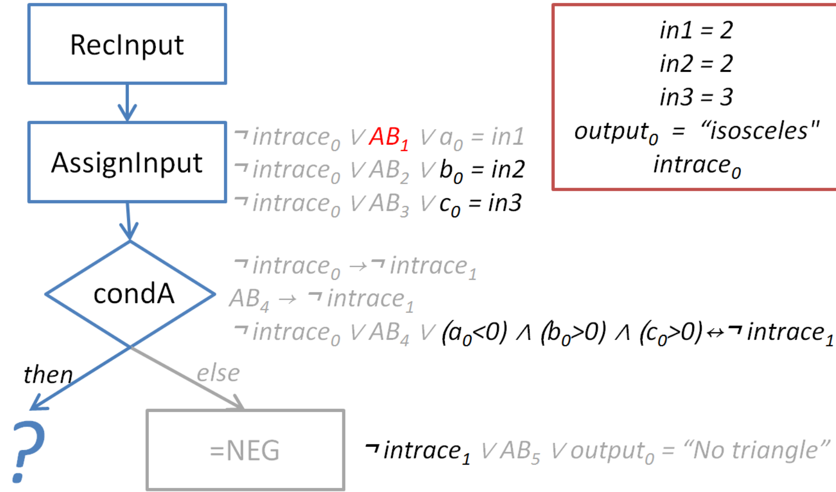
1: Let DS be $\{\}$
2: **for** $i = 1 \rightarrow n$ **do**
3: $\quad \mathrm{M}' = M \cup \left\{ \left(\sum_{j=1}^{|\pi|} \mathrm{AB}_j \right) == i \right\}$
4: $\quad D = \mathrm{Solve}(\mathrm{M}')$
5: $\quad DS = DS \cup D$
6: $\quad M = M \cup \neg \left(\vee_{d \in D} (d) \right)$
7: **end for**
8: **return** DS

Example 5. *Deploying algorithm 1 on the CSP as of Definition 20 for our running example results in the computation of the three single fault diagnoses* AB_1, AB_4, *and* AB_5.

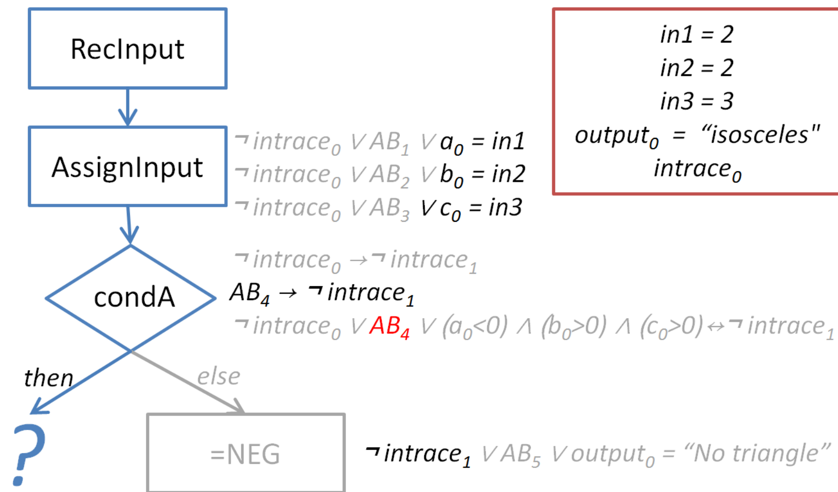
Using our diagnosis approach, we can tell if a particular segment is a part of the trace or not. For example, as shown in Figure 6.3, the disabled segments by the *intrace* variable are displayed in grey. It is also clear, that there are three single-fault explanations for the particular scenario: First, the input value might be wrong; Second, the branching constraint might be incorrect; Third, we might have an incorrect output value.

Definition 21 (Extended Diagnosis)**.** *An* extended diagnosis *for a control flow graph diagnosis problem* (G, τ) *as of Def. 19 is a tuple* $(\Delta, T_S, INTRACE)$, *where* Δ *is a diagnosis (see Def. 17) in the predicates* AB_i *for the CSP as of Def. 20 derived from* (G, τ), T_S *refers to the segmented path/trace derived for* $CSP(G, \tau)$, *and INTRACE is the corresponding minterm in the intrace variables (cf Def. 20) describing which segments are in the path* π^{corr} *depicted by* Δ.

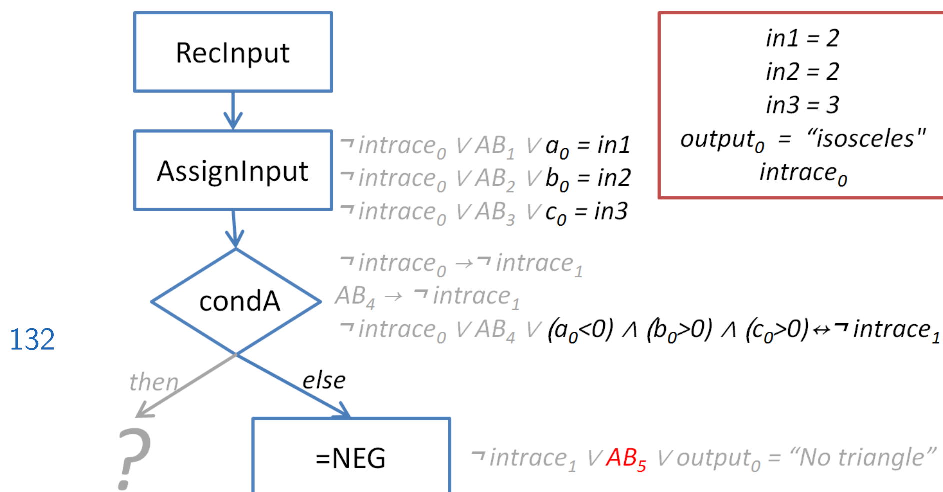
The debugging approach takes into consideration only the binary branching decisions. Programming constructs like switch and pick in BPEL, the exact choice to be taken remains part of future work. Moreover, our debugging



(a)Diagnosis AB_1: incorrect input value



(b)Diagnosis AB_4: incorrect branching constraint



(c)Diagnosis AB_5: incorrect output value

Figure 6.3: A graphical presentation of the running example's single-fault diagnoses.

encoding can not be applied for non-deterministic constructs like Flow where more than one paths can be active simultaneously. Also, our approach focusses on debugging single paths rather than the complete model. The choice was made so as to reduce the runtime diagnosing costs.

6.4 Experiments

For evaluating our approach, we built a corresponding prototype that extracts a BPEL flow graph from a corresponding process and translates it to MINION constraints. MINION (we used version 1.6.1) is an open source project, and supports arithmetic, logic, and relational operators over Boolean and Integer variables. All experiments were carried out on a MacBook Pro (Late 2011) with a 2.4 GHz Intel Core i5, 4 GB 1333 MHz DDR3, running OS X 10.7.2.

As a first step, we took three programs; two from software engineering domain, i.e., Triangle, BMI, and the one from SOA papers. We injected single faults manually to create different faulty but executable versions of the aforementioned programs. In Figure 6.4 you can see the BPEL flow graph of the Bank Loan Process. Its main objective is to approve, delay, or reject loan requests based on client's history. Loan requests for less than 10 000 are approved immediately, if the client has a credible history (then there is a low risk involved in granting it). For all other requests, an external service (*Assessor*) is invoked. The output of this external service would be "approved", "pending", or "reject".

For scenario Loan-V1, we changed the *IfLoan* condition from $loan \leq 10\,000$ to $loan \neq 10\,000$. The test case t with $I = \{loan = 1\,000\,000, clientId = 20\}$ as input and the expected output $E = \{reject\}$ unveils this fault, as the faulty version results in the output $O = \{approved\}$. That is, the mutated program takes a different branch and calls the"Risk" web service rather than the "Assessor" service, resulting in the actual output $O = \{approved\}$.

For Loan-V2, the *IfLoan* condition was changed from $loan \leq 10\,000$ to $loan > 10\,000$. For input $I = \{loan = 1\,000\,000, clientId = 20\}$, the expected output was $E = \{reject\}$, but the altered program returned $O = \{approved\}$.

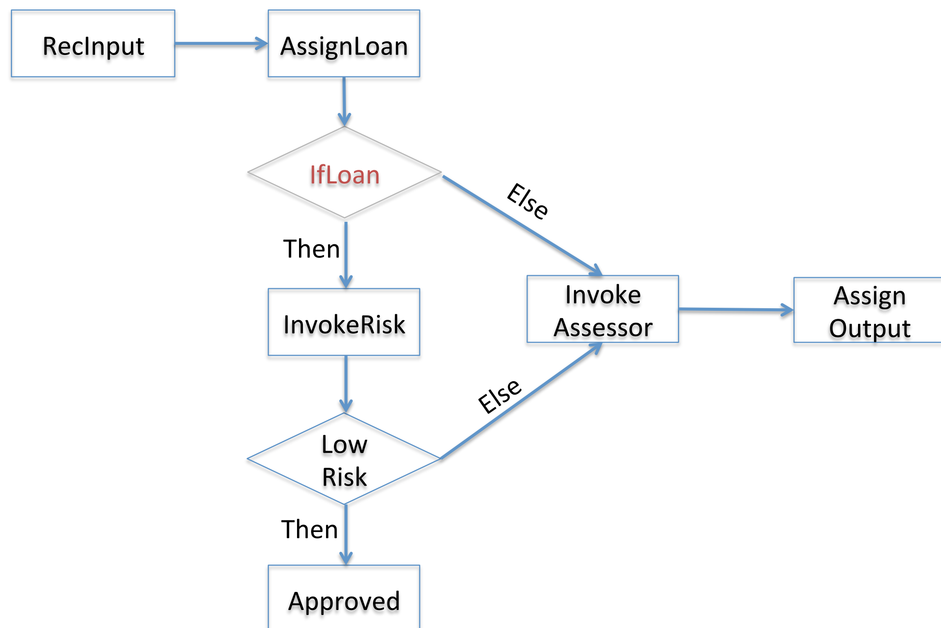


Figure 6.4: BPEL Flow Graph of the Bank Loan Business Process

Again the different branch chosen by the program results in the actual output would be $O = \{approved\}$ derived by a different assessment variant. For scenario Loan-V4, we changed the *IfLoan* condition from $loan \leq 10\,000$ to $loan \leq 100\,000$. The test case with input $I = \{loan = 100\,000, clientId = 20\}$ was expected to take the "Else" branch and based on the loan amount give the output $E = \{pending\}$. But, since the *IfLoan* condition was mutated, the "Then" branch was taken and the "Risk" web service derived the actual output $O = \{approved\}$. Mutating the *LowRisk* condition from $risk = 0$ to $risk = 1$ yielded the mutated program for scenario Loan-V5. The test case t with input $I = \{loan = 1\,000, clientId = 21\}$, was expected to yield output $E = \{pending\}$, but the mutated *LowRisk* condition resulted in the actual output $O = \{approved\}$.

BMI is a small BPEL process that computes the Body Mass Index (BMI) from two parameters: height and weight. Based on the value returned by invoked web service, the process decides if the provided result should be "underweight", "healthy", "overweight", "obese", or "very obese".

For our tests, we used the following scenarios. For BMI-V1, we changed (mutated) the *IfUnderWeight* condition from $bmiVal \leq 19$ to $bmiVal \neq 19$. For input $I = \{weight = 105, height = 160\}$, the expected output was $E = \{very\ obese\}$, but due to the mutation, the program led to output $O = \{underweight\}$. Mutating the *IfHealthy* condition from $bmiVal \leq 25$ to $bmiVal > 25$ yielded the faulty process of scenario BMI-V3. The revealing test case had $I = \{weight = 75, height = 160\}$ as input, and $E = \{overweight\}$ as expected output. Contradicting E, the actual output was $O = \{healthy\}$. For BMI-V4, we changed the *IfUnderWeight* condition from $bmiVal \leq 19$ to $bmiVal = 19$. The input $I = \{weight = 50, height = 160\}$ was expected to yield in output $E = \{healthy\}$, but the mutated program returned $O = \{underweight\}$ instead. The same *IfUnderWeight* condition was changed to $bmiVal > 19$ for scenario BMI-V5. The revealing test case had as input $I = \{weight = 75, height = 160\}$ with the experienced output $O = \{underweight\}$ contradicting the expected one $E = \{overweight\}$.

For the Triangle example as explained in the introduction, we considered following mutations. In TRI-V1, *condA* $(a_0 > 0) \wedge (b_0 > 0) \wedge (c_0 > 0)$ was changed to $(a_0 < 0) \wedge (b_0 > 0 \wedge (c_0 > 0)$. The test case with input $I = \{a = 2, b = 2, c = 3\}$ was expected to give output $E = \{IS\}$ (i.e. "isosceles"). The mutated program, however, returned $O = \{NEG\}$ (i.e. "No triangle") as output. Incase of TRI-V2, *condA* condition was mutated to $(a_0 > 0) \wedge (b_0 < 0) \wedge (c_0 > 0)$. The failing test case had an input $I = \{a = 1, b = 1, c = 1\}$, where the expected output was $E = \{EQ\}$ (i.e. "equilateral"). Contradicting E, the program returned the incorrect output $O = \{NEG\}$. In TRI-V3, *condA* was changed to $(a_0 > 0) \wedge (b_0 > 0) \wedge (c_0 < 0)$, again the revealing test case having $I = \{a = 1, b = 1, c = 1\}$ as input and $E = \{EQ\}$ as expected output. Likewise for TRI-V1 and TRI-V2, the actual output was $O = \{NEG\}$ contradicting E. For TRI-V4 the *condC* was changed from $(a_0 = b_0) \wedge (b_0 = c_0)$ to $(a_0 = b_0) \vee (b_0 = c_0)$. The revealing test case t had $I = \{a = 2, b = 2, c = 1\}$ as an input, and was expected to return $E = \{IS\}$, but due to the mutation, the program returned $O = \{EQ\}$.

One can argue that the number of constraints presented in this work are rather small. In SOA domain, however, most of the system's functionality is wrapped in web services, about which we have partial information only. The prime purpose of a business process is to integrate different web services in order to achieve a business goal. In particular this means, the focus of

Table 6.1: Single Faults Diagnoses.

Program	Inputs	Outputs	WS	S	S'	CO	Obs	Var	AllSols	T min (s)	T max (s)	T avg (s)
Loan-V1	2	1	2	9	9	17	3	14	7	0.129246	0.135582	0.1326410
Loan-V2	2	1	2	9	9	17	3	14	7	0.243236	0.302773	0.2548036
Loan-V4	2	1	2	9	9	17	3	14	8	0.023699	0.027242	0.0252019
Loan-V5	2	1	2	9	9	17	3	14	7	0.021730	0.025869	0.0238879
Bmi-V1	2	1	1	8	8	15	3	11	4	0.008145	0.009979	0.0088565
Bmi-V3	2	1	1	9	9	18	3	13	7	0.008578	0.010395	0.0092091
Bmi-V4	2	1	1	8	8	15	3	11	6	0.008361	0.010359	0.0091162
Bmi-V5	2	1	1	8	8	15	3	11	6	0.008381	0.009963	0.0091609
Tri-V1	3	1	1	5	5	12	4	10	3	0.003844	0.005370	0.0043121
Tri-V2	3	1	1	5	5	12	4	10	3	0.004226	0.005441	0.0043277
Tri-V3	3	1	1	5	5	12	4	10	3	0.003918	0.005127	0.0043014
Tri-V4	3	1	1	10	10	21	4	21	10	0.004029	0.006739	0.0048355

a business process is on integration of the involved web services' in- and outputs.

Table 6.1 shows the results of our experiments. The total numbers of program's inputs and outputs are given as *Inputs* and *Outputs* in the table. *WS* refers to the number of external web services invoked by the BPEL process. The number of constraints and variables derived for a path are reported as *CO* and *VAR*, and *Obs* represents number of observations. Column *AllSols* reports on the number of all possible single fault solutions for a particular example. Similarly, T min (s) and T max (s) define minimum and maximum time in seconds needed to compute diagnoses, where T avg (s) is the average total time over ten runs.

For our results we had some interesting findings. First, the execution time taken by our approach for diagnosing BPEL process is negligible in contrast to the time needed for testing such BPEL examples [Jehan et al., 2014]. Second, the diagnosis approach worked best for the BMI-V1 example, where our diagnoses removed half of statements from the diagnostic scope. For other variants, the reduction was about 25 %. Incase of the running example "Triangle", we got a reduction of 40 % for three programs, but in TRI-V4, not a single statement could be excluded from the search for the fault.

Figure 6.5 compares the trace size and diagnoses size for all program

variants used in our experiments. This figure shows that the amount of components needed to be investigated could be reduced.

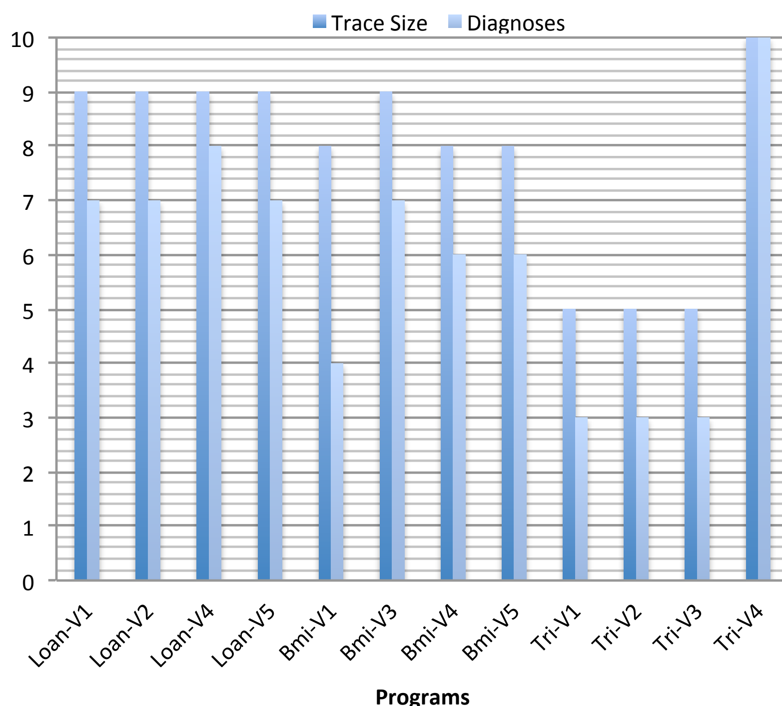


Figure 6.5: Comparison of the trace size and diagnoses size for faulty programs.

As a second step, we extended the example set with three more programs. The Geo example computes area and circumference of a circle or a square discussed in our paper [Hofer et al., 2015]. Tcas is a SOA variant of the well-known traffic collision avoidance system used in many software engineering examples[1]. The calculator (Calc) example demonstrates additions, subtractions, divisions and multiplications as well as control structures like if-then-else and while.

For experiments, we introduced single or double faults in all examples. Table 6.2 shows our results when searching for single and double fault diagnoses. Depending on the injected fault and the specific scenario, there

[1]see also http://sir.unl.edu/

are not always single and double fault diagnoses. Column F indicates the number of faults injected for a specific sample. The total number of inputs and outputs to a program are given in column *I/O*. Column S indicates the number of statements in the diagnosis problem's execution trace. The number of constraints and variables derived for the trace are reported in the columns C and V. For single fault ($|\Delta| = 1$) and double fault diagnoses ($|\Delta| = 2$) we report in the corresponding columns on the number of solutions found, as well as the average time in milliseconds over three computations.The approach's diagnostic performance for the single fault scenarios is indicated in column R by means of the reduction in statements that have to be considered by using the following formula:

$$Reduction = 1 - \frac{|\text{single fault diagnoses}|}{|\text{executed statements}|}$$

In other words, this means how many statements out of the executed ones we could eliminate from consideration. For double fault scenarios, we need to establish a more appropriate metric and thus do not report corresponding numbers. The corresponding files can be downloaded from the following link[2].

We draw several conclusions from the experimental analysis: First, the time needed to compute the diagnosis is negligible, which makes it an attractive choice for augmenting any SOA debugging engine; second, we observed average reduction of 22 % to 50 % in nearly all examples, making an attractive average reduction of 39.07 %. For TCAS, the reduction got as high as 89.1 %. The specific structure of the Calc example did not allow for any reduction.

6.5 Conclusions

Similar to testing, we modeled our debugging approach as a constraint-satisfaction problem with an underlying model being the program's control flow graph.The approach requires the execution trace of a faulty program

[2] http://a4s.ist.tugraz.at/downloads/ieaaie15.zip

with at least one test case. The partial behavior of the called web services is specified using the auxiliary data in the form of pre- and post conditions. We assume that the specification documents are provided for the debugging purposes. Moreover, we make use of existential constraints in addition to propagated constraints to model the debugging problem. In order to reduce the debugging cost, we debug only a single execution's trace in a control flow graph, rather than a full temporal model proposed by DIAMOND project. Considering the fact that actual business processes tend to be long running workflows, we segment the trace in different segments. The purpose is to inform the user about probable positions in the flow graph, where a program might have deviated from the intended flow.

The first experiments using SOA and software engineering examples showed promising results. However, the approach can be extended in many ways: For example, the effect of weak and strong pre- and post- conditions is a planned future work. Also, we presently work on a weak fault model with functional faults, strong fault models specifying non-functional properties like SLA agreements can be useful for SOA domain. Also, the approach is general enough and can be applied to functional debugging of any software program. One limitation of our approach is that it only works for sequential processes. A fruitful extension could be done by extending the approach to debug non-deterministic control-flow graphs. Since we make use of dynamic slicing in our work, it can be interesting to compare our work with other slicing-based debugging approaches for SOA programs.

Table 6.2: Results for programs with single and double faults.

Program	F	I/O	S	C	V	$\lvert\Delta\rvert = 1$		$\lvert\Delta\rvert = 2$		R(%)
						#	$T_{avg}(ms)$	#	$T_{avg}(ms)$	
Loan-V1	1	2/1	9	17	12	8	19.03	0	218.97	11.0
Loan-V2	1	2/1	9	17	12	8	15.52	0	14.15	11.0
Loan-V3	2	2/1	8	15	12	3	18.84	4	15.82	
Loan-V4	2	2/1	8	15	12	3	17.85	4	15.99	
Loan-V5	2	2/1	8	15	12	3	15.37	0	12.54	
Loan-V6	2	2/1	8	15	12	3	15.81	0	12.89	
Bmi-V1	1	2/1	8	15	11	4	10.93	0	3.86	50.0
Bmi-V2	1	2/1	9	18	13	7	12.50	0	4.56	22.2
Bmi-V3	1	2/1	8	15	11	6	12.33	0	4.21	25.0
Bmi-V4	1	2/1	8	15	11	6	12.07	0	4.08	25.0
Bmi-V5	2	2/1	8	15	11	4	8.95	0	4.08	
Bmi-V6	2	2/1	8	15	11	4	8.73	0	3.93	
Bmi-V7	2	2/1	8	15	11	4	9.29	0	3.87	
Bmi-V8	2	2/1	8	15	11	4	8.88	0	3.72	
Tri-V1	1	3/1	5	12	10	3	4.31	0	3.28	40.0
Tri-V2	1	3/1	5	12	10	3	4.33	0	3.15	40.0
Tri-V3	1	3/1	5	12	10	3	4.30	0	3.37	40.0
Tri-V4	1	3/1	10	29	21	10	4.53	0	3.36	0.0
Tri-V5	2	3/1	5	12	10	2	4.52	1	3.80	
Tri-V6	2	3/1	5	12	10	2	4.53	1	3.77	
Tri-V7	2	3/1	5	12	10	2	3.97	1	3.17	
Tri-V8	2	3/1	5	12	10	2	4.22	1	3.21	
Geo-V1	1	2/2	5	13	10	2	5.52	2	6.48	60.0
Geo-V2	1	2/2	5	13	10	2	7.21	2	6.51	60.0
Geo-V3	2	2/2	5	13	10	2	9.47	1	5.46	
Geo-V4	2	2/2	5	13	10	3	9.70	0	4.16	
Geo-V5	2	2/2	5	13	10	1	8.21	5	12.15	
Geo-V6	2	2/2	5	15	10	1	7.74	2	9.34	
Tcas-V1	1	12/1	74	98	71	14	597.86	15	6200.84	81.0
Tcas-V2	1	12/1	74	98	71	18	602.45	2	4402.64	75.6
Tcas-V3	1	12/1	74	98	71	8	527.77	12	6374.44	89.1
Tcas-V4	1	12/1	74	98	71	18	653.73	9	4891.03	75.6
Tcas-V5	1	12/1	75	103	73	18	619.85	9	4217.57	76.0
Tcas-V15	2	12/1	75	96	77	7	413.66	13	3632.64	
Tcas-V40	2	12/1	73	90	73	13	613.24	9	2135.63	
Calc-V1	1	3/1	12	27	22	12	2.87	0	2.38	0.0
Calc-V2	1	3/1	8	22	19	8	2.79	0	1.78	0.0
Calc-V3	1	3/1	13	35	27	13	19.68	0	5.33	0.0
Calc-V4	2	3/1	12	28	18	12	2.32	0	2.36	
Calc-V5	2	3/1	15	39	27	15	3.13	0	3.60	

7 Conclusions

7.1 Results summary

The thesis addresses following key challenges with respect to reliability of SOA-based processes.

1. Test case generation and execution
2. Test case minimization
3. Model-based Debugging

One of many hurdles in testing SOA-based processes is that of limited observability, that is, services' business logic is hidden making it a black-box for the tester. Even at workflow level, where the basic workflow logic is available, major portion of functionally still lies in external services. Since these services are basic building blocks of any SOA workflow, appropriate testing strategies have been an area of research for a decade. Another challenge related to effective testing of SOA-based workflows is the limited control over these services. Based on SOA principle of loose-coupling, services change independently making integration testing more intricate than in traditional software. This limited observability makes test case generation more challenging, because generating models from service descriptions is cumbersome and inefficient. Similarly, restricted control over external services in any SOA-based process poses serious threat to efficient test case execution.

There are number of solutions suggested to tackle this issue: We propose model-based grey-box testing solution, which is a combination of verification and testing for SOA processes. Formal verification helps in detecting unreachable parts in service workflows, and is widely investigated by many

researchers for efficient test suite generation. Our work employs graph-based approach to generate control flow graph out of BPEL workflow. This is particularly useful in SOAs where formal specifications of system under test are hardly available. In order to generate test data, we make use of state-of-art MINION constraint solver. Since workflow model contains partial behavioral information, that is, the functioning of external services called in working is not available, we make use of pre- and post-conditions to augment BPEL model. These contracts provided by the developer or service providers can be employed to solve the observability issues regarding services participating in a workflow. These contracts are also of greater help to integrator both in verifying external services contracts and in specifying test oracles. Tester can easily add these contracts on BPEL flow graph representation in our tool *BPELTester*.

Once tester has augmented BPEL control flow graph model, our tool provides different options for test case generation. There are two main algorithms, that is, *AllPaths* and *RandomPaths* for generating test paths (abstract test cases) on BPEL control flow graph model. The purpose was to compare structured and random approaches for test case generation. Both algorithms have further two variants, that is, each for sequential and concurrent BPEL processes. Incase of *AllPaths* algorithm, tester needs to input the *MaxLen* parameter, which specifies the maximum path length searched by algorithm. The model along with the test case represents the constraint-satisfaction problem. The feasibility of these test paths is checked with the help of MINION constraint solver, which also generates test inputs (executable test cases) in case of feasible test paths. For *RandomPaths*, instead of *MaxLen*, tester needs to specify the required number of random paths for some fixed *MaxLen* parameter. For test execution, we rely on external tools, that is, BPELUnit and MuBPEL. *BPELUnit* measures the unit coverage of our generated test suites, and *MuBPEL* is a mutation testing tool. We observed that the structure-based algorithm performed better both in attained coverage and execution time. We achieved 100% coverage in all of the examples, but the highest achievable mutation score was about 90%. The test case generation time was negligible as compared to the test execution time. Moreover, offline testing can be useful to a number of SOA stakeholders, that is, developers, providers and integrators, thereby reducing the testing cost.

Another issue is high testing cost of SOA applications. This could imply either cost of testing at service level; i.e. cost of invoking real services every time; or cost of testing at composition level, where Integrator needs to make sure that all the services in the composition are functioning according to service-level- agreements. There is strong need for solutions that can reduce both such costs.

Test suite minimization based on test case deletion aims at eliminating all redundancies from the test suite. Therefore minimizing test suites is important for reducing high test execution cost involved SOAs regression testing. Regression testing can be classified as test suite minimization (reduction), test suite selection, or test suite prioritization. First, by employing *AllRandomPaths* algorithm, we randomly selected subsets of original test suites in order to have a look at the decline of failure detection capabilities. *AllRandomPaths* takes two parameter, that is, *setSize* and *solSize* in addition to *MaxLen*. The *setSize* determines the number of subsets to be generated randomly. The generated subsets represent the number of test suites. Note that, we only execute the test suite of max. subset size for the analysis. The third parameter *solSize* defines the number of test values generated by the constraint solver. For our experiments, we chose solutions of size 1, 3 and 5. That means, for one test case, we asked constraint solver for different test values, and study the impact on the attained coverage. The initial study observed redundancy of 0.538 in one of the examples. We computed the probabilities from randomly selected 10 subsets of a particular size. We see that even for larger subsets there is no guarantee to select the right ones that do not reduce mutation score.

Then, we applied three test suite reduction algorithms, that is, *LinMIN*, *BinarySearch* and *Delta-Debugging*. The target was to reduce the test suite size keeping the quality measuring attributes such as unit coverage and mutation scores within specified range. It was observed that upon changing alpha, we get maximum reduction in *Delta-Debugging* algorithm, since we need less number of tests to attain the maximum reduction rate. Similarly, *LinMIN* also performs better on changing alpha, but shows no improvement on varying the subset size. Also, this algorithm takes longer to compute than the other two algorithms. *BinarySearch* algorithm, on the other hand, always finds a subset with desired reduction on increasing the subset size, but it does not always find the required set upon changing alpha.

Although addition of multiples tests per path increases redundancy in the test suite, but *LinMIN* and *BinarySearch* algorithms converge better with larger test suite. Interestingly, the Delta-Debugging algorithm is not effected by increasing the test suite size. The execution time is however doubled for all algorithms by increasing the subset size.

In the context of debugging service compositions, we also adapted a model-based approach in combination with dynamic slicing technique. The focus was to present a light-weight debugging solution for locating functional faults in service compositions. The approach requires an execution trace along with one test case. We observed an average reduction of 39.07% in our examples with a negligible time overhead.

7.2 Open Questions and Future Work

There are many ways to extend the work presented in the thesis:

First, one of the key challenge in testing SOAs is the lack of real-world case-studies. Although there has been a lot of research done in devising new testing methodology for SOA-based applications, but 71% of the publications had no empirical evaluations. This absence of benchmark examples made the comparative analysis of different testing techniques next to impossible.

Second, in the context of test case generation, we generate test inputs based on boundary-value analysis and constraints-based reasoning. However, in some scenarios, real test data (RTD) would be desirable, in generating realistic test data. This is particularly useful in case of ”string” data types.

Third, debugging process should consider Flow constructs. But, the most important step should be availability of some benchmark suites. With out real-world problems, it would not be fair to compare and contrast different testing and debugging methodologies.

List of Definitions

1 Definition (BPEL Flow Graph) 53
2 Definition (Path) 53
3 Definition (Path Condition) 53
4 Definition (Feasible Path) 54
5 Definition (An Extended BPEL Flow Graph) 54
6 Definition (Finite Path) 55
7 Definition (Finite Path Segment) 55
8 Definition (Finite Run) 55
9 Definition (Feasible Run) 55
10 Definition (Run Constraints) 56
11 Definition (Run Scope) 56
12 Definition (Test Case and Test Suite) 57

13 Definition (Reduction) 106
14 Definition (Redundancy) 106

15 Definition (Trace) 124
16 Definition (Test Case) 124
17 Definition (Diagnosis) 124
18 Definition (Segmented Path) 127

LIST OF DEFINITIONS

19 Definition (DiagnosisProblem) 129

20 Definition (DiagnosisCSP) 129

21 Definition (Extended Diagnosis) 131

List of Theorems and Lemmas

1 Lemma (ComputingAllpaths) 67
2 Lemma (CheckSatisfiability) 67
3 Lemma (Valid Solution) 67
4 Theorem (AllPathsSoundness) 67
5 Theorem (AllPathsCompleteness) 67
6 Lemma (ComputingAllruns) 71
7 Lemma (CheckRunSatisfiability) 71
8 Theorem (StructRunsSoundness) 71
9 Theorem (StructRunsCompleteness) 71
10 Lemma (RandomTestSuiteSize) 83
11 Lemma (ComputingFeasibleRandompaths) 83
12 Theorem (RandomPathsCorrectness) 83
13 Lemma (ComputingFeasibleRandomruns) 91
14 Theorem (RandomRunsCorrectness) 91

15 Theorem (LinMINCorrectness) 109

16 Theorem (GetDiagnosesCorrectness) 130

Bibliography

[ATM,] Jboss example. http://docs.jboss.com/jbpm/bpel/v1.1/userguide/tutorial.atm.html. 85

[Xpa, 2011] (2011). Xml path language (xpath) 2.0. http://www.w3.org/TR/xpath20/. 63

[Bpe, 2012] (2012). Eclipse bpel designer tutorial. http://servicetechnologies.wordpress.com/exercises/. 49

[Abreu et al., 2009] Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. C. (2009). A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792. 44

[Active Endpoints, 2010] Active Endpoints (2010). Active VOS engine. http://www.activevos.com. 76, 77

[Adrion et al., 1982] Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C. (1982). Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192. 32, 33, 41

[Ardagna et al., 2007] Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., and Plebani, P. (2007). Paws: A framework for executing adaptive web-service processes. *IEEE Software*, 24(6):39–46. 46

[Ardissono et al., 2008] Ardissono, L., Bocconi, S., Console, L., Furnari, R., Goy, A., Petrone, G., Picardi, C., Segnan, M., and Dupré, D. T. (2008). Enhancing web service composition by means of diagnosis. In *Business Process Management Workshops*, Lecture Notes in Business Information Processing, pages 468–479. 46

[Balepur Venkatanna Kumar, 2010] Balepur Venkatanna Kumar, Prakash Narayan, T. N. (2010). *Implementing SOA Using Java EE*. Prentice Hall. 11, 16

[Bentakouk et al., 2011] Bentakouk, L., Poizat, P., and Zaïdi, F. (2011). Checking the behavioral conformance of web services with symbolic testing and an smt solver. In *Proceedings of the 5th International Conference on Tests and Proofs*, TAP'11, pages 33–50, Berlin, Heidelberg. Springer-Verlag. 38

[Berkelaar,] Berkelaar, M. lp solve, a public domain mixed integer linear program solver. available at http://groups.yahoo.com/group/lp_solve/. 41

[Berners-Lee, 1989] Berners-Lee, T. (1989). Hypertext Transfer Protocol. https://www.w3.org/Protocols/. 16

[Bhushan, 1985] Bhushan, A. (1985). File Transfer Protocol. https://tools.ietf.org/html/rfc959. 16

[Binder et al., 2015] Binder, R. V., Legeard, B., and Kramer, A. (2015). Model-based testing: Where does it stand? *Commun. ACM*, 58(2):52–56. 51

[Blanco et al., 2009] Blanco, R., Garcia-Fanjul, J., and Tuya, J. (2009). A first approach to test case generation for bpel compositions of web services using scatter search. In *International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW '09.*, pages 131–140. 43

[Boubeta-Puig et al., 2011] Boubeta-Puig, J., Medina-Bulo, I., and Garcìa-Domìnguez, A. (2011). Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011*, pages 398–407. 35

[Boyer et al., 1975] Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA. ACM. 37

[Bozkurt, 2013] Bozkurt, M. (2013). Cost-aware pareto optimal test suite minimisation for service-centric systems. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1429–1436. ACM. 102

[Bozkurt and Harman, 2011] Bozkurt, M. and Harman, M. (2011). Automatically generating realistic test input from web services. In *International Symposium on Service-Oriented System Engineering (SOSE), 2011*, pages 13–24. 43

[Bozkurt et al., 2013] Bozkurt, M., Harman, M., and Hassoun, Y. (2013). Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, 23(4):261–313. 2, 3, 20, 36, 37, 39, 42, 43, 49, 58, 77, 78, 104

[BPEL Mutation tool, 2011] BPEL Mutation tool (2011). Mubpel- a mutation testing tool for ws-bpel. `https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL`. 52, 77, 85

[BPELDesigner, 2006] BPELDesigner (2006). Eclipse BPEL designer. `https://eclipse.org/bpel/`. 76

[Brandis and Mössenböck, 1994] Brandis, M. M. and Mössenböck, H. (1994). Single-pass generation of static assignment form for structured languages. *ACM TOPLAS*, 16(6):1684–1698. 53, 56

[Budd and Angluin, 1982] Budd, T. A. and Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45. 35

[Cadar et al., 2011] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA. ACM. 38

[Canfora and Penta, 2009a] Canfora, G. and Penta, M. (2009a). Software engineering. chapter Service-Oriented Architectures Testing: A Survey, pages 78–105. Springer-Verlag, Berlin, Heidelberg. 1, 2, 29, 36

[Canfora and Penta, 2009b] Canfora, G. and Penta, M. (2009b). Software engineering. chapter Service-Oriented Architectures Testing: A Survey, pages 78–105. Springer-Verlag, Berlin, Heidelberg. 80

[Canfora and Penta, 2006] Canfora, G. and Penta, M. D. (2006). Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17. 2, 36, 49, 102

[Cardoso, 2006] Cardoso, J. (2006). Complexity analysis of bpel web processes. *Software Process: Improvements and Practice Journal*, 12:35–49. 79

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA. ACM. 82

[Clarke, 2008] Clarke, E. M. (2008). 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg. 40

[Clarke et al., 1983] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1983). Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 117–126, New York, NY, USA. ACM. 39

[Clarke et al., 2012] Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. (2012). *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg. 40

[Clarke and Lerda, 2007] Clarke, E. M. and Lerda, F. (2007). Model checking: Software and beyond. 13(5):639–649. 39

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA. ACM. 44

[Cloud, 2013] Cloud, J. (2013). Decomposing twitter- adventures in service-oriented architecture. http://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture. 1, 7

[COM, 1993] COM (1993). Common Object Model. https://www.microsoft.com/com/. 8

[Console et al., 2007] Console, L., Picardi, C., and Dupré, D. T. (2007). A framework for decentralized qualitative model-based diagnosis. In *IJCAI*, pages 286–291. 1, 46

[CORBA, 1991] CORBA (1991). Common Object Request Broker Architecture. http://www.omg.org/cgi-bin/doc?formal/98-12-01. 8

[Corporation(IDC), 2015] Corporation(IDC), I. D. (2015). Worldwide big data technology and services market. http://www.idc.com/getdoc.jsp?containerId=prUS40560115. 1

[Dai et al., 2007] Dai, G., Bai, X., Wang, Y., and Dai, F. (2007). Contract-based testing for web services. In *Computer Software and Applications Conference, 2007. 31st Annual International*, volume 1, pages 517 –526. 42

[Daigneau, 2011] Daigneau, R. (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 1 edition. 8, 29

[Davis, 1984] Davis, R. (1984). Diagnostic reasoning based on structure and behavior. *Artif. Intell.*, 24(1-3):347–410. 44

[Davis, 1993] Davis, R. (1993). Retrospective on "diagnostic reasoning based on structure and behavior". *Artif. Intell.*, 59(1-2):149–157. 44

[de Kleer and Williams, 1987] de Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130. 44

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: an efficient smt solver. In *Proceedings of TACAS'08*, pages 337–340, Berlin, Heidelberg. Springer-Verlag. 38

[DeMillo et al., 1978] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41. 33, 35

[DeMillo et al., 1996] DeMillo, R. A., Pan, H., and Spafford, E. H. (1996). Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA. ACM. 44

[Di Penta et al., 2007] Di Penta, M., Canfora, G., Esposito, G., Mazza, V., and Bruno, M. (2007). Search-based testing of service level agreements. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1090–1097, New York, NY, USA. ACM. 43

[Diamond, 2010] Diamond (2010). Web service diagnosis, monitoring, and diagnosability. http://wsdiamond.di.unito.it/. 3

[Dong, 2009] Dong, W. (2009). Test case generation method for bpel-based testing. In *International Conference on Computational Intelligence and Natural Computing, 2009. CINC '09.*, volume 2, pages 467 –470. 40

[Dorsey, 2006] Dorsey, J. (2006). Twitter inc.-social networking service. https://twitter.com/. 7

[Eich, 1995] Eich, B. (1995). Java script programming language. https://www.javascript.com/. 7

[Englander, 2002] Englander, R. (2002). *Java and SOAP*. O'Reilly Media, Inc. 9, 16

[Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 13, 14, 19, 23

[Erl, 2007] Erl, T. (2007). *SOA Principles of Service Design*. Prentice Hall PTR. 12

[Fiedling, 2000] Fiedling, R. T. (2000). Architectural styles and the design of network-based software architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. 19

[Fielding, 2000] Fielding, R. T. (2000). REpresentational State Transfer. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. 10

[Friedrich, 2010] Friedrich, G. (2010). Repair of service-based processes -an application area for logic programming. 47

[Friedrich et al., 2010a] Friedrich, G., Fugini, M., Mussi, E., Pernici, B., and Tagni, G. (2010a). Exception handling for repair in service-based processes. *Software Engineering, IEEE Transactions on*, 36(2):198–215. 2, 47

[Friedrich et al., 2010b] Friedrich, G., Mayer, W., and Stumptner, M. (2010b). Diagnosing process trajectories under partially known behavior. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 111–116, Amsterdam, The Netherlands, The Netherlands. IOS Press. 1, 32, 46

[Garcia-fanjul et al., 2006] Garcia-fanjul, J., Tuya, J., and Riva, C. D. L. (2006). Generating test cases specifications for bpel compositions of web services using spin. 39

[Gent et al., 2006] Gent, I. P., Jefferson, C., and Miguel, I. (2006). Minion: A fast scalable constraint solver. In *In: Proceedings of ECAI 2006*, pages 98–102. 41, 46, 51, 57

[GlassFish, 2006] GlassFish (2006). Glassfish application server. https://glassfish.java.net/. 76

[Godefroid et al., 2005a] Godefroid, P., Klarlund, N., and Sen, K. (2005a). Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223. 37

[Godefroid et al., 2005b] Godefroid, P., Klarlund, N., and Sen, K. (2005b). Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223. 82

[Goodenough and Gerhart, 1975] Goodenough, J. B. and Gerhart, S. L. (1975). Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, New York, NY, USA. ACM. 32

[Gosling, 1995] Gosling, J. (1995). Java programming language. https://www.oracle.com/java/. 7

[Gotlieb et al., 1998] Gotlieb, A., Botella, B., and Rueher, M. (1998). Automatic test data generation using constraint solving techniques. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 53–62. 58

[Greiner et al., 1989] Greiner, R., Smith, B. A., and Wilkerson, R. W. (1989). A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88. 44

[Hallwyl et al., 2010] Hallwyl, T., Henglein, F., and Hildebrandt, T. (2010). A standard-driven implementaion of ws-bpel 2.0. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2472–2476, New York, NY, USA. ACM. 78

[Harvey, 2005] Harvey, M. (2005). *Essential Business Process Modeling*. O'Reilly Media, Inc. 21, 22, 23

[Heckel and Lohmann, 2005] Heckel, R. and Lohmann, M. (2005). Towards contract-based testing of web services. *Electron. Notes Theor. Comput. Sci.*, 116:145–156. 42

[Heimdahl and George, 2004] Heimdahl, M. P. and George, D. (2004). Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 176–185. IEEE Computer Society. 103

[Hewitt, 2009] Hewitt, E. (2009). *Java SOA Cookbook - SOA implementation recipes, tips, and techniques.* O'Reilly Media, Inc. 15

[Hierons and Ural, 2009] Hierons, R. and Ural, H. (2009). Overcoming controllability problems with fewest channels between testers. *Computer Networks*, 53(5):680–690. 31, 36, 49

[Hierons and Ural, 2008] Hierons, R. M. and Ural, H. (2008). The effect of the distributed test architecture on the power of testing. *The Computer Journal*, 51(4):497–510. 31, 36

[Hofer et al., 2014] Hofer, B., Jehan, S., Pill, I., and Wotawa, F. (2014). Functional diagnosis of a SOA's BPEL processes. In *25th International Workshop on Principles of Diagnosis (DX)*. 5, 123

[Hofer et al., 2015] Hofer, B., Jehan, S., Pill, I., and Wotawa, F. (2015). Focused diagnosis for failing software tests. In Ali, M., Kwon, Y. S., Lee, C.-H., Kim, J., and Kim, Y., editors, *Current Approaches in Applied Artificial Intelligence*, volume 9101 of *Lecture Notes in Computer Science*, pages 712–721. Springer International Publishing. 5, 123, 137

[Hofer and Wotawa, 2012] Hofer, B. and Wotawa, F. (2012). Combining slicing and constraint solving for better debugging: The conbas approach. *Adv. Software Engineering*. 126

[Hou et al., 2008] Hou, S.-S., Zhang, L., Xie, T., and Sun, J.-S. (2008). Quota-constrained test-case prioritization for regression testing of service-centric systems. In *IEEE International Conference on Software Maintenance, ICSM 2008.*, pages 257–266. IEEE. 103

[Howden, 1978] Howden, W. E. (1978). Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, (4):293–298. 33

[Humble, 2011] Humble, C. (2011). Twitter shifting more code to jvm, citing performance and encapsulation as primary drivers. http://www.infoq.com/articles/twitter-java-use. 8

[IEEE, 1983] IEEE (1983). IEEE Standard Glossary of Software Engineering Terminology, Institute of Electrical and Electronics Engineers. *ANSI/IEEEStd729-1983*. 31, 32

[JavaRMI, 1997] JavaRMI (1997). Java api. https://docs.oracle.com/javase/tutorial/rmi/. 8

[Jehan et al., 2014] Jehan, S., Pill, I., and F.Wotawa (2014). SOA testing via random paths in BPEL models. In *10th Workshop on Advances in Model Based Testing; 2014 IEEE Seventh Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 260–263. 5, 50, 69, 136

[Jehan et al., 2013a] Jehan, S., Pill, I., and Wotawa, F. (2013a). Functional SOA testing based on constraints. In *8th Int. Workshop on Automation of Software Test (AST)*, pages 33–39. 5, 50, 85

[Jehan et al., 2013b] Jehan, S., Pill, I., and Wotawa, F. (2013b). SOA grey box testing - a constraint-based approach. In *5th Int. Workshop on Constraints in Software Testing, Verification and Analysis; 2013 IEEE Sixth Int. Conf.*

on Software Testing, Verification and Validation Workshops (ICSTW), pages 232–237. 5, 50

[Jehan et al., 2015] Jehan, S., Pill, I., and Wotawa, F. (2015). BPEL Integration Testing. In *18th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 69–83. 5, 50, 79

[Jia and Harman, 2011] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678. 35

[Josuttis, 2007] Josuttis, N. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc. 11, 29

[Juric and Krizevnik, 2010] Juric, M. B. and Krizevnik, M. (2010). *WS-BPEL 2.0 for SOA Composite Applications with Oracle SOA Suite 11G*. Packt Publishing. 20

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394. 37, 53

[Korel, 1990] Korel, B. (1990). Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879. 44, 47

[Korel and Laski, 1988] Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155–163. 45, 125

[Korel and Rilling, 1998] Korel, B. and Rilling, J. (1998). Dynamic program slicing methods. *Information & Software Technology*, 40(11-12):647–659. 125

[Langdon et al., 2010] Langdon, W. B., Harman, M., and Jia, Y. (2010). Efficient multi-objective higher order mutation testing with genetic programming. *J. Syst. Softw.*, 83(12):2416–2430. 85

[Lapadula et al., 2008] Lapadula, A., Pugliese, R., and Tiezzi, F. (2008). A formal account of ws-bpel. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, COORDINATION'08, pages 199–215, Berlin, Heidelberg. Springer-Verlag. 29, 78

[Leitner et al., 2013] Leitner, P., Schulte, S., Dustdar, S., Pill, I., Schulz, M., and Wotawa, F. (2013). The dark side of SOA testing: Towards testing contemporary SOAs based on criticality metrics. In *2013 5th International*

Workshop on Principles of Engineering Service-Oriented Systems (PESOS), pages 45–53. xvii, 12, 49

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562. 47

[Leymann, 2001] Leymann, F. (2001). Web Services Flow Language. http://xml.coverpages.org/WSFL-Guide-200110.pdf. 20

[Li et al., 2009] Li, Y., Ye, L., Dague, P., and Melliti, T. (2009). A decentralized model-based diagnosis for bpel services. In *21st International Conference on Tools with Artificial Intelligence, 2009. ICTAI '09.*, pages 609–616. 46

[Lin et al., 2006] Lin, F., Ruth, M., and Tu, S. (2006). Applying safe regression test selection techniques to java web services. In *International Conference on Next Generation Web Services Practices*, pages 133–142. IEEE. 103

[Lowis and Accorsi, 2011] Lowis, L. and Accorsi, R. (2011). Vulnerability analysis in SOA-Based Business Processes. *IEEE Transactions on Services Computing*, 4(3):230–242. 2

[Lübke, 2006] Lübke, D. (2006). Bpel unit. http://bpelunit.github.com. 51, 76, 85

[Lubke et al., 2009] Lubke, D., Singer, L., and Salnikow, A. (2009). Calculating bpel test coverage through instrumentation. In *ICSE Workshop on Automation of Software Test, 2009. AST '09.*, pages 115 –122. 76, 80

[Matsumoto, 1995] Matsumoto, Y. (1995). Ruby programming language. https://www.ruby-lang.org/. 7

[Mayer et al., 2012] Mayer, W., Friedrich, G., and Stumptner, M. (2012). On computing correct processes and repairs sing partial behavioral models. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pages 582–587. 2, 46, 121, 127

[Mayer and Stumptner, 2007] Mayer, W. and Stumptner, M. (2007). Model-based debugging – state of the art and future challenges. *Electron. Notes Theor. Comput. Sci.*, 174(4):61–82. 44

[McMinn, 2004] McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156. 43

[McMinn et al., 2012] McMinn, P., Shahbaz, M., and Stevenson, M. (2012). Search-based test input generation for string data types using the results of web queries. In *5th Int. Conf. on Software Testing, Verification and Validation (ICST), 2012*, pages 141–150. 43

[Mei et al., 2009] Mei, L., Chan, W.-K., Tse, T., and Merkel, R. G. (2009). Tag-based techniques for black-box test case prioritization for service testing. In *2009 Ninth International Conference on Quality Software*, pages 21–30. IEEE. 104

[Meyer, 1992] Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51. 42

[Microsoft, 2001] Microsoft (2001). XLANG. https://msdn.microsoft.com/en-us/library/aa577463.aspx. 20

[Modafferi et al., 2006] Modafferi, S., Mussi, E., and Pernici, B. (2006). Sh-bpel: a self-healing plug-in for ws-bpel engines. In *MW4SOC*, volume 184, pages 48–53. ACM. 3, 46

[Model, 2006] Model, O. R. (2006). The oasis reference model for service oriented architecture 1.0. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf. 12

[Murata, 1989] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580. 40

[Myers, 1979] Myers, G. J. (1979). *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA. 2

[Myers et al., 2011] Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. Wiley Publishing, 3rd edition. 31, 32, 33, 34

[Nica et al., 2013] Nica, I., Pill, I., Quaritsch, T., and Wotawa, F. (2013). The route to success - a performance comparison of diagnosis algorithms. In *International Joint Conference on Artificial Intelligence*, pages 1039–1045. 130

[Nica, 2010] Nica, M. (2010). *On the Use of Constraints in Automated Program Debugging*. PhD thesis, IST- TU Graz. 2, 51, 59, 60, 121, 126

[ODE, 2006] ODE (2006). Apache ode. `http://ode.apache.org/`. 76

[Odersky, 2004] Odersky, M. (2004). Scala programming language. `https://www.scala-lang.org/`. 7

[OMG, 2005] OMG (2005). Business Process Modeling Noration. `www.bpmn.org/`. 20

[Oracle Process Server, 2010] Oracle Process Server (2010). Oracle BPEL Process Manager. `http://www.oracle.com/technetwork/middleware/bpel`. 76

[Ouyang et al., 2007] Ouyang, C., Verbeek, E., van der Aalst, W. M. P., Breutel, S., Dumas, M., and ter Hofstede, A. H. M. (2007). Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198. 40

[OW2, 2012] OW2 (2012). Orchestra ow2. `http://orchestra.ow2.org/xwiki/bin/view/Main/WebHome`. 76

[Pill et al., 2015] Pill, I., Jehan, S., Wotawa, F., and Nica, M. (2015). Analyzing the reduction of test suite redundancy. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015*, page 65. 5, 101

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. 39

[Raman, 2009] Raman, T. V. (2009). Toward 2w, beyond web 2.0. *Commun. ACM*, 52(2):52–59. 9

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artif. Intelligence*, 32(1):57–95. 44, 124, 125, 126

[RFC, 1982] RFC (1982). Simple Mail Transfer Protocol. https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol. 16

[Richardson and Ruby, 2007] Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media, Inc. 19

[Rothermel and Harrold, 1997] Rothermel, G. and Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210. 103

[Rothermel et al., 2002] Rothermel, G., Harrold, M. J., Von Ronne, J., and Hong, C. (2002). Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249. 102

[Ruth et al., 2007] Ruth, M., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., and Tu, S. (2007). Towards automatic regression test selection for web services. In *31st Annual International Computer Software and Applications Conference, 2007. COMPSAC 2007.*, volume 2, pages 729–736. IEEE. 104

[Ruth and Tu, 2007] Ruth, M. and Tu, S. (2007). A safe regression test selection technique for web services. In *Second International Conference on Internet and Web Applications and Services, 2007. ICIW'07.*, pages 47–47. IEEE. 103

[Sen and Agha, 2006] Sen, K. and Agha, G. (2006). Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, Heidelberg. Springer-Verlag. 38

[SOAP, 2007] SOAP (2007). Simple Object Access Protocol. https://www.w3.org/TR/soap12/. 10, 15

[Stumptner and Wotawa, 2001] Stumptner, M. and Wotawa, F. (2001). Diagnosing tree-structured systems. *Artificial Intelligence*, 127(1):1 – 29. 45

[ter Hofstede, 2010] ter Hofstede, A. (2010). Yet Another Workflow Language. http://yawlfoundation.org//. 20

[Tillmann and De Halleux, 2008] Tillmann, N. and De Halleux, J. (2008). Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg. Springer-Verlag. 38

[Tip, 1995] Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189. 45, 124, 125

[Tomcat, 2006] Tomcat (2006). Apache tomcat. http://tomcat.apache.org/. 76

[Travé-Massuyès et al., 2006] Travé-Massuyès, L., Escobet, T., and Olive, X. (2006). Diagnosability analysis based on component-supported analytical redundancy relations. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 36(6):1146–1160. 46

[UDDI, 2001] UDDI (2001). Universal Description Discovery and Integration. https://www.oasis-open.org/committees/uddi-spec/. 19

[Ufone SOA Integration, 2012] Ufone SOA Integration (2012). Ufone soa integration. http://www.techlogix.com/PDFs/casestudy/case%20study%20-%20BPM%20-%20SOA%20Ufone.pdf. 7

[Utting et al., 2012] Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312. 51

[van Harmelen et al., 2007] van Harmelen, F., van Harmelen, F., Lifschitz, V., and Porter, B. (2007). *Handbook of Knowledge Representation*. Elsevier Science, San Diego, USA. 47

[W3C, 2011] W3C (2011). Web services glossary. https://www.w3.org/TR/2004/NOTE-ws-gloss-2004021. 13

[Wang et al., 2008] Wang, D., Li, B., and Cai, J. (2008). Regression testing of composite service: An xbfg-based approach. In *IEEE Congress on Services Part II, 2008. SERVICES-2.*, pages 112–119. IEEE. 103

[WebSphere, 2006] WebSphere (2006). Ibm websphere process server. http://www-01.ibm.com/software/integration/wps/. 76

[Weiser, 1982] Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452. 45, 124, 125

[Wotawa, 2002] Wotawa, F. (2002). On the relationship between model-based debugging and program slicing. *Artif. Intell.*, 135(1-2):125–143. 45, 47, 125

[Wotawa et al., 2012] Wotawa, F., Nica, M., and Moraru, I. (2012). Automated debugging based on a constraint model of the program and a test case. *J. Log. Algebr. Program.*, pages 390–407. 45, 126, 129, 130

[Wotawa et al., 2013] Wotawa, F., Schulz, M., Pill, I., Jehan, S., Leitner, P., Hummer, W., Schulte, S., Hoenisch, P., and Dustdar, S. (2013). Fifty shades of grey in SOA testing. In *9th Workshop on Advances in Model Based Testing; 2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 154–157. 5, 50, 69

[WSDL, 2001] WSDL (2001). Web Services Description Language. https://www.w3.org/TR/wsdl. xvii, 10, 16, 18

[Yan et al., 2006] Yan, J., Li, Z., Yuan, Y., Sun, W., and Zhang, J. (2006). Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06*, pages 75–84. IEEE Computer Society. 41

[Yan et al., 2009] Yan, Y., Dague, P., Pencolé, Y., and Cordier, M. (2009). A model-based approach for diagnosing fault in web service processes. *Int. J. Web Service Res.*, 6(1):87–110. 47, 126

[Yang et al., 2005] Yang, Y., Tan, Q., and Xiao, Y. (2005). Verifying web services composition based on hierarchical colored petri nets. In *Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, IHIS '05, pages 47–54, New York, NY, USA. ACM. 40

[Yoo and Harman, 2012] Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120. 103, 104

[Yuan Yuan and Sun, 2006] Yuan Yuan, Z. L. and Sun, W. (2006). A graph-search based approach to bpel4ws test generation. In *Software Engineering Advances, International Conference on*, page 14. 41

[Zakaria et al., 2009] Zakaria, Z., Atan, R., Ghani, A. A. A., and Sani, N. F. M. (2009). Unit testing approaches for bpel: A systematic review. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 316–322, Washington, DC, USA. IEEE Computer Society. 39

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM. 104, 107, 110

[Zhang et al., 2009] Zhang, L., Hou, S.-S., Guo, C., Xie, T., and Mei, H. (2009). Time-aware test-case prioritization using integer linear programming. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 213–224. ACM. 103

[Zhang et al., 2005] Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)*, pages 33–42. 125

[Zheng et al., 2007] Zheng, Y., Zhou, J., and Krause, P. (2007). A model checking based test case generation framework forweb services. *Information Technology: New Generations, Third Int. Conference on*, 0:715–722. 39

[Zhou et al., 2007] Zhou, J., Zheng, Y., and Krause, P. (2007). An automatic test case generation framework for web services. *Journal of Software*, 2(3). 39