

Enhancing Coverage Adequacy of Service Compositions after Runtime Adaptation

DISSERTATION

Dem Wirtschaftswissenschaften der
Universität Duisburg-Essen
zur Erlangung des akademischen Grades eines
Dr. rer. nat.

eingereicht von
Osama Sammoudi
aus
Abu Dhabi

Datum der Einreichung: 2016/05/27

Tag der mündlichen Prüfung: 2016/09/15

Erstgutachter: Prof. Dr. Klaus Pohl

Zweitgutachter: Prof. Dr. Wilhelm Hasselbring

Selbständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Ich versichere, dass ich ausschließlich die angegebenen Quellen und Hilfen Anspruch genommen habe.

Essen, den 27.05.2016

(Unterschrift Osama Sammoudi)

Acknowledgements

It is time to express my deep gratitude to all people who supported me in one way or another during my PhD journey.

First, I would like to thank Prof. Dr. Klaus Pohl for giving me the great opportunity to complete my PhD under his supervision. He gave me all sorts of support, guidance, and feedback which helped me to finish my thesis work.

I am very much grateful to Dr. Andreas Metzger for his endless support, stimulating discussions, and fruitful comments during my PhD journey. I have learned much from him.

I would also like to thank all my colleagues at paluno especially at the Software Systems Engineering group. They have made a very productive and friendly work atmosphere which I enjoyed till the last day of my thesis journey.

Many thanks as well to Prof. Dr. Wilhelm Hasselbring for accepting to be the second reviewer of my thesis.

I would also like to thank my parents for their continuous emotional support, and my wife Rawan, for her patience all the time.

The research leading to these results has received funding from the European Union's Seventh Framework Programme FP7/2007-2013 under grant agreements 215483 (S-Cube) and 285248 (FIWARE).

Abstract

Runtime monitoring (or monitoring for short) is a key quality assurance technique for self-adaptive service compositions. Monitoring passively observes the runtime behaviour of service compositions. Coverage criteria are extensively used for assessing the adequacy (or thoroughness) of software testing. Coverage criteria specify certain requirements on software testing. The importance of coverage criteria in software testing has motivated researchers to adapt them to the monitoring of service composition. However, the passive nature of monitoring and the adaptive nature of service composition could negatively influence the adequacy of monitoring, thereby limiting the confidence in the quality of the service composition.

To enhance coverage adequacy of self-adaptive service compositions at runtime, this thesis investigates how to combine runtime monitoring and online testing. Online testing means testing a service composition in parallel to its actual usage and operation. First, we introduce an approach for determining valid execution traces for service compositions at runtime. The approach considers execution traces of both monitoring and (online) testing. It considers modifications in both workflow and constituent services of a service composition. Second, we define coverage criteria for service compositions. The criteria consider execution plans of a service composition for coverage assessment and consider the coverage of an abstract service and the overall service composition. Third, we introduce online-test-case prioritization techniques to achieve a faster coverage of a service composition. The techniques employ coverage of a service composition from both monitoring and online testing, execution time of test cases, and the usage model of the service composition. Fourth, we introduce a framework for monitoring and online testing of services and service compositions called PROSA. PROSA provides technical support for the aforementioned contributions.

We evaluate the contributions of this thesis using service compositions frequently used in service-oriented computing research.

Contents

Acknowledgement	v
Abstract	vii
Contents	ix
List of Tables	xv
List of Figures	xvii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	3
1.3 Goal and Research Questions	4
1.4 Thesis Contributions	5
1.5 Thesis Structure	7
2 Fundamentals	9
2.1 Service-oriented Computing (SOC)	9
2.2 Adaptation	11
2.3 Software Testing	12
2.3.1 Regression Testing	13
2.3.2 Online Testing	14
2.4 Runtime Monitoring	16
2.5 Coverage Adequacy	17
2.5.1 Testing Adequacy	17
2.5.2 Runtime Monitoring Adequacy	18
2.5.3 The Impact of Program Modifications on Coverage	18
3 Related Work	19
3.1 Evaluation Framework	20

3.2	Test and Runtime Monitoring Coverage Assessment (TMCA) . . .	22
3.2.1	Mei et al. 2008	22
3.2.2	Tsai et al. 2008	23
3.2.3	Bartolini et al. 2008, 2009, and 2011	24
3.2.4	Lubke et al. 2009	24
3.2.5	Bai et al. 2009	25
3.2.6	Hummer et al. 2011 and 2013	26
3.2.7	Bertolino et al. 2012	27
3.2.8	Ye and Jacobsen 2013	28
3.2.9	Evaluation of TMCA Contributions	29
3.3	Regression Test Selection (RTS)	30
3.3.1	Tarhini et al. 2006	30
3.3.2	Ruth et al. 2007, Ruth and Tu 2007, and Ruth 2008 . . .	30
3.3.3	Liu et al. 2007	31
3.3.4	Wang et al. 2008	32
3.3.5	Li et al. 2010 and 2012	32
3.3.6	Mei et al. 2012	33
3.3.7	Evaluation of RTS Contributions	33
3.4	Regression Test Case Prioritization (RTP)	34
3.4.1	Hou et al. 2008	34
3.4.2	Mei et al. 2009 and 2011	35
3.4.3	Chen et al. 2010	35
3.4.4	Zhai et al. 2010 and 2014	36
3.4.5	Nguyen et al. 2011	37
3.4.6	Evaluation of RTP Contributions	37
3.5	Online Testing (OT)	38
3.5.1	Deussen et al. 2003	38
3.5.2	Chan et al. 2007	38
3.5.3	Bei et al. 2007 and 2009	39
3.5.4	Hielscher et al. 2008	40
3.5.5	Greiler et al. 2009 and 2010	40
3.5.6	Dranidis et al. 2010	41
3.5.7	Angelis et al. 2011 and Bertolino et al. 2012	41
3.5.8	Lahami et al. 2013	42

3.5.9	Ali et al. 2014	43
3.5.10	Evaluation of OT Contributions	44
3.6	Joint Runtime Monitoring and Testing Efforts (JMTE)	44
3.6.1	Challagulla et al. 2007	44
3.6.2	Bai et al. 2007	45
3.6.3	Di Penta et al. 2007	46
3.6.4	Metzger et al. 2010 and Sammodi et al. 2011	47
3.6.5	Evaluation of JMTE	47
3.7	Summary	48
4	Main Contributions	49
4.1	Determining Valid Execution Traces (A)	49
4.2	Coverage Criteria (B)	52
4.3	Online-Test-Case Selection and Prioritization (D)	54
4.4	Online Testing and Runtime Monitoring Framework (E)	54
4.5	Summary	55
5	Determining Valid Execution Traces	57
5.1	Preliminaries	57
5.1.1	Execution Traces for Service Composition	58
5.1.2	Invalid Execution Traces	60
5.2	Determining Invalid Execution Traces	62
5.2.1	Algorithm for Safe Regression Test Case Selection	63
5.2.2	The Extended Algorithm	63
5.2.3	Complexity Analysis of the Extended Algorithm	67
5.3	Summary	68
6	Coverage Criteria	69
6.1	Intra-plan and Inter-plan Coverage Criteria	70
6.1.1	Preliminaries	71
6.1.2	The Local Criteria	73
6.1.3	The Global Criteria	77
6.1.4	Subsumption Relations	82
6.2	Summary	84

7	Online Test Case Selection and Prioritization	85
7.1	Online Test Case Selection	85
7.2	Information Used for Test Case Prioritization	86
7.3	Online Test Case Prioritization Techniques	88
7.3.1	Coverage-based Test Case Prioritization	90
7.3.2	Time-based Test Case Prioritization	91
7.3.3	Usage-based Test Case Prioritization	91
7.3.4	Hybrid Test Case Prioritization	94
7.4	Summary	96
8	Online Testing and Monitoring Framework	97
8.1	The PROSA Framework	97
8.2	Runtime Monitoring Module	98
8.2.1	Service Composition Monitor	98
8.2.2	Listener	99
8.3	Online Testing Module	100
8.3.1	Service Composition Tester	100
8.3.2	Service Tester	102
8.4	Data Repository Module	103
8.4.1	Coverage Data	104
8.4.2	Usage Model	104
8.4.3	Dynamic Binding Information (DBI)	105
8.5	Summary	105
9	Evaluation	107
9.1	The Goal Question Metric Paradigm	107
9.2	Evaluation of Determining Valid Execution Traces	108
9.2.1	Goals, Questions, and Metrics	108
9.2.2	Experimental Plan	110
9.2.3	Results	112
9.3	Evaluation of Coverage Criteria	115
9.3.1	Goals, Questions, and Metrics	115
9.3.2	Experimental Plan	117
9.3.3	Results	118
9.4	Evaluation of Online Test Case Selection and Prioritization . . .	122

9.4.1	Goals, Questions, and Metrics	122
9.4.2	Experimental Plan	126
9.4.3	Results	128
9.5	Threats to Validity	133
9.5.1	Construct Validity	133
9.5.2	Internal Validity	133
9.5.3	External Validity	133
10	Conclusion and Future Work	135
10.1	Summary	135
10.2	Revisiting Research Questions	137
10.3	Future Work	139
A	Detailed Results from the Evaluation	141
A.1	Results for 1000 Execution Traces	142
A.2	Results for 2000 Execution Traces	147
A.3	Results for 3000 Execution Traces	152
A.4	Results for 4000 Execution Traces	157
A.5	Results for 5000 Execution Traces	162
	References	167

List of Tables

3.1	Possible answers to the questions used in the evaluation framework	20
3.2	Evaluation of TMCA Approaches	29
3.3	Evaluation of RTS Contributions	34
3.4	Evaluation of RTP Contributions	37
3.5	Evaluation of Online Testing Contributions	44
3.6	Evaluation of Online Testing Contributions	47
6.1	Notation Summary	72
7.1	Overview of Test Case Prioritization Techniques	89
7.2	Abstract Path Probabilities in Figure 7.1	93
7.3	Concrete Path Probabilities in Figure 7.1	94
9.1	Characteristics of the service compositions used in the experiments	110

List of Figures

4.1	Main Contributions	50
6.1	Our Proposed Coverage Criteria	71
6.2	Illustration of the local criteria for <i>operation</i> coverage	73
6.3	Illustration of the global coverage criteria using the entity type <i>operatation</i>	78
6.4	Subsumption Relation between the Coverage Criteria	83
7.1	Example Usage Model and Execution Plans of a Service Com- position	92
8.1	The PROSA Framework	98
9.1	The average execution time of the algorithm vs. the <i>number of</i> <i>bindings</i> of the service compositions (Q1.2)	113
9.2	The average execution time of the algorithm vs. the <i>number of</i> <i>traces</i> of the service compositions (Q1.3)	114
9.3	Results measured with metric M2.3.	120
9.4	Results measured with metric M2.6. The service composition Supply Chain has no branches.	121
9.5	Results measured with metric M3.3.	123
9.6	Results measured with metric M3.6. Supply Chain has no branches. DSL Service has no abstract service where local branch coverage is computed.	124
9.7	APOC results of all prioritization techniques for each service composition, individually.	129
9.8	APOC results of all prioritization techniques for all service com- positions, together.	130

9.9	APBC results of all prioritization techniques for Loan Approval, DSL Service, Trip Planning, and Image Processing, individually. The service composition Supply Chain has no branches.	131
9.10	Average Percentage Branch Coverage results of all prioritization techniques for Loan Approval, DSL Service, Trip Planning, and Image Processing, together. The service composition Supply Chain has no branches.	132

Chapter 1

Introduction

1.1 Motivation

Service-oriented Computing (SOC) is a paradigm for building highly dynamic, distributed software systems known as service compositions [89, 85]. A service composition is realized by integrating individual software services, possibly provided by third parties, to build new value-added services. According to the SOC paradigm, services may separate ownership, maintenance and operation from the use of the software. Service users thus do not need to acquire, deploy, and run software because they can access its functionality remotely through service interfaces. The services used in a service composition can be dynamically discovered and selected even while the service composition is running.

Service compositions operate in a highly dynamic settings of changing business requirements, context and users, and constituent services [90]. The dynamic nature of business produces continuous pressure to reduce expenses, increase revenues, generate profits, and remain competitive. Changes in user types, preferences, and constraints, require service composition customization and personalization, as means to “adapt” the service composition to particular users. In addition, third-party services may change unpredictably after deployment which may result in fluctuations in Quality of Service (QoS) such as performance, availability and reliability. Thus, to remain sustainable, competitive, and reliable, service compositions should be equipped with mechanisms that enable them to adapt to such changes [79]. Over the past years, many efforts have been made towards adaptive service compositions. *Adaptation*

refers to the ability of a system to dynamically modify its behaviour and/or structure in response to its perception of the environment and the system itself. Extensive surveys on adaptive service compositions are provided in [90] and [71], for example.

Given the highly dynamic settings in which service compositions may operate, and the need for adapting service compositions at runtime in particular, quality assurance techniques which can be applied at runtime are essential [85]. Generally speaking, quality assurance can contribute to building confidence in software quality. Thus, runtime quality assurance can support (re-)assuring the service composition’s quality during operation after adaptation. Runtime monitoring, (online) testing, and runtime verification are key runtime quality assurance techniques.

Runtime monitoring is the dominant runtime quality assurance technique for service compositions [90, 45]. Generally speaking, runtime monitoring observes the behaviour of a running system in order to determine whether the behaviour is consistent with a given specification [31]. The observed behaviour can be expressed in the form of execution traces obtained through instrumentation. In the following, we will refer to execution traces collected by runtime monitoring as *monitoring traces*.

Software testing is a widely-used technique for assuring the quality of traditional software at design-time. The goal of testing is to systematically *execute* the software in order to uncover failures [83, 72, 88, 46]. The software is executed with input data, and the produced outputs are observed and examined against a given specification. Software testing can also be performed after deployment, in parallel to the normal use and operation of the software, known as *online testing* (see [15, 2, 25, 6, 10, 33, 52, 80, 99, 37, 60]).

Besides runtime monitoring and online testing, runtime verification may also be applied to verify the software properties against a given specification [41, 40]. Yet, we focus in this thesis only on runtime monitoring and online testing.

Coverage criteria are extensively used for assessing the adequacy (or thoroughness) of software testing [111]. Coverage criteria explicitly specify certain requirements on testing to be satisfied. Typically, coverage criteria target error-prone aspects or parts of the software. For example, the statements or the branches of the software’s control flow. To this end, coverage criteria are

intuitively appealing, since it is clear that a test suite cannot find errors in software code not executed (i.e., covered) by the test suite [56]. In this view, coverage criteria and the associated coverage assessment contribute to building confidence in the quality of a tested software.

1.2 Problem Statement

The importance and the widespread of coverage criteria and coverage assessment in traditional software testing have motivated researchers to apply them to the runtime of service composition. In particular, Bertolino et al. [16] proposed assessing the adequacy of runtime monitoring of service composition. To this end, a set of monitoring traces is used as reference for coverage assessment. A set of monitoring traces is thus considered *adequate*, if the traces cover all parts or aspects of the service composition which need to be covered, according to the employed coverage criterion.

A distinguishing nature of runtime monitoring is that it does not stimulate the application. Instead, runtime monitoring is limited to *passively* observing the execution of the application. Thus, runtime monitoring is “passive” by its very nature; it can only report what has occurred [16]. In fact, runtime monitoring approaches are sometimes called *passive testing* [13].

The passive nature of runtime monitoring implies that the collected monitoring traces might not be adequate according to the employed coverage criteria, thereby, limiting the confidence in service composition’s quality. This is due to the fact that the collected monitoring traces can cover only those parts of the service composition which are executed when users invoke the service composition. Therefore, the available monitoring traces for the service composition might be not adequate when only few users have started to use the service composition. Additionally, there might exist no execution traces from runtime monitoring covering the remaining parts (i.e., not-yet used) of the service composition.

Moreover, coverage measures are sensitive to modifications in the application’s code or model [38, 29, 92]. Thus, for adaptive service compositions, even if an adequate set of execution traces is obtained, some of the execution traces might be impacted by the modifications in the service composition as a

result of adaptation. The modifications may be performed by either the service composition (i.e., self-adaptation) or by the providers of the constituent third-party services. Therefore, not considering the impact of adaptation on the achieved coverage might result in wrong insights about the service composition’s quality.

1.3 Goal and Research Questions

The problem outlined in Section 1.2 motivates the research conducted as part of this thesis work. The overall goal of the thesis is the following:

Goal:

Combining Runtime Monitoring and Online Testing to Enhance Coverage Adequacy of Self-adaptive Service Compositions at Runtime

Runtime coverage assessment for service composition requires considering the impact of runtime adaptation on the service composition’s execution traces. Online testing can be performed for obtaining additional execution traces for the service composition to enhance the runtime coverage as necessary.

Towards achieving this goal, the following research questions are investigated:

Research Question I:

How to assess coverage of self-adaptive service compositions at runtime?

Ensuring the availability of adequate execution traces for a service composition during runtime requires assessing the runtime coverage achieved from the available execution traces for the service composition. It is necessary to re-assess the runtime coverage after modifications in the service composition in order not gain wrong insights about the quality of the service composition from using the “outdated” coverage.

Additionally, due to the dynamic binding feature, self-adaptive service compositions can have a large number of potential realizations (i.e., instances), each having different implementation and behaviour. Consequently, achieving adequate coverage for all potential instances can require a large set of execution

traces. Whereas, some of the instances might never be invoked by the users of the service composition, e.g., because the instances do not meet the user constraints. Thus, it remains unclear how the runtime coverage assessment for dynamic service compositions can be performed.

Research Question II:

How to combine runtime monitoring and online testing to enhance coverage adequacy at runtime?

The result of runtime coverage assessment might indicate that inadequate coverage is achieved by the available set of execution traces. In this case, online testing can be started for obtaining additional execution traces, and thus, enhancing coverage adequacy.

During the execution of online testing, some execution traces might be collected while monitoring the actual usage of the service composition. These traces might cover parts of the service composition. However, it remains unclear how online test cases should be selected and executed in such a way that considers those “potential” runtime monitoring traces.

1.4 Thesis Contributions

To address the aforementioned research questions, the thesis provides the following five main contributions. A detailed overview of the motivation and the novelty of the contributions is provided in Chapter 4. The technical details of the contributions are provided in Chapters 5 - 9.

Contribution I:

Determining Valid Execution Traces

The first contribution of the thesis is an approach for determining valid execution traces for self-adaptive service compositions at runtime. The approach employs execution traces of both (online) testing and runtime monitoring.

To compute coverage of a service composition, invalid execution traces are not considered. Invalid execution traces are execution traces which cover entities which are modified or impacted by modifications in the service composition. If invalid execution traces were considered, wrong insights about the service composition quality might be obtained. Therefore, the approach considers modifications which might result in invalid execution traces at two levels: *workflow* and *concrete services*.

For determining invalid execution traces, the approach extends an existing graph-walk algorithm from regression testing. Where existing graph-walk algorithms employed the control-flow graphs of programs, the approach extended these algorithms to the control-flow graph of service compositions to consider concrete service bindings.

Contribution II:

Coverage Criteria

The second contribution of the thesis is a set of specific coverage criteria for self-adaptive service compositions. The criteria consider the actual execution plans of a service composition as reference for coverage assessment. Execution plans specify the binding of concrete services to abstract services of the service composition and thus define the combination of concrete services for a service composition [100, 108, 5]. Considering execution plans means that only the combinations of concrete services which are actually used for realizing the service composition are considered by the criteria.

In addition to execution plans, the criteria consider coverage at two different scopes: *abstract service* and *a whole service composition*. Combining execution plans and different coverage scopes, four new coverage criteria are defined.

Contribution III:

Online-Test-Case Selection and Prioritization

The third contribution of the thesis is an approach for online-test-case selection and prioritization. The goal is to achieve coverage of a service composition

at a faster rate.

Our test case prioritization approach considers the fact that coverage of service compositions can also be obtained from runtime monitoring. To this end, the approach exploits the execution time of test cases as well as the usage profile of a service composition [99].

Contribution V:

Online Testing and Runtime Monitoring Framework

The fourth contribution of the thesis is an online testing and runtime monitoring framework called PROSA [99]. PROSA introduces the idea of exploiting synergies between runtime monitoring and online testing, thereby achieving a better coverage of service compositions. The PROSA framework collects and classifies the service compositions' execution traces as runtime monitoring traces or online testing traces. The classification allows traceability between existing test cases and their executions traces. The classification further allows deriving the usage profile of a service composition considering only runtime monitoring traces.

Contribution VI:

Empirical Evaluation

The last contribution of the thesis is an empirical evaluation of the thesis contributions. The evaluation is performed through a set of controlled experiments using service compositions frequently used in service-oriented computing research.

1.5 Thesis Structure

The remainder of the thesis is organized as the following:

- Chapter 2 provides foundations and background information for the key parts of the thesis.

- Chapter 3 provides an overview and discussion of research contributions related to the different parts of the thesis work.
- Chapter 4 provides a detailed overview of the main contributions of the thesis.
- Chapter 5 presents our proposed approach for determining valid execution traces.
- Chapter 6 presents our proposed coverage criteria.
- Chapter 7 presents our proposed online-test-case selection and prioritization techniques.
- Chapter 8 illustrates the online testing and runtime monitoring framework (PROSA).
- Chapter 9 provides details about the performed evaluation.
- Chapter 10 provides a summary and conclusions.

Chapter 2

Fundamentals

2.1 Service-oriented Computing (SOC)

SOC is paradigm for developing rapid, inter-operable, revolvable and loosely coupled distributed applications [89]. SOC is based on the concept of *services*, which are autonomous and platform-independent network-available software components that can be described, published, and dynamically discovered. Following the SOC paradigm, a *service composition* (aka. service-based application) is created by integrating different services, possibly provided by third parties, to provide the complete functionalities of a composition.

Two types of service compositions can be distinguished: *service orchestration* and *service choreography*. Service orchestration represents a single centralized executable service composition (i.e., service orchestrator) that controls the interactions among different participating services. The composite service is responsible for invoking and combining the services. Service orchestration is a centralized approach for service composition. Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between different services. Service choreography is a decentralized approach for service composition.

Service Oriented Architecture (SOA) is an architectural style that supports SOC [89]. SOA defines a model for the interaction between service providers and clients. In this model, known as SOA triangle, the *provider* makes a service available to the prospective *consumers* (or clients) by publishing the service description in a *service registry*. A service registry stores the service

descriptions and acts as intermediary between the service provider and service consumer. To discover a service, the service consumer can send a query to the service registry specifying the desired service characteristics such as structural, behavioural, quality, and contextual characteristics. The service registry retrieves all the services that match the service consumer's query and returns their interface descriptions to the service consumer.

Very often, several services exist which offer the same functionality, e.g., currency conversion or flight booking [51, 23, 108, 5]. Thus, for a given service specification a large number of services which match the specification might be discovered.

A service composition consists of a set of tasks (or abstract services), each one associated with required functionality, and their interaction, i.e., control and data flow between those tasks. Each task in the service composition is realized by binding it to an appropriate service which implements its functionality.

The fact that there exist many candidate service implementations for one task requires to make selection between them [51, 23, 108, 5]. The selection is typically not done arbitrarily, but to maximize and/or limit some special properties of the service composition. One of the most important properties is the Quality of Service (QoS). For instance, selecting the cheapest service, the fastest, or maybe a compromise between the two. Performing the selection for whole service composition results in the so called *execution plan* of service composition.

Execution plans specify the binding of concrete services to abstract services of a service composition [100, 108, 5]. Often, different execution plans for the same service composition are defined in order to address different user groups with varying end-to-end requirements, such as performance or availability.

The binding between the abstract services and the selected concrete services can be made at design-time, but often the binding is delayed to deployment and even runtime [91]. Additionally, re-binding of concrete services might occur at runtime in case of adaptation [35].

2.2 Adaptation

Over the past years, many efforts have been made towards adaptive service compositions. *Adaptation* refers to the ability of a system to dynamically modify its behaviour and/or structure in response to its perception of the environment and the system itself [27, 30].

Adaptation can thus happen to achieve different goals, including [90]: (1) optimizing the service composition even if it runs correctly; (2) repairing faults; (3) modifying the service composition in response to changes in its environment; (4) preventing future faults; (5) extending the functionality of the service composition. Adaptation can be accomplished by different means including re-configuration, re-binding, re-execution, or re-planning.

Moreover, adaptation can occur at different levels including the *service composition model* and *service composition instance* (e.g., service composition customized to a particular user). Adapting the service composition model is called service composition evolution and will be the primary focus of this thesis. In the following, “adaptation” and “evolution” will be used interchangeably.

The adaptation capabilities introduced in the literature fall into the following two major clusters: *reactive* adaptation and *proactive* adaptation [90, 79]. Reactive adaptation refers to the case in which the system is modified in response to deviations in system quality, i.e., failures that are actually observed by the users of the system. Repair and/or compensation activities have to be executed as part of the adaptation in order to mitigate the effects of those failures; e.g., the user is paid a compensation, or certain service invocations are rolled back. Besides leading to additional costs due to such compensations, reactive adaptation may have a severe impact on how a system can respond to changes [78, 52]. As examples, the execution of reactive adaptation activities on the running system can considerably increase execution time and therefore reduce the overall performance of the running system, or an adaptation of the system might not be possible at all, e.g., because the system has already terminated in an inconsistent state.

Proactive adaptation refers to the case in which the need for adaptation is anticipated and thus preventive action can be taken to avoid failures. Proactive adaptation is thus based on “short-term” predictions, i.e., forecasting imminent

failures that require an adaptation of the running system. One class of proactive adaptation aims to execute countermeasures to compensate the impact of *actual* service failures before they negatively impact the system quality. Another class of proactive adaptation aims to execute activities to avoid the impact of *predicted* service failures. Such type of proactive adaptation, allows modifying the system even before a faulty service is actually executed. If the system is able to predict a service failure (which did not yet occur), and to predict that this failure may impact the service quality, the system can be modified before the faulty service is executed.

2.3 Software Testing

Software testing (or testing for short) is widely used in industry as a key quality assurance activity [13].

The goal of testing is to systematically *execute* the software (the test object) in order to uncover failures [83, 72, 88, 46]. During testing, the test object is executed with input data, or a set of test cases (also called *test suite*), and the produced outputs are observed and examined. The observed outputs can deviate from the expected outputs with respect to functionality as well as quality (e.g., performance or availability). When the observed output deviates from the expected output, a *failure* is uncovered. Because testing requires the execution of the software, it is considered to belong to *dynamic analysis techniques* [50].

It is infeasible (except for trivial cases) to test all potential inputs of the test object, therefore a sub-set of all potential inputs has to be determined for testing. The quality of the tests strongly depends on how well this sub-set covers the test object. Ideally this sub-set should include concrete inputs that are representative for all potential inputs (even those which are not tested) and it should include inputs that – with high probability – uncover failures. However, since choosing such an ideal sub-set is typically infeasible, it is important to employ other quality assurance techniques which complement testing [46]. In this thesis, we focus only on runtime monitoring (see Section 2.4), as a dynamic analysis technique for complementing testing.

2.3.1 Regression Testing

A tested object (e.g., software) can be subject to frequent modifications for numerous reasons such as to add new functionalities, fix bugs, or for optimization purposes. Regression testing is a testing activity performed when changes are made to a tested software. It involves testing the modified software with some test cases in order to re-establish confidence that the software will perform according to the (possibly modified) specification [62]. During regression testing a set of test cases already executed may be available for reuse. The most straight forward regression testing technique is to repeat all the test cases. However, this technique can be costly both in terms of time and resources. Additionally, not all test cases can be re-executed. Therefore, researchers and practitioners select a subset of the test cases to reduce the cost of regression testing known as *regression test case selection*.

Regression Test Case Selection (RTS)

The goal of RTS is to select a subset of the original test cases to establish confidence that the software was modified correctly and that its functionality has been preserved [93]. A wide range of techniques have been proposed for RTS (see [18, 107] for recent surveys). Rothermel and Harrold [93] formally define RTS as follows:

Definition 2.1 (Regression Test Selection Problem). **Given:** the program P , the modified version of P , P' , and a test suite, T .
Problem: Find a subset of T , T' , with which to test P' .

One family of RTS techniques, called *safe* RTS, aims to select the test cases that are *modification-traversing*. A test case is modification-traversing if it will traverse a new or modified part of the modified software, or it previously traversed a deleted part that was in the original software [93]. If a technique selects all modification-traversing test cases, then it is considered safe [93]. Achieving safety can be at the expense of the precision of the technique, where the technique may select test cases that may not expose faults in the modified software. In [93], Rothermel and Harrold present a framework for analysing

RTS techniques according to certain criteria including *safety* and *precision*. The framework is also used in [107] and [18].

Regression Test Case Prioritization (RTP)

Although RTS can help to reduce the number of test cases to be re-run, the number of test cases may still be high, while the allocated time for regression testing is constrained. To this end, testing researchers and practitioners apply another activity during regression testing process. The activity, known as *test case prioritization*, tries to schedule (order) test cases such that those which are more important, by some measure (e.g., early fault detection), are executed earlier in the regression testing process [39]. A wide range of test case prioritization techniques have been proposed for regression testing (see [107] for a recent survey).

We adopt the following popular definition for the test case prioritization problem [95]:

Definition 2.2 (Test Case Prioritization Problem). **Given:** T , a test suite, PT , the set of permutations of T , and f , a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'' (T'' \in PT) (T'' \neq T') [f(T') \geq f(T'')])$.

In this definition, PT represents the set of all possible prioritizations (orderings) of T , and f is a function that, applied to any such ordering, yields an award value for that ordering.

2.3.2 Online Testing

In [85] the general need for runtime quality assurance techniques for service compositions is motivated, focusing on automated techniques.

The major type of runtime quality assurance techniques used today is runtime monitoring (see Section 2.4). However, as discussed in Section 2.4, the passive nature of runtime monitoring makes it less powerful than testing [13]. As result, research activities have appeared that suggest extending test activities to runtime.

Online testing means actively stimulating the application (by feeding it with dedicated test input) after deployment, in parallel to its normal use and operation. Online testing can be performed periodically, at scheduled intervals, or event driven. Online testing helps in the timely detection of functional and nonfunctional failures [15]. Nonetheless, performing online testing on services and service compositions could produce undesired side effects. In the following, we discuss these issues along with potential solutions.

Firstly, although several services are available free of charge (e.g., Google Search), or have fixed (monthly or yearly) flat fees, where the cost of service usage remains the same regardless of the number of invocations, there are also commercial services charged per invocation (i.e., pay-per-use pricing model). For such services, online testing can be costly as each test invocation can be charged.

Secondly, several services when invoked produce only responses and thus performing online testing on them does not produce side effects. Examples of such services are those performing computations such as image processing [19]. However, online testing can be problematic for services which, when tested, produce side effects such as shipping items or charging credit cards.

Thirdly, although services could be deployed on elastic infrastructures where the resources can scale to accommodate increased load, online testing could be problematic for services with limited resources. The load caused by online testing will be added to the load of normal operations, thereby degrading its performance or in worst case the availability of its resources.

To conclude, a fundamental premise for online testing is that service providers offer some practical way for their services to be tested without incurring additional costs, producing side effects, or degrading the performance of a service.

There are several ways in which this can be realized from a technical point of view [20, 21, 47, 37]. In general, it is accomplished by offering a test interface or a sandbox allowing a service to be executed in a special testing environment or configuration mode, which allows the functionality of a service to be fully exercised in isolation from the real production environment. This requires the special test-mode instance of the service which is actually executed during testing to be identical to the actual production instance that will be invoked shortly after. Furthermore, the impact of online testing on the

non-functional properties could be mitigated by trading off testing accuracy with performance [2].

Finally, we subscribe to the opinion of other researchers [15] who argue that the increased costs and efforts associated with performing online testing are compensated by a more reliable system, increased reputation and user satisfaction, and less penalties due to less violations of contracts.

2.4 Runtime Monitoring

Runtime monitoring (or monitoring for short) is an essential and wide spread dynamic quality assurance technique. Runtime monitoring observes the behaviour of a system and determines if it is consistent with a given specification of software properties [31]. Functional and quality properties are targeted by runtime monitoring [45].

One distinguishing nature of runtime monitoring is that it does not stimulate the application, but is limited to passively observing the application during its execution. In contrast to testing, runtime monitoring always provides statements about the current execution (i.e., about current execution traces). Thus, runtime monitoring is “passive” by its very nature; it can only report what has occurred [16]. In fact, runtime monitoring approaches are sometimes called *passive testing* [13].

Runtime monitoring can uncover failures which have escaped testing, because the concrete input which lead to the current execution trace might have never been tested. Runtime monitoring therefore provides a complementary measure to ensure the quality of a service composition. Yet, the passive nature of runtime monitoring implies that failures are only detected as they spontaneously appear, possibly after real clients have already experienced them [90, 15].

Runtime monitoring has been applied since decades for different types of systems [13, 31]. The distinguishing features of service-oriented computing such as stringent quality requirements, loose coupling and dynamic binding of independent services [16], has renewed the interest in runtime monitoring. As a result, many different approaches and frameworks have recently been proposed for runtime monitoring of service compositions with the aims of, for instance,

supporting service composition optimization, enabling context-driven adaptation, or uncovering failures. The runtime monitoring approaches and frameworks can be distinguished using for example the type of monitored properties (functional vs. quality), the type of monitored entity (atomic vs. composite service), the used methods for collecting data, the degree of invasiveness of the monitoring technique, and the timeliness of the runtime monitoring technique in discovering anomalies [45]. A relatively recent survey which provides a detailed and comprehensive discussion of existing runtime monitoring techniques can be found in [90].

2.5 Coverage Adequacy

2.5.1 Testing Adequacy

An adequacy criterion is an essential part of any testing method [111]. Adequacy criteria specify certain requirements on test suites to be satisfied. These requirements refer to certain aspects of the software to be covered when executing the test suite. A test suite is considered *adequate*, if it covers all the aspects of the software which need to be covered, according to the criterion used [111, 16]. As motivated in Section 2.3, the goal of testing is to find as many failures as possible to increase confidence in the software quality. Thus, adequacy criteria typically target the coverage of software aspects which might contain undetected faults, such as the entities of the control-flow or data-flow graphs of the software.

Adequacy criteria can serve numerous purposes for software testing [111]. For example, a test adequacy criterion can be used as a rule to indicate whether testing can stop. When all the test requirements have been achieved no further testing is required. Otherwise, more tests need to be executed. Adequacy criteria can also guide the creation and the selection of test cases to satisfy the requirements of the adequacy criteria. Adequacy criteria provide quantitative measurements of the test quality [48, 56], i.e., its ability to detect faults. In this respect, a percentage of achieved requirements indicates the degree of adequacy of a test suite used during testing. Moreover, testing techniques are often compared in terms of their underlying adequacy criteria [111].

The role of adequacy criteria in software testing has motivated many research efforts. As a result, a large number of adequacy criteria have been proposed and studied. Zhu et al. [111] provide an excellent and comprehensive overview on the topic.

2.5.2 Runtime Monitoring Adequacy

The fundamental role of test coverage adequacy has inspired researchers to use a similar notion for runtime monitoring of service composition. In [16], Bertolinio et al. define the novel notion of *runtime monitoring adequacy*. Runtime monitoring coverage is analogous to test coverage and assesses the percentage of service composition control-flow graph entities which were covered during normal service composition execution. The assessment is performed with reference to the monitored execution traces, by measuring the coverage of the control-flow graph entities belonging to these traces. The intuition behind runtime monitoring coverage adequacy is similar to test coverage adequacy: if some entities are not covered, these might contain undetected faults.

Bertolinio et al. defined runtime monitoring adequacy over a sliding observation window over a time measurement unit, which could be either continuous (e.g., considering the traces collected in the last 120 seconds) or discrete (considering the most recent 1000 traces).

2.5.3 The Impact of Program Modifications on Coverage

Testing researchers and practitioners have early realized the sensitivity of the test coverage achieved to the modifications/changes in the program code and structure. As a result, several efforts have been made to better understand and to quantify the impact of the changes on test coverage. To this end, researchers have conducted experiments and case studies (see [38, 29, 28, 92]). Additionally, researchers have proposed models to reflect the impact of the changes on test coverage (see [42, 43, 44]).

Chapter 3

Related Work

This chapter provides an analysis of the existing research efforts which provide contributions related to the research problem and the research questions addressed in this thesis. Specifically, this chapter analyses and evaluates the existing contributions in service-oriented computing research related to:

- *Test and runtime monitoring coverage assessment:* as in our approach, we assess test and runtime monitoring coverage of service compositions at runtime.
- *Regression test selection:* as we determine invalid execution traces. The task of determining invalid execution traces is close to the regression test selection task.
- *Regression test case prioritization:* as we propose techniques for prioritizing the re-execution of test cases.
- *Online testing:* as we apply our testing approach during the operation of the service composition.
- *Joint efforts on runtime monitoring and testing:* as we use execution traces from both runtime monitoring and (online) testing for assessing coverage of service compositions, and we exploit the usage model of a service composition in test case prioritization.

The related contributions are evaluated using the framework which we introduce in Section 3.1.

3.1 Evaluation Framework

The analysis of the related work is based on an evaluation framework which we have developed for this purpose. The evaluation framework consists of concrete questions used to characterize the related research efforts based on the proposed solutions for the research problem and the research questions addressed in this thesis. Table 3.1 provides an illustration of the possible answers to the questions used in the framework.

Table 3.1: Possible answers to the questions used in the evaluation framework

	<i>Question X</i>		
	+ Yes	o Maybe	– No
<i>Research work Y</i>	<i>Research work Y</i> explicitly affirms <i>Question X</i>	<i>Research work Y</i> partially affirms and partially negates <i>Question X</i>	<i>Research work Y</i> explicitly negates <i>Question X</i>

In the following we present the concrete questions used in the framework. Additionally, we will illustrate for each question what are the conditions for affirming, partially affirming and partially negating, or negating the question.

1. **Service Orchestration (SO):** *does the contribution target service orchestration?*

Orchestrated service composition is the main subject of the thesis. The question is affirmed if the contribution is applied to a service orchestration. The question is partially affirmed and partially negated if the contribution is applied to a service composition but not service orchestration (e.g., choreography). Finally, the question is denied if the contribution is not applied to service composition but to atomic services.

2. **Dynamic Binding (DB):** *does the contribution address dynamic binding of service composition?*

The thesis provides contributions for service composition which employ

dynamic binding. The question is affirmed if the contribution provides a solution that is applied in the presence of dynamic binding. The question is partially affirmed and partially negated if the contribution does not consider dynamic binding, but the proposed solution can be applied in the presence of dynamic binding. Finally, the question is negated if the contribution cannot address dynamic binding.

3. **Test Coverage (TC):** *does the contribution introduce an approach for test coverage assessment?*

The thesis introduces coverage criteria for service compositions. The question is affirmed if the contribution does assess test coverage. The question is partially affirmed and partially negated if the contribution considers test coverage but it is not of its focus. The question is negated if the contribution does not at all consider test coverage.

4. **Runtime Monitoring Coverage (MC):** *does the contribution assess the coverage achieved from runtime monitoring execution traces?*

The thesis considers the coverage achieved from both (online) testing traces and runtime monitoring execution traces. The question is affirmed if the contribution does assess the coverage achieved by runtime monitoring. The question is partially affirmed and partially negated if the contribution collects runtime monitoring execution traces but does not use it for coverage assessment. The question is negated if the contribution does not at all consider the coverage achieved by runtime monitoring execution traces.

5. **Change Impact on Coverage (CIC):** *does the contribution consider the impact of modifications in the service composition on the achieved coverage?*

For coverage assessment, the thesis considers only execution traces which are not impacted by modifications in service compositions and the constituent services. The question is affirmed if the contribution does consider the impact of modifications on the achieved coverage and proposes a solution to address this issue. The question is partially affirmed and partially negated if the contribution considers the impact of modifications on the achieved coverage, but does not solve this issue. The question is

negated if the contribution does not at all consider the impact of modifications on the achieved coverage.

6. **Execution Plan (EP):** *does the contribution leverage service composition execution plans for testing?*

The thesis exploits execution plans of service compositions for coverage assessment and online testing. The question is affirmed if the contribution does explicitly leverage execution plans for coverage assessment or for testing. The question is partially affirmed and partially negated if the contribution leverages execution plans but not for coverage assessment nor for testing. The question is negated if the contribution does not at all leverage execution plans.

7. **Runtime Solution (RT):** *has the contribution been explicitly applied at runtime during the normal operation of service composition?*

The testing activities of the thesis contributions are performed during the normal operation of service composition. The question is affirmed if the contribution is explicitly applied at runtime during the normal operation of the service composition. The question is partially affirmed and partially negated if the contribution is applied at runtime but not during the normal operation of the service composition. The question is negated in case the contribution is applied at design-time.

In the following, we use these questions to characterize the reviewed contributions.

3.2 Test and Runtime Monitoring Coverage Assessment (TMCA)

3.2.1 Mei et al. 2008

Mei et al. [75] address the challenge of testing Web Service Business Process Execution Language (WS-BPEL) process with special attention to the important role of the Extensible Markup Language (XML) and the XML Path Language (XPath) in integrating workflow steps. XPath expressions are used to extract information used as input for a service. Therefore, a mismatch

among components (e.g., extracting the wrong contents or failing to extract any content from a correct XML message) may cause a WS-BPEL application to function incorrectly. To address this challenge, Mei et al. propose a family of data-flow testing criteria to test WS-BPEL applications. To this end, Mei et al. rewrite XPath expressions using graphs, named XPath Rewriting Graphs (XRG), that make explicit different paths through an XML schema. Then, Mei et al. represent BPEL programs by building models coined X-WSBPEL, that combine the control-flow graph extracted from WS-BPEL with the XRG. For the data-flow testing, Mei et al. target the X-WSBPEL models and define a set of def-use criteria, based on variable definition and usages over XPath expressions.

Mei et al. evaluate their approach by performing experiments on eight open-source WS-BPEL applications. The experimental results show that their approach detects over 90% of all faults and uses much fewer test cases than random testing to achieve the same effectiveness.

3.2.2 Tsai et al. 2008

Tsai et al. [104] address the problem of efficiently testing large number of candidate services implementing the same service specification. To this end, Tsai et al. propose a testing technique for both atomic and composite services, inspired from the group testing which is originally developed for testing blood samples. The group testing mechanism broadcasts the test cases to all atomic services under test. A voting service, which can automatically generate an oracle for each test case based on the majority principle, collects the outputs. It compares each service output with the oracle. It then dynamically logs the number of disagreements into each service profile and uses this number to evaluate and rank the service's reliability and the test cases' effectiveness and to establish each test case's oracle. In the next testing phase, Tsai et al. apply the most effective test cases first, to reduce the testing time as a service that fails can be eliminated from further consideration. They use these results to automatically update the service and test case rankings and the oracles' reliability. Integration testing can be conducted using the same approach but with the best candidates from each atomic service only. This process greatly reduces the number of combinations. If the remaining combination is large, group testing at this level can eliminate candidate combinations. The frame-

work identifies and eliminates test cases with overlapping coverage. It also ranks newly added test cases based on their potency and re-ranks existing test cases considering updated coverage relationships and recent test results.

Experiments using independently developed Web services and developed test cases were conducted to show that the proposed approach saves testing efforts by eliminating test cases, while retaining its effectiveness.

3.2.3 Bartolini et al. 2008, 2009, and 2011

Bartolini et al. [12, 17, 11] introduce an approach that adapts the notion of coverage testing to the service-oriented domain, while preserving its key principles of loose coupling and implementation neutrality. The approach provides testers with feedback about how much a service is exercised by their tests without revealing the service internals (e.g., code). This is achieved through the addition of an intermediary service that provides the coverage information to the testers through dedicated interfaces. The approach thus requires third-party services to be testable (i.e., instrumented to collect coverage information) and provide this information to the intermediate services that collect this information for the testers. The proposed approach requires intermediate services to be published with an interface for the customary logging and reporting operations. Testers thus need to launch test sessions and invoke the intermediate services through their interface to obtain the coverage reports during testing the services.

Bartolini et al. [11] assess the feasibility of their approach and provide a preliminary evaluation using a case study. The results show that the proposed approach can be used to generate useful information for improving the testing of SOA orchestration, without introducing significant overhead on the services while operating with hundreds of requests.

3.2.4 Lubke et al. 2009

Lubke et al. [68] address the problem of assessing the quality of tests for Business Process Execution Language (BPEL) processes. To this end, Lubke et al. describe test coverage metrics for BPEL processes and an instrumentation strategy as a way of measuring them. Lubke et al. deal with code coverage

metrics that can be used to judge the quality of white box tests.

More specifically, Lubke et al. define Activity Coverage for BPEL, which is based on Statement Coverage known from existing approaches. Additionally, Lubke et al. present Branch Coverage for BPEL, which is also an equivalent for the already widely-used coverage metric with the same name. To account for the differences between classic programming paradigms and those present in BPEL, Lubke et al. introduce three coverage metrics specifically targeted to BPEL. They define *Link Coverage* to address the traversal of link elements in BPEL's flow elements, and two *Handler Coverage* metrics to measure the execution of *fault* and *compensation handler* elements, two important error handling mechanisms found in BPEL.

The functionality and the value of the metrics to identify the shortcomings of the test cases have been demonstrated using a case study.

3.2.5 Bai et al. 2009

Bai et al. [9] address the problem of maintaining sensors and enforcing policies for runtime monitoring, in reaction to the updates of runtime monitoring strategies, software architecture, and business requirements. To this end, the authors present model-based approach for automatic generation of sensors and enforcement policies. Policies represent the expected software behaviour. They are enforced at runtime to ensure that the software execution conforms to the requirements. The Web services standards, Web Service Definition Language (WSDL) and Web Ontology Language for Web Services (OWL-S), are taken as the models of service interface, workflow, and semantic. In this contribution, sensors are generated based on the models from two perspectives: (1) *dependency analysis* of the data, operations, and services with respect to the ontology model of domain concepts and usage context; (2) *coverage strategies* to decide the specific logic and paths to cover and the data to capture by the runtime monitoring sensors. Bai et al. argue that as service compositions are model-based, the coverage needs to be done at the model level based on service interfaces and workflows. The following coverage models have been considered: control-based, path-based (longest path, shortest-path, and all path), data-based (data flow, in message, out message), collaboration-based (binding, session), and random.

Control-based strategies require sensors for particular types of controls such as condition, decision, iteration, and all control. Path strategies are based on the topology of the graph. Sensors are instrumented to the selected paths such as the longest, shortest and all paths of the control-flow graph. Data-based strategies are used for data flow analysis. Given a data selected for runtime monitoring, the system generates a slice of services that take the data as input/output parameters of interface operations. Sensors are instrumented to the sliced services to monitor the data transformations during the application execution. In addition, it can also select to monitor all the received data (the in message strategy) and sent data (the out message strategy) of each service. Collaboration-based strategies are defined to monitor the bindings and sessions at runtime. The binding sensors can trace the re-composition of services and compare different service implementations. The session sensors can trace the inter-operations during an established session and evaluate the availability and stability of the service. Random strategies are used to randomly select the monitored features such as data, path, and control construct based on certain coverage metrics.

Experiments are conducted on an example travel system and show that the runtime monitoring results in low overhead of the system performance, reduced effort and enhanced flexibility of sensor instrumentation.

3.2.6 Hummer et al. 2011 and 2013

Hummer et al. [54, 55] address the problem of integration testing of data-centric dynamic service compositions. Specifically they address the challenge that testing all possible runtime instances of a service composition is often infeasible, due to the fact that dynamic service binding is combinatorial in the number of services. That is, for any non-trivial binding, the number of runtime combinations may be prohibitively large.

They propose an approach which provides techniques for restricting the combinations of services and for applying the generated tests to concrete Web service composition technology. To this end, the approach considers data dependencies between services as potential points of failure and introduce the k-node data flow test coverage metric to significantly reduce the number of test combinations. This novel coverage metric expresses to what extent the

data dependencies of a dynamic service compositions are tested. On the basis of this test coverage metric, Hummer et al. formulate the problem of finding a minimal number of test cases for a service composition as a combinatorial optimization problem. To solve the optimization problem, they make use of an existing tool for coverage analysis and combinatorial test design, and provide an automated transformation from the service composition model to the tool's data model. Past instantiations of the service compositions are specified, which constitute existing solutions and can be ignored by the solver, thereby further narrowing down the search space.

A prototype implementation of the proposed approach is presented along with analysis of various performance characteristics as well as demonstration of the end-to-end practicability of the solution.

3.2.7 Bertolino et al. 2012

Bertolino et al. [16] define a general novel notion of runtime monitoring adequacy for service compositions. Runtime monitoring adequacy is analogous to testing adequacy and assesses the percentage of application entities (e.g., operations and branches) that were covered during normal service composition execution. The assessment is performed with reference to the observed execution traces by measuring coverage of the entities belonging to these traces. Coverage measures are used to determine runtime monitoring adequacy of those entities (i.e., if they reach a predefined goal). In testing, test adequacy is measured over a test session and the obtained measure refers to the executed test suite. Analogous to that, runtime monitoring adequacy is measured over a sliding observation window over a time measurement unit, which could be either continuous (e.g., considering the traces collected in the last 120 sec) or discrete (considering the most recent 1000 traces). The authors defined two specific instantiations of a monitor adequacy criterion for service choreographies: operations and branches, of an abstract behavioural model, such as a Finite State Machine.

Bertolino et al. claim that adequate runtime monitoring of service composition enhances standard monitors with a notion of coverage, and thus makes them capable to measure coverage of the relevant entities under a defined monitor adequacy criterion. Runtime monitoring adequacy helps understand the

changes in users behaviours or raising alarms for “potential” problems that should receive further investigation.

Bertolino et al. have presented a high-level architecture as well as a proof-of-concept implementation of adequate runtime monitoring framework. Furthermore, they provide a preliminary assessment of the framework using two case studies of service choreographies: Travel Reservation System and Future Market.

3.2.8 Ye and Jacobsen 2013

Ye and Jacobsen [106] address a challenge facing integration testing of service compositions which is the lack of information about test coverage caused by the use of third-party services. To this end, Ye and Jacobsen introduce an approach to design and test service compositions by providing test coverage information about the used third-party services, while limiting the amount of private information that is “leaked” from the service provider to the service consumer. The approach is based on the concept of event exposure, i.e., exposing events (state changes) about the third-party services through an event interface, which can be used to determine (real) service coverage.

By observing the events exposed by service providers in testing, service consumers can assess the test coverage using the event-based coverage criteria proposed by Ye and Jacobsen, reflecting the actual test coverage of an entire service composition. The proposed criteria adopt data-flow coverage and control-flow coverage from traditional software testing.

The approach is complemented by a test case generation technique and is backed by experimental evidence which quantitatively evaluates the proposed approach and compares it to existing work in terms of test coverage rate, effectiveness in fault-detection, and test case generation. Additionally, an analysis and evaluation of the trade off between information leakage and effectiveness of test case generation, is provided, along with the runtime complexity of the proposed algorithms and the overhead of event exposure.

3.2.9 Evaluation of TMCA Contributions

Table 3.2 provides a characterization of the reviewed TMCA contributions by answering the questions used in the evaluation framework presented in Section 3.1. Based on this characterization, we can see the following observations.

Table 3.2: Evaluation of TMCA Approaches

Research Contribution	Section	Question						
		1 (SO)	2 (DB)	3 (CA)	4 (MC)	5 (CIC)	6 (EP)	7 (RS)
Mei et al. 2008	3.2.1	+	-	+	-	-	-	-
Tsai et al. 2008	3.2.2	+	+	+	-	-	-	o
Bartolini et al. 2008, 2009, and 2011	3.2.3	+	-	+	-	-	-	o
Lubke et al. 2009	3.2.4	+	-	+	-	-	-	-
Bai et al. 2009	3.2.5	+	-	+	o	-	-	o
Hummer et al. 2011 and 2013	3.2.6	o	+	+	-	-	-	o
Bertolino et al. 2012	3.2.7	o	-	+	+	-	-	+
Ye and Jacobsen 2013	3.2.8	o	-	+	-	-	-	-

The large and wide adoption of coverage assessment and adequacy criteria for testing traditional software have motivated researchers to apply them to service compositions. As a natural consequence, novel ways of measuring coverage and novel coverage criteria have appeared which are tailored to the specifics of service compositions and their implementation languages (e.g., BPEL). Nonetheless, only few research efforts exist which apply coverage measures to execution traces obtained from runtime monitoring of service compositions [16, 9]. Moreover, only Tsai et al. [104] and Hummer et al. [54, 55] apply coverage measures and adequacy criteria to service compositions with dynamic binding. However, these approaches consider all possible combinations of services. These approaches do not take into account information about how the service composition is used, such as execution plans. Additionally, the approaches which consider runtime monitoring coverage do not take into the impact of the dynamic changes and adaptations on the achieved coverage. However, as we discussed in Section 2.5.3, coverage measures are sensitive to the changes in the application code and model. Therefore, not considering this issue might lead to wrong insights about coverage.

3.3 Regression Test Selection (RTS)

3.3.1 Tarhini et al. 2006

Tarhini et al. [101] present a safe regression testing algorithm that selects an adequate number of non-redundant test sequences aiming to find modification-related errors. A two-level abstract model represented as a Timed Labeled Transition System (TLTS) is used to specify the Web application and the behaviour of its composed components. When modifications occur, a TLTS for the modified version is constructed, reflecting all additions and/or deletions of states or edges performed by the modification.

After any of the modifications, the approach identifies all modifications done, and then selects a set of tests that may reveal modification-related errors. Moreover, it creates a new test set to test required modifications in the specification of the Web application or any of its composed components.

The approach is not backed by any experimental evaluation.

3.3.2 Ruth et al. 2007, Ruth and Tu 2007, and Ruth 2008

Ruth et al. [96, 97] address the problem of applying safe regression test selection (RTS) technique to Web services considering the distributed and autonomous nature of Web services. To this end, they propose a safe RTS technique for the verification of Web service systems in an end-to-end manner. Their approach is based on the safe RTS algorithm by Rothermel and Harrold which was developed for monolithic applications using control-flow graphs. Rather than requiring all the source code of the participating services and applications, they require control-flow graphs from every party. The granularity of the control-flow graphs can vary from very detailed to very abstract. Using hash code, the control-flow graphs will be able to indicate changes but shield the program source code. The following are the three main steps of the safe RTS technique: (1) constructing global control-flow graphs of old and new program; (2) identifying dangerous edges by comparing the corresponding control-flow graphs; (3) selecting test cases that need to be rerun.

Ruth and Tu [97] also proposed that the test cases and a table of test

cases' coverage information over the control-flow graph must also be provided along with WSDL file via WS-Metadata Exchange Framework. The required control-flow graph needs to be constructed at the statement level, meaning every node in the control-flow graph will represent a statement. These nodes will also keep a hash code of their corresponding statements. When a change is made to the system, the hash of the modified service will be different from the hash of the original service so that the RTS algorithm detects the modified parts in the service without seeing the actual source code.

Ruth et al. [96] also proposed an automated extension to their RTS technique that tackles the concurrency issues that might arise. This approach helps in solving the multiple modified service problem by using call graphs. It is possible to determine the execution order of the modified services by using the call graphs. A strategy called “downstream services first” is applied in order to achieve fault localization. In this strategy, if a fault is found in a downstream service, none of the upstream services are tested until the fault is fixed. Ruth et al. also took the situation into consideration where a service makes multiple calls to different services in parallel.

Ruth and Tu report in [98] an empirical study of the framework designed to compare the cost of performing the proposed approach and running the selected tests with the cost of running all tests without performing a selection step. The results indicate that the framework can be effective in reducing the costs of performing RTS.

3.3.3 Liu et al. 2007

Liu et al. [66] argue that in the presence of concurrent control flow, even a minor change to the synchronization condition may affect many concurrent execution paths that do not contain the changed condition. Furthermore, the impact is not only related with the changed synchronization condition, but also with other synchronization conditions. Based on this observation, they propose a regression test selection approach that addresses the issues that occur because of concurrency in BPEL regression testing. The proposed approach classifies all test paths into four categories for a regression test cases selection: reusable, modified, obsolete, and new-structural. In order to do the test path classification, the differences between the old and the new processes are firstly

identified. An impact analysis rule is proposed to analyse the affected test paths by the changes of concurrent control structures (caused by the link activity in BPEL). Based on the process change information and impact analysis result, a test path selection algorithm is used to select the test paths. Liu et al. select the impacted test paths based on the business processes comparison instead of path comparison.

The proposed approach is not backed by any experimental evaluation.

3.3.4 Wang et al. 2008

Wang et al. [105] propose an approach for generating and selecting test cases during the evolution of BPEL service composition. The approach uses control-flow analysis to identify the changes and uses Extended BPEL Flow Graph (XBFG) to compare the paths in the new service composition version and the old one. Evolution includes: process alternation, addition or deletion of abstract services, change of activities and execution sequence, and static and dynamic binding. All XBFG elements are assigned hash code, id, source, target and category fields. The change of BPEL element could be detected by comparing the hash code of elements in two XBFGs. Therefore, only those activities whose names, attributes, and sub-elements remain the same can be regarded as unchanged.

The approach is not backed by an experimental evaluation.

3.3.5 Li et al. 2010 and 2012

Li et al. [63, 64] address the problem of regression testing (i.e., test case selection and the generation of new test cases) of BPEL service composition associated with evolution and maintenance. Li et al. conclude that, changes of BPEL service composition can be classified into three kinds, (1) process change, which means the BPEL modification, (2) binding change, which means the service integrator replaces a service with another service having the same functionality and interface, (3) interface change, which means the WSDL modification. Based on this classification, Li et al. propose a solution for regression test case selection of BPEL service composition using an extensible BPEL flow graph, which can express the behaviour of the service composition. Binding informa-

tion and predicate constraints are added in graph elements for path selection and test case generation. Comparisons are made on path elements, interfaces and path conditions so that the affected paths could be selected. Some paths can be validated by selecting test cases of baseline version but others may need new test cases. The decision of whether to select or to generate is made based on the results of comparison.

The approach is backed by experimental evaluation.

3.3.6 Mei et al. 2012

Mei et al. [76] address the problem of testing adaptive systems, such as workflow-based service compositions, to allow them to dynamically switch some external services and successfully continue with execution. Specifically, the contribution considers the ultra-late binding of external Web services in a workflow-based service composition at runtime, and propose a preemptive regression testing (PRT) to address this adaptive issue. During regression testing, whenever a late-change on the service under test is detected, the approach preempts the currently executed test suite, searches for additional test cases as fixes that collectively covers the missed workflow coverage, and runs these fixes. The execution of the test suite is then continued from the preemption point, until all the test cases in the regression test suite are executed and no further late-change is detected. Mei et al. present three strategies to realize the PRT which use the difference in the workflow coverage as an indicator of whether a late-change may have occurred.

Mei et al. evaluate their approach by performing experiments on open-source WS-BPEL applications.

3.3.7 Evaluation of RTS Contributions

Table 3.3 provides a characterization of the reviewed RTS contributions by answering the questions used in the evaluation framework presented in Section 3.1.

As can be seen from the table, a handful of RTS approaches for service compositions exist. However, existing approaches for RTS of service compositions do not consider execution plans. These approaches assume only one service

Table 3.3: Evaluation of RTS Contributions

Research Contribution	Section	Question						
		1 (SO)	2 (DB)	3 (CA)	4 (MC)	5 (CIC)	6 (EP)	7 (RS)
Tarhini et al. 2006	3.3.1	o	-	-	-	-	-	o
Ruth et al. 2007, Ruth and Tu 2007, and Ruth 2011	3.3.2	+	-	-	-	-	-	o
Liu et al. 2007	3.3.3	+	-	-	-	-	-	-
Wang et al. 2008	3.3.4	+	-	-	-	-	-	-
Li et al. 2010 and 2012	3.3.5	+	-	o	-	o	-	-
Mei et al. 2012	3.3.6	+	-	+	-	+	-	o

binding for each abstract service during the selection process. Therefore, modifications in all service bindings of each abstract service are not considered during the selection.

3.4 Regression Test Case Prioritization (RTP)

3.4.1 Hou et al. 2008

Hou et al. [53] argued that quota constraint (e.g., the upper limit of the number of requests that a user can send to a Web service during a certain time range), may delay fault exposure and the subsequent debugging. Hence, they proposed a quota-constrained test-case prioritization techniques for service compositions to maximize requirement coverage testing.

The general idea behind the proposed prioritization techniques is to divide the regression testing (and thus test cases) into several time slots according to the request quotas. Thus, for each time slot, test cases are selected and then prioritized with the aim of maximizing the testing coverage per slot. After selecting test cases for each time slot, test cases are sorted using traditional test case prioritization techniques. After each test selection and prioritization per time slot, updated information regarding remaining selectable test cases, requirement coverage, and request quota are considered for scheduling test cases for the next time slot.

Hou et al. performed an experimental study (using a Travel Agent System) on their proposed techniques and compared them with the random technique, the traditional total technique, and the traditional additional technique. The results showed their prioritized test cases using their techniques are more ef-

fective for achieving total or additional branch coverage and exposing faults with the constraint of request quotas.

3.4.2 Mei et al. 2009 and 2011

Mei et al. [73, 74] presented black-box test case prioritization approach for single Web services in a service composition using WSDL-tag coverage. The approach reorders test cases based their coverage of WSDL-Tags (both ascending and descending).

Focusing on business process service compositions, in [77] Mei et al. argue that industrial service compositions are typically written in languages such as WS-BPEL, and are integrated with workflow steps and Web services via XPath and WSDL. Faults in these artefacts may cause the service composition to extract wrong data from messages, thereby leading to failures in service compositions. To this end, Mei et al. propose a family of test case prioritization techniques based on multilevel coverage model. The multilevel coverage model captures the business process, XPath, and WSDL.

Thus, Mei et al. first order test cases according to the business process coverage. To resolve ties in case of similar process coverage, the coverage of the XPath expressions are considered. To resolve ties in case of similar XPath coverage, the coverage of the WSDL schema elements are considered.

Mei et al. present extensive evaluation of their contributions by performing experiments on a set of WS-BPEL applications.

3.4.3 Chen et al. 2010

Chen et al. [26] argue that when a service evolves, service interactions and change impacts are two key points in service composition's regression testing. To this end, Chen et al. proposed a test case prioritization approach based on impact analysis of BPEL processes. In their approach, change impact analysis is performed on a weighted dependence propagation graph constructed from BPEL processes. Both data and control dependencies are considered in their analysis. The authors give weights to different impacted parts. The weight is defined as difference to the changed part. The executions of test cases are scheduled according to the impact weight.

Chen et al. demonstrate the applicability of their proposed approach by applying it on Automated Teller Machine (ATM) example.

3.4.4 Zhai et al. 2010 and 2014

Focusing on location-aware service compositions, Zhai et al. [110] suggest to incorporate service selection in test case prioritization in order to reduce the number of service invocations. Additionally, Zhai et al. propose a suite of metrics and test case prioritization techniques tailored to testing location-aware services.

The idea behind incorporating service selection in testing is to maintain – during testing process – a blacklist which contains faulty services. These faulty services are detected in previous test case executions and therefore should not be used by the service selection process. That is, if a service is found to be faulty, it should not be considered by the service selection approach. Discarding faulty services from the set of candidate service considered by the selection process reduces the number of services that need to be tested.

The test case prioritization idea is based on modelling test cases as a sequence of Global Positioning System (GPS) locations. Consequently, several location-based metrics can be applied to the test cases and the results are used for scheduling the test cases. The proposed metrics are classified into two groups: (1) input-guided and (2) Point-of-Interest (POI) aware. The input-guided techniques are based on the observation that the more diverse a sequence is, the more effective they will be in fault detection. The input-guided techniques thus prioritize a test suite in descending order of the diversity of locations in test cases. The POI-aware techniques are based on the observation that test cases that are closer to POIs or cover more POIs are more effective in fault detection and thus should be assigned highest priorities.

In [109], Zhai et al. present an extended evaluation using a controlled experiment, measuring the cost-effectiveness, and also comparing the proposed techniques in terms of several criteria.

3.4.5 Nguyen et al. 2011

Nguyen et al. [84] present an approach for prioritizing test cases for audit testing of service compositions using information retrieval (IR) techniques. In their approach, test cases are prioritized based on their relevance to the service change, based on matching service change description with the code portions exercised by the relevant test cases. The matching is performed using IR techniques.

The proposed approach assumes the availability of service change descriptions. Service change descriptions are used to define queries which are then matched – using IR techniques – with identifier documents. Identifier documents, each associated with a test case, are extracted by analysing past execution traces of test cases. The result of the match is used to search for the most relevant test cases for each change. Search results form a ranked list are used to prioritize the test cases.

Nguyen et al. perform a case study using the internally developed eBayfinder service composition, to evaluate the effectiveness of their IR-based prioritization, and compared it with coverage-based prioritization.

3.4.6 Evaluation of RTP Contributions

Table 3.4 provides a characterization of the reviewed RTP contributions by answering the questions used in the evaluation framework presented in Section 3.1.

Table 3.4: Evaluation of RTP Contributions

Research Contribution	Section	Question						
		1 (SO)	2 (DB)	3 (CA)	4 (MC)	5 (CIC)	6 (EP)	7 (RS)
Hou et al. 2008	3.4.1	+	–	◦	–	–	–	–
Mei et al. 2009 and 2011 (black box)	3.4.2	–	–	◦	–	–	–	◦
Mei et al. 2009 (white box)	3.4.2	+	–	◦	–	–	–	–
Chen et al. 2010	3.4.3	+	–	–	–	–	–	–
Zhai et al. 2010 and 2014	3.4.4	+	+	–	–	–	–	+
Nguyen et al. 2011	3.4.5	+	–	–	–	–	–	◦

The results show that there are efforts for RTP for services and service compositions. However, existing approaches for service compositions are not

applied at runtime during the operation of the service composition. Therefore, these approaches do not consider the potential coverage obtained from runtime monitoring of service compositions. Additionally, the coverage-based approaches do not use execution plans of service composition.

3.5 Online Testing (OT)

3.5.1 Deussen et al. 2003

Deussen et al. [32] address the challenge of testing a system in the production phase with the aim to monitor the system in case of dynamic changes. To this end, an architecture and a conceptual framework of an online testing concept have been presented, including the following major components.

Probes to collect events and information from the system under test (SUT) about the functional aspects of the components within the execution environment. *Injectors* to stimulate the SUT so that different aspects of its functionality can be tested. *Online test cases (OTS)* to validate and control predefined and expected sequence of events and notify errors in the case of not corresponding behaviour. *The configuration controller* to perform coordination inside the OTS as well as controlling the creation and removal of the other components created on the fly during execution (e.g., probes and OTS). *The communication bus* to realize all the distributed communication functions. *The system model* to store some knowledge about the SUT and to configure the OTS. For instance, the changes in the structure of the SUT and the dynamic behaviour of the SUT. Finally, the *gauges* to hook into the communication bus and listen to the circulated events to keep up an up-to-date abstract view of the SUT.

Deussen et al. describe an implementation using the Test and Testing Control Notation, 3rd edition (TTCN3) testing language, and demonstrate experimentally the applicability of the proposed concepts using a case study in which an active network is tested.

3.5.2 Chan et al. 2007

Chan et al. [25] address the problem of generating test oracles to define functional correctness of the service, varying according to its environment. To this

end, Chan et al. propose a metamorphic online testing, which uses oracles created during offline testing for online testing. Offline testing determines a set of successful test cases to construct their corresponding follow-up test cases for the online testing. These test cases will be executed by metamorphic services that encapsulate the services under test as well as the implementations of metamorphic relations. Thus, any failure revealed by the metamorphic testing approach will be due to the failures in the online testing mode.

Chan et al. have also experimentally evaluated the feasibility of their approach.

3.5.3 Bei et al. 2007 and 2009

Bai et al. [6, 10] address the feature of dynamic reconfiguration in service composition, which requires testing to be adaptive to the changes of the service compositions at runtime. To this end, Bai et al. propose adaptive testing approach, where tests are executed during the operation of the service composition and can be adapted to changes of the service composition's environment or of the service composition itself.

The approach, coined ConfigTest based on their previous research on the MAST (Multi-Agents-based Service Testing) framework [7], enables the online change of test organization, test scheduling, test deployment, test case binding, and service binding. ConfigTest extends MAST with a new test broker architecture, configuration management, and event-based subscription/notification mechanism. The test broker decouples test case definition from its implementation and usage, and the testing system from the services under test. The configuration management allows the test agents to bind dynamically to each other and build up their collaborations at runtime. The event mechanism enables that a change in one test artefact can be notified to all the others which subscribe their interests to the change event.

Bai et al. analyse collaboration diagrams of various testing reconfiguration scenarios and illustrate the ConfigTest approach with an example of a service composition for book ordering.

3.5.4 Hielscher et al. 2008

Hielscher et al. [52] present the PROSA (Proactive Self-adaptation of Service-Based Applications) framework, which aims to enable proactive self-adaptation. To this end, PROSA exploits online testing techniques to detect changes and deviations before they can lead to undesired consequences. Hielscher et al. illustrate the key online testing activities needed to trigger proactive adaptation, and discuss how those activities can be implemented by utilizing and extending existing testing and adaptation techniques.

Hielscher et al. illustrate how PROSA enables the proactive adaptation of a service composition using an example for travel planning. The proposed framework is not backed by experimental evaluation.

3.5.5 Greiler et al. 2009 and 2010

Greiler et al. discuss in [47] the industry challenges and open issues of integrating and testing SOA infrastructures during runtime. Those challenges and issues come from SOA literature and from the authors experience in helping a number of enterprises migrate their existing infrastructures to SOA, and extending them. The challenges include stakeholder separation, service integration, service versioning and migration, service binding and reconfiguration. The authors also provide directions for addressing the challenges. Those include extended life cycle support, central auditing, and additional information to improve test quality, automated test generation, test isolation and test awareness.

In [49], Greiler et al. present a case study demonstrating the capabilities of online testing in detecting faults/failures during system reconfiguration that could not be found offline in a test environment. The case study focuses on integration testing of service compositions. They evaluate online testing in terms of effectiveness to identify typical SOA faults: publishing, discovery, composition, and execution faults. The outcome of the case study suggests that typical reconfiguration faults can be found through online testing, and that online testing has additional value over offline testing.

3.5.6 Dranidis et al. 2010

Dranidis et al. [37] introduce an automated technique to determine when to proactively trigger adaptations in the presence of conversational services, thus avoiding costly compensation actions. A conversational service is one that only accepts specific sequences of operation invocations. Dranidis et al. advocate performing just-in-time testing; “shortly” before a conversational service is invoked for the first time within the service composition, the service is tested to detect potential deviations from the specified protocol. Stream X-machines (SXMs) have been utilized for the automated generation of test cases, which, under well-defined conditions, guarantee to reveal all inconsistencies among the implementation of a service and its expected behaviour. To ensure that just-in-time testing can be done with feasible cost and effort, as well as in reasonable time, Dranidis et al. propose a new way of reducing the number of test cases, such that they can still guarantee that the conversational service behaves as expected in the context of the concrete service composition.

The approach is backed by experimental results.

3.5.7 Angelis et al. 2011 and Bertolino et al. 2012

Angelis et al. [14] and Bertolino et al. [15] address the problem of how to verify that collaborating providers in a service federation abide legal bindings and business rules that govern the service federation. They argue that online testing of federated services can contribute to enhancing the federation’s trustworthiness and quality assurance, by verifying whether the behaviour of a service actually manifests while interacting with other services, abides by policies and complies with functional and non-functional specifications. When a service under test fails compliance validation, a federation’s trust and reputation infrastructure can use such feedback as a source for trust models. For example, a successful online test session might enhance the reputation of both the service and its provider.

To realize this vision, the authors present the Role Compliance Service On-Line Testing ((role)CAST) component, providing a testing framework for SOA infrastructures based on the Apache Axis2 (an engine for Web services). The high-level architecture of the (role)CAST component has two layers: the

framework core, which implements an Axis2 client, and a collection of Axis2 modules that implement the interface with security protocols. The layer that implements the framework core has the following main components: a test driver implementing the test planner scheduling logic and activating online testing sessions either periodically or event-driven basis, and a test robot responsible for loading the test cases from a repository as well as executing them on a specified service under test. For each online testing session, the test driver configures and runs instances of the test robot. The test robot compares the response from the interaction with the service under test to the expected result that the oracle computed for each test case. Finally, the test results repository stores the test execution logs. (role)CAST assumes the test robot collaborates via the test credential interface with an entity which authenticates it as a user of the federation.

The proposed framework is not backed by experimental evaluation.

3.5.8 Lahami et al. 2013

Lahami et al. [60] address the problem of checking behaviours and detecting faults introduced after dynamic changes in service composition's at runtime. To this end, Lahami et al. present the Runtime Testing Framework for Adaptable and Distributed Systems (RTF4ADS), which realizes a runtime testing approach.

The framework offers a generic and platform independent test execution infrastructure based on the Test and Testing Control Notation, 3rd edition (TTCN3). The framework is resource aware; the assignment of test components to execution nodes is done while considering some resource and connectivity constraints. The framework exploits dependence analysis by parsing a Component Dependency Graph to ensure that only the test cases (written in TTCN-3 language) covering affected software components or compositions by the dynamic change, selected from a repository of test cases, are executed. The execution of selected test cases is performed by a TTCN-3 test system for Runtime Testing (TT4RT). It supports four test isolation techniques: cloning the system under test (SUT), blocking the SUT, tagging test data, and built-in test.

Lahami et al. illustrated the usefulness of the proposed runtime testing

approach by applying it to a case study in the telemedicine field, called Tele-services and Remote Medical Care System.

3.5.9 Ali et al. 2014

Ali et al. [2] address the problem of achieving controllability and observability aspects when testing choreographed services. They observe that combining continuous online testing mechanisms with an appropriate governance framework could help ensure that unspecified events do not lead to undesirable interactions among services, thereby enhancing the reliability of enacted choreographies. To this end, Ali et al. have developed a framework that supports the continuous online testing of such services, supporting model-based automated test generation, storage and execution of derived test cases, and trustworthiness rating of service choreographies and subscribing services.

In the online testing process, end users trigger a choreography enactment at runtime according to the specification published by the Choreography Board, which includes the establishment of runtime binding among all services (all registered services to participate in the choreography). Testing engineers select suitable test strategies and derive concrete test cases for candidate services, and make them available in a dedicated repository so that all service providers can test the interaction of their services with those of other providers. Engineers can continuously test registered services online to periodically verify they are playing their roles as intended within the execution environment.

Ali et al. further present an online testing architecture supporting the proposed online testing process, in which all components are implemented as Web services. In this architecture, *ServicePot* constitutes the central publishing infrastructure component. It extends a service registry supporting activities, such as publishing and discovery, and augments them with governance and testing functionality. *ServicePot* includes the following three plug-in extensions. *ParTes* derives test cases for the services in a choreography, considering the structural dependencies embedded the Business Process Model and Notation (BPMN 2) control flow and between input and output messages. The *Test Driver* retrieves test suites related to a specified service in a given choreography from a repository, executes them, and generates reports. *CRank* rates both services and choreographies based on the results of online testing sessions

launched by the Test Driver.

Ali et al. have validated the online testing framework using the Passenger-friendly Airport use case scenario.

3.5.10 Evaluation of OT Contributions

Table 3.5 provides a characterization of the reviewed OT contributions by answering the questions used in the evaluation framework presented in Section 3.1.

Table 3.5: Evaluation of Online Testing Contributions

Research Contribution	Section	Question						
		1 (SO)	2 (DB)	3 (CA)	4 (MC)	5 (CIC)	6 (EP)	7 (RS)
Deussen et al. 2003	3.5.1	o	-	-	-	-	-	+
Chan et al. 2007	3.5.2	-	-	-	-	-	-	+
Bei et al. 2007 and 2009	3.5.3	o	-	-	-	-	-	+
Hielscher et al. 2008	3.5.4	o	-	-	-	-	-	+
Greiler et al. 2009 and 2010	3.5.5	o	-	-	-	-	-	+
Dranidis et al. 2010	3.5.6	o	-	-	-	-	-	+
Angelis et al. 2011 and Bertolino et al. 2012	3.5.7	-	-	-	-	-	-	+
Lahami et al. 2013	3.5.8	o	-	-	-	-	-	+
Ali et al. 2014	3.5.9	-	-	-	-	-	-	+

The table clearly shows that there are contributions which apply online testing for services and service compositions. However, the existing approaches do not target service compositions with execution plans. Additionally, existing approaches do not apply coverage measures for assessing the adequacy of testing and runtime monitoring. Since these contributions do not employ coverage assessment, the impact of changes on the achieved coverage is not considered.

3.6 Joint Runtime Monitoring and Testing Efforts (JMTE)

3.6.1 Challagulla et al. 2007

Challagulla et al. [24] present a machine learning-based reliability assessment and prediction approach for mission-critical SOA-based systems. The approach

is based on combining data from operational profile testing with runtime monitoring data during runtime. Operational profile testing was used at design-time and not at runtime.

In this approach, operational profile based testing is applied to the system and then dynamic runtime monitoring is used to enable the reliability of the system to be accurately determined as it executes. The testing and monitored data from the software system are fed to a common repository and dynamic reliability assessment is performed. Machine learning techniques are used to combine the test data and monitored data for dynamically predicting the reliability of service. The failure rate and the operational profile of the service are used to calculate the reliability for the service. An offline process periodically publishes the reliability information of the service to the service registry for supporting service discovery.

Challagulla et al. illustrate the framework using a provider's SOA-based system.

3.6.2 Bai et al. 2007

Bai et al. [8] propose an ontology-based approach for automatic testing of services. In this approach, test artefacts are specified by a Test Ontology Model (TOM) based on the Web Ontology Language (OWL). TOM serves as the contract among test components, including: test generator, test master, and test agent. The test generator parses the Web service specifications and generates test cases encoded in TOM. The test master coordinates the test plan, tasks, and execution. The test agents execute the test cases on the services.

Bai et al. propose capturing service usage profile by intercepting the messages of the Simple Object Access Protocol (SOAP), and then log the input/output data to be used in test cases. The idea was proposed for filling values for parameters into test cases generated based on the service input ontology (i.e., test case definition).

Experiments are exercised on a Travel Search service composition using a prototype tool implementation. The results show the effectiveness of the approach compared to random testing in terms of the needed number of test cases.

3.6.3 Di Penta et al. 2007

Di Penta et al. [33] present a regression testing strategy to test whether or not, during its lifetime, a (possibly new version of the) service is compliant to the behaviour specified by test and Quality of Service (QoS) assertions the integrator downloaded during the service discovery and Service Level Agreement (SLA) negotiation (along additional ones generated by the integrator from runtime monitoring data). These test cases and QoS assertions form a sort of an executable contract. They proposed to use runtime monitoring data to reduce number of invocations when executing a test suite (e.g., by mimicking service behaviour) and thus, reducing the cost of regression testing.

Di Penta et al. [34] address the problem that violation of SLAs – negotiated between service provider and service consumer – would lead to consumer dissatisfaction and loss of money for the provider. Therefore, they suggest that SLAs should be stress-tested before offering them.

To this end, Di Penta et al. propose a search-based technique for test data generation to produce test cases that are likely to violate the SLAs.

The presented solution is to use Genetic Algorithms (GAs) for the generation of test data. During test case execution, QoS parameters are observed through runtime monitoring mechanisms, and those QoS parameters are used in turn to guide the search for better test cases.

The approach for testing SLAs is presented from two perspectives: a white box approach which can be used by integrators for which the service composition's source code is available and a black box approach for all other groups which do not have access to the source code. The white box approach starts with the identification of potentially QoS-risky paths (which “are likely to exhibit high values for upper-bounded QoS attributes and low values for lower-bounded QoS attributes”). After this identification, a GA is used to generate test cases that cover the path and violate the SLA. The GA considers combinations of inputs and bindings (between abstract and concrete services). Mutation, crossover and selection operators are adapted to services. The presented fitness function considers the distance of an individual to QoS constraint violation and the coverage of the QoS-risky paths. In the black box approach QoS-risky paths cannot be computed so the fitness function only considers the QoS constraint distance.

Di Penta et al. introduced two case studies for the evaluation of the effectiveness of the presented testing approach. The results provide evidence that the approach is able to generate test cases for the violation of SLAs.

3.6.4 Metzger et al. 2010 and Sammodi et al. 2011

Focusing on Quality of Service (QoS) properties of atomic services, we [80] present an approach that augments runtime monitoring with online testing for producing failure predictions with confidence. The goal of the work is to avoid unnecessary adaptations as they can lead to severe shortcomings, such as increased costs or follow-up failures.

In [99], we introduce a framework and prototypical implementation that exploits synergies between runtime monitoring, online testing and quality prediction. The framework’s core element is a test selection activity that utilizes information about the usage of the service compositions’ services in order to select test cases that lead to better coverage of service executions, while utilizing a limited number of online test executions. The framework has preliminary experimental results.

3.6.5 Evaluation of JMTE

Table 3.6 provides a characterization of the reviewed JMTEs by answering the questions used in the evaluation framework presented in Section 3.1.

Table 3.6: Evaluation of Online Testing Contributions

Research Contribution	Section	Question						
		1 (SO)	2 (DB)	3 (CA)	4 (MC)	5 (CIC)	6 (EP)	7 (RS)
Challagulla et al. 2007	3.6.1	o	–	–	–	–	–	o
Bai et al. 2007	3.6.2	–	–	–	–	–	–	o
Di Penta et al. 2007 (regression testing)	3.6.3	–	–	–	–	–	–	o
Di Penta et al. 2007 (SLA testing)	3.6.3	+	+	–	–	–	–	o
Metzger et al. 2010 and Sammodi et al. 2011	3.6.4	–	–	–	–	–	–	+

As can be seen from the table, some efforts have envisaged the combined usage of runtime monitoring and testing for service compositions. However, using the coverage information obtained from runtime monitoring as indicator for the need for additional tests (and thus generating execution data) has not

been considered by the existing approaches. This idea has been advocated by Bertolino et al. [16], but no concrete implementation of this idea is available. Moreover, the review of the literature has showed the lack of approaches that aim at combining execution traces resulting from both testing and runtime monitoring, and using them in coverage assessment.

3.7 Summary

To conclude, the literature review has revealed relevant contributions to partial parts of the work conducted within this thesis. However, no end-end approach exists which combines online testing with runtime monitoring for enhancing coverage adequacy of self-adaptive service compositions; taking into account the impact of adaptations on the coverage of the service composition.

Chapter 4

Main Contributions

As motivated in Section 1.3, the thesis addresses two main research questions:

- RQ 1 is concerned with how to assess coverage of service compositions at runtime (*“online coverage assessment”*).
- RQ 2 is concerned with which test cases to be re-executed at runtime in case coverage is not sufficient (*“online testing”*).

The thesis provides four main contributions that address the aforementioned research questions. These contributions are visualized in Figure 4.1 (grey shapes labelled A-D), structured along the research questions.

In the following, we describe the contributions of the thesis as depicted in Figure 4.1. The technical details of each contribution are presented in the next chapters.

4.1 Determining Valid Execution Traces (A)

The first contribution of the thesis is an approach for determining valid execution traces for self-adaptive service compositions at runtime.

Considering runtime monitoring traces and online testing traces

As presented in Section 2.5, execution traces provide the basis for each kind of coverage assessment. As a key difference from existing approaches for coverage assessment (see Section 3.2), we employ execution traces of both (online) testing and runtime monitoring. In the case of (online) testing, execution traces are obtained by executing test cases on a service composition.

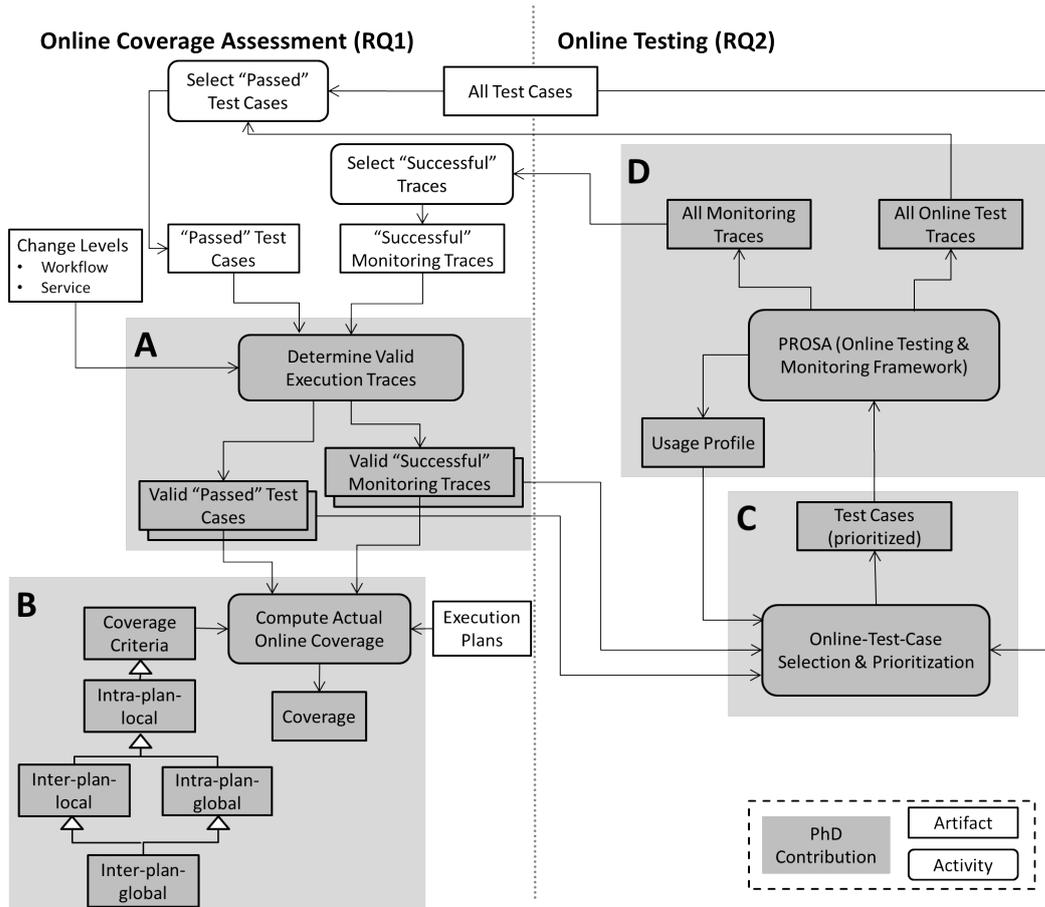


Figure 4.1: Main Contributions

In the case of runtime monitoring, execution traces are obtained when users invoke a service composition to use the functionalities offered by the service composition.

Both the workflow and concrete services of the self-adaptive service composition may evolve during runtime. Therefore, our approach considers modifications that might result in invalid execution traces at two levels: *workflow* and *concrete services*.

Considering concrete service bindings

To compute runtime coverage of a service composition achieved from runtime monitoring and online testing, valid execution traces are used. Valid execution traces are the execution traces which remain after leaving out invalid execution traces. Invalid execution traces are execution traces covering entities modified or impacted by modifications in the service composition. Invalid execution traces cannot be considered for coverage assessment. If the executions of these traces were repeated on a modified version of the service composition, they might produce different results from the results of their previous executions (i.e., on the original version of the service composition).

For determining such invalid execution traces, our approach extends an existing graph-walk algorithm developed by Rothermel and Harrold [94]. Where existing graph-walk algorithms employed the control-flow graphs of programs, we extended these algorithms to the control-flow graph of service compositions. In particular, we extended Rothermel and Harrold’s algorithm to consider concrete service bindings. A service composition is specified as a workflow of abstract services [51, 23, 108, 5]. Each abstract service specified in the workflow of a service composition may be realized by binding concrete services. Thus, in order for a service composition to become executable, for each abstract service a concrete service has to be bound. This implies that each node in a control-flow graph can have multiple implementations. Both the workflow and concrete services of a service composition may be modified during runtime, leading to invalid execution traces. If the graph-walk algorithm did not consider those concrete service bindings, the invalid the algorithm would not determine the execution traces resulting from modifications in those bindings, leading to wrong conclusions about the coverage of service composition.

More technically, our extended algorithm performs a synchronous traversal

on the control-flow graphs of the original and the modified versions of a service composition. To identify potential modifications, the algorithm syntactically compares each reached pair of nodes. In case no modifications were identified, the algorithm additionally checks whether any of corresponding concrete services is modified. The algorithm returns all execution traces that traverse the identified modifications.

4.2 Coverage Criteria (B)

The second contribution of the thesis is a set of specific coverage criteria for self-adaptive service compositions.

Considering execution plans

Different from existing coverage criteria (see Section 3.2), our criteria consider the actual execution plans of a service composition as reference for coverage assessment. Execution plans specify the binding of concrete services to abstract services of a service composition and thus define the combination of concrete services for the service composition [100, 108, 5]. Often, different execution plans for the same service composition are defined in order to cater for different user groups that have varying end-to-end requirements, such as overall performance or availability. The binding between the abstract services and the selected concrete services can occur at design-time, but often is delayed until deployment and even runtime [91]. Additionally, re-binding of concrete services might occur at runtime in case of adaptation [35].

Considering execution plans means that only the combinations of concrete services which are actually used for realizing the service composition are considered by the criteria. This results in a more realistic assessment of coverage compared with considering all potential combinations of candidate services. Some candidate concrete services and/or combinations of candidate services may never be used for realizing the service composition. Thus, from quality assurance perspective, not covering them should not be of concern. If these potential combinations would be considered, the achieved coverage might be perceived superfluous. Moreover, redundant testing might be initiated.

Considering execution plans allows us to define the following two types of criteria:

- *Intra-plan* criteria focus on coverage of a service composition within one execution plan. For example, coverage of all service composition's operations used in one execution plan.
- *Inter-plan* criteria consider coverage of a service composition achieved across all execution plans. For example, coverage of service composition's operations used in all execution plans.

Considering different coverage scopes

Traditional coverage criteria consider only a whole service composition. Since an abstract service of a service composition may have multiple bindings when considering execution plans, it would be helpful to have coverage criteria focusing on coverage of an abstract service. This scope of coverage is especially helpful for critical abstract services, for which coverage may be more important than for others. Thus, we define separate coverage criteria for an abstract service and for a whole service composition:

- *The local* criteria focus on coverage of one abstract service of a service composition.
- *The global* criteria focus on a whole service composition.

Combined coverage criteria

Considering execution plans and different coverage scopes allows us to define four new coverage criteria: (1) intra-plan-local, (2) inter-plan-local, (3) intra-plan-global, and (4) inter-plan-global. The intra-plan-local criterion is concerned with the coverage of one abstract service in one execution plan. The inter-plan-local criterion is concerned with the coverage of one abstract service in all execution plans. The intra-plan-global criterion is concerned with the coverage of the all abstract services in one execution plan. Finally, the inter-plan-global is concerned with the coverage of all abstract services in all execution plans.

4.3 Online-Test-Case Selection and Prioritization (D)

The third contribution of the thesis is an approach for online-test-case selection and prioritization. The intention is to select and prioritize test cases for re-execution in cases where coverage is insufficient.

Considering runtime monitoring coverage

Different from existing test case prioritization approaches for service compositions (see Section 3.3), our approach considers the fact that coverage of service compositions can also be obtained from runtime monitoring (see above). In order to consider potential runtime monitoring coverage, our test case selection and prioritization approach exploits the usage profile of a service composition [99]. The usage profile of a service composition provides information about how services have been used in the past (i.e., usage frequencies stored in runtime monitoring traces). Thus, the usage profile provides information about how these services are likely to be used in the future.

Inverse usage-based test selection and prioritization

Different from the traditional usage-based testing [82], we prioritize online test cases using the inverse of usage frequencies [99]. The test cases that will (1) achieve highest coverage of a service composition, and (2) cover the paths with the lowest usage frequencies, will be assigned the highest execution priority. The rationale behind this approach is that, a path with a higher usage frequency than the others is likely to be invoked, and thus covered, before the others. Thus, our approach considers this potential coverage by assigning a lower test priority to that path than to others. In case the path is covered by normal usage of a service composition, our approach does not re-test it.

4.4 Online Testing and Runtime Monitoring Framework (E)

The fourth contribution of the thesis is an online testing and runtime monitoring framework called PROSA [99].

Combination of runtime monitoring and online testing

Different from earlier efforts for service runtime monitoring and testing (see Chapter 3), PROSA introduced the idea of exploiting synergies between runtime monitoring and online testing, thereby achieving a better coverage of service compositions [99]. A similar idea for the combined usage has *later* been introduced by Bertolino et al. [16].

Differentiation between runtime monitoring traces and online testing traces

The PROSA framework collects the service compositions' execution traces. The traces are classified as runtime monitoring traces or online testing traces. The differentiation between runtime monitoring traces and online testing traces is important since: (1) it provides traceability between existing test cases and their executions traces; (2) the usage profile of a service composition is derived only from runtime monitoring traces which represent the actual usage of the service composition.

4.5 Summary

This chapter presented the four main contributions of the thesis. Technical details as well as validation of the contributions are presented in the next chapters.

Chapter 5

Determining Valid Execution Traces

As presented in Section 2.5, execution traces provide the basis for each kind of coverage assessment. For service compositions, execution traces can be obtained from both (online) testing and runtime monitoring.

Both the workflow and concrete services of the service composition may adapt during runtime. These adaptations might result in invalid execution traces. If the executions of these traces were repeated on a modified version of the service composition, they might produce different results from the results of their previous executions (i.e., on the original version of the service composition). Therefore, invalid execution traces are not to be considered for coverage assessment.

In this chapter, we present an approach for determining invalid execution traces. Our approach extends an existing graph-walk algorithm developed by Rothermel and Harrold [94]. We extended the algorithm to the control-flow graph of service compositions such that it considers concrete service bindings. By doing so, we enable the algorithm to determine all invalid execution traces including those caused by modifications in service bindings.

5.1 Preliminaries

In this section, we define *execution traces* and *invalid execution traces*. To this end, we present how execution traces for service compositions can be collected

at runtime. Additionally, we present the types of modifications based on which execution traces might become invalid execution traces.

5.1.1 Execution Traces for Service Composition

Execution traces are collected to provide information about certain aspects of program execution [31]. When used as reference for coverage assessment, these aspects include entities of program's control-flow graph (e.g., nodes, branches and paths) that have been traversed during program execution.

Definition 5.1 provides our definition for execution trace of a service composition.

Definition 5.1 (Execution Trace). Execution trace et of a service composition SC is a 4-tuple $\langle identifier, input, E, output \rangle$, where:

- $identifier$ identifies the execution trace,
- $input$ is the input for that execution,
- E is the sequence of entities traversed by that execution, and
- $output$ is the output produced by the service composition for that execution.

In structural testing, two basic control-flow graph entities are often targeted for coverage assessment: the *node* and the *branch* [111]. For a service composition, a control-flow graph corresponds to the workflow of the service composition. Thus, each node in the control-flow graph represents an abstract service or a control node. The edges of the control-flow graph represent the flow of control between the nodes.

Since for each abstract service more than one concrete service binding exist, a node in the control-flow graph may represent multiple implementations (i.e., bindings) for one abstract service. These implementations are provided by the *operations* of the service bindings. The functionalities of service composition are delivered through invocations of these operations.

Therefore, when targeting the entity *node* of service composition's control-flow graph, the coverage of all implementations of the control-flow graph nodes

should be collected. In this case, execution traces contain the sequence of operations and control nodes traversed during service composition executions.

Additionally, the fact that a node in the service composition’s control-flow graph may represent multiple operations implies that, a branch between two nodes in the service composition’s control-flow graph may represent multiple control transfers, in case one of the nodes represent abstract service. Therefore, similar to the *node*, when targeting the entity *branch*, the coverage of all implementations of the control-flow graph branches of a service composition should be collected. In this case, execution traces contain the sequence of branches traversed during service composition executions.

Types of Execution Traces

Execution traces can be obtained from performing (online) testing and/or runtime monitoring on a service composition. In the case of (online) testing, execution traces are obtained by executing test cases on a service composition. In the case of runtime monitoring, execution traces are obtained when users invoke a service composition to use the functionalities offered by the service composition.

Both types of execution traces can be collected using conventional runtime monitoring facilities of service compositions. Besides the typical usage of runtime monitoring such as to uncover failures and detect deviations from expected quality, runtime monitoring can thus be used to collect execution traces including coverage information [87, 16]. In this case, as motivated in Section 4.4, it is important to distinguish the source of the execution trace being collected (i.e., (online) testing or runtime monitoring).

Collecting Execution Traces

From a technical point of view, runtime monitoring can collect execution traces from different sources. One source could be the probes that are instrumented in the service composition. It should be noted that a heavy use of probes – which might be needed for a detailed tracking of coverage – may degrade the performance of the service composition. Another source could be the events generated by the execution engine while executing a service composition.

Some service composition execution engines (e.g., ActiveBPEL¹ and Apache ODE²) emit events whenever a change in the service composition’s execution state occurs. Execution traces can thus be obtained by analysing the events associated to these changes.

We consider only successful execution traces for coverage assessment. These traces represent executions which successfully terminated and produced the expected output. Failed executions do not increase our confidence in the quality of a service composition using coverage assessment. Therefore, the corresponding traces (i.e., failed execution traces) are not considered for coverage assessment. In the remaining of the thesis, we simply use the term “execution traces” to refer to the successful execution traces.

In addition, as motivated in Chapter 4, invalid execution traces are also not considered for coverage assessment.

5.1.2 Invalid Execution Traces

After modifying a service composition, the execution traces of the service composition can be classified into invalid execution traces and valid execution traces.

Definition 5.2 (Invalid Execution Trace). An execution trace et of a service composition SC is *invalid* w.r.t. a modified version SC' of SC , if executing SC' with the *input* of et results in entity sequence E' which is different from E of et .

Thus, for an existing execution trace of a service composition to be considered invalid, the definition requires a traversal of a different sequence of entities when the modified version of the service composition is executed using the same input of the execution trace. For instance, this can happen when the execution on the modified service composition traverses a modified entity or new entity, or when it does not traverse an entity which was initially traversed in the original service composition.

Valid execution traces are the execution traces which remain after removing the invalid execution traces.

¹<http://www.activebpel.org/>

²<http://ode.apache.org/>

Types of Modifications Leading to Invalid Execution Traces

According to Definition 5.2, the following are types of modifications to a service composition, based on which some execution traces of the service composition might become invalid:

1. *Workflow adaptation*: changing the structure of the workflow of the service composition, such as: (1) adding, removing, or modifying an abstract service, (2) adding and removing paths in the workflow, (3) modifying predicates in the workflow.
2. *Service adaptation*: updating the interface, behaviour, or performance of constituent 3rd party services used in the service composition. After service adaptation, the service might remain the same but might behave differently (e.g., produces different output using the same input).

Both workflow adaptation and service adaptation are typically performed to react to changes in business policies, user requirements, execution environment, or performance (see [90]). In contrast to workflow adaptation which is under the control of the service composition engineer, service adaptation is performed by the service provider without necessarily notifying the service consumers. Therefore, service adaptation modifications need be observed from sources such as service registries [86], service discovery frameworks [70], and service release notes [84] (see Chapter 8 for more details). Regardless of how a modification is observed, the modification typically diminishes our confidence in the quality of the impacted service. Consequently, the coverage of the paths traversing the modified service or interactions with it might become invalid, thereby decreasing the coverage of the overall service composition.

Concerning the control-flow graph of service compositions, only workflow adaptation modifications can be reflected in the control-flow graph. These modifications can be represented in three forms of *elementary* changes in the control-flow graph [58]:

- *addition* of a node to the control-flow graph, e.g., reflecting the addition of an abstract service in the service composition's workflow.
- *deletion* of a node from the control-flow graph, e.g., reflecting the removal of an abstract service in the service composition's workflow.

- *modification* of a node in the control-flow graph, e.g., reflecting the modification of an abstract service in the service composition’s workflow.

Other types of *complex* modifications in the control-flow graph can be expressed using combinations of these elementary modifications. It should be noted that addition, deletion, or modification of directed edges of control-flow graph are always associated with addition, deletion, or modification of nodes in the control-flow graph, respectively.

Since each node in the control-flow graph may represent multiple concrete services, service adaptation modifications are not represented in the control-flow graph. However, as motivated in Section 4.1, considering these modifications is essential for determining all invalid execution traces of the service composition.

5.2 Determining Invalid Execution Traces

For determining invalid execution traces, our approach extends an existing graph-walk algorithm developed by Rothermel and Harrold [94] for safe regression test case selection. Where existing graph-walk algorithms employed the control-flow graphs of programs, we extended these algorithms to the control-flow graph of service compositions.

In particular, we extended Rothermel and Harrold’s algorithm to consider concrete service bindings. As motivated in Section 4.1, in order for a service composition to become executable, for each abstract service a concrete service has to be bound. This implies that each node in a control-flow graph can have multiple implementations. As discussed in Section 5.1.2, both the workflow and concrete services of a service composition may be modified during runtime, leading to invalid execution traces. If those concrete service bindings were not considered in the graph-walk algorithm, the invalid execution traces resulting from modifications in those bindings would not be determined, leading to wrong conclusions about the coverage of service composition.

5.2.1 Algorithm for Safe Regression Test Case Selection

Rothermel and Harrold’s algorithm for safe regression test case selection uses the control-flow graph for representing the original and modified versions of a program code. To select test cases to be rerun in regression testing, the algorithm performs a synchronous depth-first traversal on the control-flow graph of the original program and the control-flow graph of the modified version of the program. When two nodes are found to be lexicographically different³, the algorithm includes the incoming edge of the node from the control-flow graph of the original program into the so called *dangerous entity set*, and selects the test cases that cover such an edge for retesting. The algorithm selects those test cases from the *edge* coverage matrix; which stores information about which edges in the control-flow graph are traversed by which test cases, when the set of test cases are executed on the program.

Rothermel and Harrold’s algorithm accounts for all structural changes we consider in our approach; i.e., the addition, deletion and modification of nodes in the control-flow graph. Moreover, any other nodes which have control dependence⁴ or data dependence⁵ on a modified node will be in the execution traces which traverse the modified node.

5.2.2 The Extended Algorithm

Our extended algorithm performs a synchronous traversal on the control-flow graphs of the original and the modified versions of a service composition. To identify potential modifications, the algorithm syntactically compares each reached pair of nodes. In case no modifications were identified, the algorithm additionally checks whether any of the corresponding concrete services is modified. The algorithm returns all execution traces that traverse the identified modifications. Algorithm 1 provides the pseudocode of the extended

³The comparison is based on the texts representing the nodes. According to Rothermel and Harrold [94], two text strings are lexicographically equivalent if their texts (ignoring extra white space characters when not contained in character constants) are identical.

⁴Control dependence represents the case that one node may affect the execution of another node.

⁵Data dependence represents the case that one node defines a value to a variable and another node may use it, and no intermediate node manipulates the variable.

algorithm.

Algorithm 1 Pseudocode for Our Algorithm for Determining Invalid Execution Traces

Input: G, G' ▷ The control-flow graphs of the original and modified versions of a service composition

Input: DBI ▷ Information about modified concrete services

Output: ET' ▷ A set of invalid execution traces

- 1: $ET' \leftarrow \phi$
 - 2: $E \leftarrow G.entryNode$ ▷ The entry node of G
 - 3: $E' \leftarrow G'.entryNode$ ▷ The entry node of G'
 - 4: $COMPARE(E, E', DBI)$ ▷ Invoke function $COMPARE$
 - 5: **return** ET'
-

Algorithm 1 takes as input: (1) the control-flow graph of the original service composition (CFG); (2) the control-flow graph of the modified version of the service composition (CFG'); (3) information about modifications in concrete services bound to the abstract services (DBI). Algorithm 1 returns the set of invalid execution traces ET' .

DBI is a table for bookkeeping the modifications with respect to the bound operations of each abstract service in the service composition. For each candidate service binding, DBI stores the operations used in the service composition and whether the operations were modified. Any new operations used in the service composition are added in DBI . DBI is updated to reflect the observed modifications for any of the existing operations. The update can be performed either automatically using the service monitoring techniques, or manually by the service composition engineer.

When Algorithm 1 begins, it initializes ET' to the empty set. Next, Algorithm 1 invokes the function $COMPARE$ with the entry nodes of CFG and CFG' (i.e., E and E' , respectively), and DBI . Function 2 provides the pseudocode for $COMPARE$.

First, $COMPARE$ determines whether any pair of nodes (say, N and N'), which have been simultaneously reached, have successors whose labels differ (i.e., lexicographically inequivalent), along pairs of identically labelled edges. If $COMPARE$ finds such nodes, it adds to ET' the execution traces that cover

Function 2 Pseudocode for the COMPARE Function

```

1: function COMPARE( $N, N', DBI$ )   ▷  $N$  and  $N'$  are nodes in  $G$  and  $G'$ 
2:   for each successor  $C$  of  $N \in G$  do
3:      $L \leftarrow$  the label on edge  $\langle N, C \rangle$  OR  $e$  if  $\langle N, C \rangle$  is unlabelled
4:      $C' \leftarrow$  the node in  $G'$  such that  $\langle N', C' \rangle$  has the label  $L$ 
5:     if  $C$  is not marked  $C$ -visited then
6:       if NOT LEquivalent( $C, C'$ ) then
7:          $ET' \leftarrow ET' \cup$  getExecutionTracesOnEdge( $N, C$ )
8:       else
9:          $B \leftarrow$  hasChangedBindings( $C', DBI$ )
10:      if  $B \neq \phi$  then
11:        for each binding  $b \in B$  do
12:           $ET' \leftarrow ET' \cup$  getExecutionTracesOnEdge( $N, C, b$ )
13:        end for
14:      end if
15:      COMPARE( $C, C', DBI$ )
16:    end if
17:  end if
18: end for
end function

```

the dangerous edges. A dangerous edge is the edge between the source node and its successor node (i.e., $\langle N, C \rangle$). In case the successor nodes are lexicographically equivalent, **COMPARE** continues to check whether the operations bound to the successor nodes are modified. To this end, **COMPARE** invokes itself on the successor nodes for further checking potential modifications in the successor nodes of those successors.

In some cases, we can have both a modified operation of a node N and, at the same time, modifications in the label of one of N 's successor nodes. Therefore, **COMPARE** additionally invokes itself on the successors of the reached pair of nodes *after* checking the bound operations of these successor nodes.

COMPARE uses the utility function **LEquivalent(Node C , Node C')** to determine whether a pair of nodes are lexicographically equivalent (i.e., their labels are equivalent). To determine whether any operations bound to the successor nodes are modified, **COMPARE** invokes the function **HasChangedBindings**, with the node under check C' and DBI . Function 3 provides the pseudocode for **HasChangedBindings**.

Function 3 Pseudocode for the **HasChangedBindings** Function

```

1: function HASCHANGEDBINDINGS( $N$ ,  $DBI$ )
2:    $B \leftarrow \phi$ 
3:   for each binding  $b$  of  $N$  in  $DBI$  do
4:     if  $b$  is modified then
5:        $B \leftarrow B \cup b$ 
6:     end if
7:   end for
8: end function

```

HasChangedBindings checks the status of each bound operation of N as stored in DBI , and returns the modified ones. **HasChangedBindings** thus takes as input the node for which the check is performed as well as the dynamic binding information DBI .

COMPARE uses the utility function **getExecutionTracesOnEdge(Node N , Node N)** to search for the execution traces which traverse a dangerous edge. Additionally, **COMPARE** uses the utility function **getExecutionTracesOnEdge(Node N , Node N , Binding b)** to search for the execution traces which traverse

the dangerous edge using a particular binding.

5.2.3 Complexity Analysis of the Extended Algorithm

In this section, we perform asymptotic analysis for the runtime of the extended graph-walk algorithm. In order for our extended graph-walk algorithm to be applicable for service compositions at runtime, the runtime of the algorithm is a relevant aspect. To support timely adaptation activities, the algorithm should be efficient when applied to service compositions at runtime.

Rothermel and Harrold [94] have analysed the runtime of their graph-walk algorithm using the **big-Oh** notation. Therefore, to facilitate benchmarking with the original algorithm, we adopt the same notation for our analysis.

The runtime of the extended graph-walk algorithm depends on the runtime of the calls to `COMPARE` plus the runtime of the other operations for initializing the sets. However, since the initialisations have constant time, the runtime of the algorithm is bounded by the runtime of the calls to `COMPARE`. An upper bound on the number of calls to `COMPARE` is reached when assuming that `COMPARE` can be called with each pair of nodes N and N' in the control-flow graphs G and G' , respectively.

To estimate the runtime of `COMPARE`, we initially assume that `COMPARE` does not include the extension to examine modifications in service bindings. Under this assumption, the runtime of `COMPARE` is similar to the runtime of the original algorithm of Rothermel and Harrold [94]. Rothermel and Harrold analyse the runtime of `COMPARE` as follows. A call to `COMPARE` results in at most two calls to `LEquivalent`, resulting in either a set union operation or examination of at most two successors of N and N' . The set union operation is bounded by the number of execution traces ($|ET|$) in the examined set of execution traces ET . The runtime of the function `LEquivalent` is bounded by the number of characters in the compared texts; a constant k , the maximum text length, for practical reasons [94]. Thus, the runtime of `COMPARE` is bounded by $O(k |ET|)$ for some constant k .

Therefore, given a pair of control-flow graphs G and G' containing n and n' nodes, respectively, and given a set of $|ET|$ execution traces, if `COMPARE` is called for each pair of nodes ($N \in G, N' \in G'$), the runtime of the algorithm is $O(|ET| n n')$. The assumption that `COMPARE` may be called for each pair of

nodes N and N' from G and G' applies only to graphs G and G' for which the multiply-visited-node condition holds. In these cases, the runtime of the graph-walk algorithm is $O(|ET|(\min\{n, n'\}))$.

To add the cost of the extension to the runtime of the algorithm, we assume that the number of bindings in a single node is bounded by a constant c ; a maximum number of bindings for all nodes.

Each call to COMPARE results in at most b lookups on the table DBI for the modified bindings of the examined node. This results in a runtime bounded to c for the worst case (assuming all bindings of the node are modified). Based on the value of c , additional set union operations are invoked leading to a runtime of $|ET|c$. It follows that these additional steps are bounded by $|ET|c$. Thus, the time required by the algorithm in the worst case is $O(|ET|nn'c)$. When the multiply-visited-node condition does not hold, the expression $|ET|nn'$ becomes $|ET|(\min\{n, n'\})$ leading to a worst case is of $O(|ET|(\min\{n, n'\})c)$ for the extended algorithm.

To conclude, the runtime of the algorithm is linear to the number of control-flow graph nodes when the other bounding parameters are hold constant. The runtime of the algorithm is linear to the number of bindings when the other bounding parameters are hold constant. The runtime of the algorithm is linear to the number of execution traces when the other bounding parameters are hold constant.

5.3 Summary

For computing coverage of service composition, valid execution traces are used. In this chapter we defined invalid execution traces. Additionally, we presented our algorithm for determining invalid execution traces. The algorithm extends an existing graph-walk algorithm of Rothermel and Harrold for safe regression test case selection to account for modifications in service bindings.

Chapter 6

Coverage Criteria

Traditional cover criteria are based on a static model of the program code structure (i.e., control-flow graph), which maps to a concrete implementation of the program. For instance, if the nodes in the control-flow graph are used to represent the functions of a program, then each node represents one concrete implementation of one function in the program. However, dynamic binding implies that the control-flow graph generated from the service composition code can have a large number of possible realizations, each having different implementation and behaviour. For example, an abstract service can have more than one candidate service to be bound during service execution.

The coverage criteria can be fulfilled using only one realization of the service composition. However, the remaining potential realizations remain uncovered and thus may contain undetected faults. Achieving adequate coverage for all potential realizations using the traditional coverage criteria can require a large number of test cases. Additionally, the cost of testing a service composition can be very high, since each test case can require the invocation of the service from the service provider [20]. Moreover, some of the candidate concrete services and/or combinations of the candidate services may never be used when executing the service composition. From quality assurance perspective, not covering the unused (combination of) candidate services should not be of concern. If these unused (combination of) candidate services would be considered, the achieved coverage might be perceived superfluous. Moreover, redundant testing might be initiated.

The above limitations of traditional coverage criteria call for the definition

of a set of coverage criteria more appropriate for service composition. The coverage criteria proposed in this chapter consider the actual execution plans of a service composition as reference for the coverage assessment. Execution plans specify the binding of concrete services to the abstract services defined in the service composition and thus define the combination of concrete services for a service composition [100, 108, 5]. Considering execution plans guarantees that only the combinations of concrete services actually used for realizing the service composition are considered.

6.1 Intra-plan and Inter-plan Coverage Criteria

Often, different execution plans for the same service composition are defined for different user groups with varying end-to-end requirements, such as overall performance or availability. The binding between the abstract services and the concrete services can be defined at design-time, but often the binding is delayed until deployment or even runtime [91]. Additionally, re-binding of concrete services might occur at runtime in case of adaptation [35]. Considering execution plans allows us to define the following two types of coverage criteria:

- *Intra-plan* coverage criteria focus on the coverage within one execution plan. For example, coverage of all service composition's operations defined in one execution plan.
- *Inter-plan* coverage criteria consider coverage across all execution plans. For example, coverage of service composition's operations defined in all execution plans.

Traditional coverage criteria consider only a whole service composition. An abstract service of a service composition may have multiple bindings defined in different execution plans. It would then be helpful to have coverage criteria focusing on coverage of one abstract service considering execution plans. This scope of coverage is especially helpful for critical abstract services whose coverage might be more important than for other services. Thus, we define separate coverage criteria for one abstract service and for a whole service composition considering execution plans:

- *The local* coverage criteria focus on coverage of one abstract service of a service composition.
- *The global* coverage criteria focus on a whole service composition.

Considering execution plans and different coverage scopes leads us to define four new coverage criteria: (1) intra-plan-local, (2) inter-plan-local, (3) intra-plan-global, and (4) inter-plan-global.

Figure 6.1 provides an overview of the criteria.

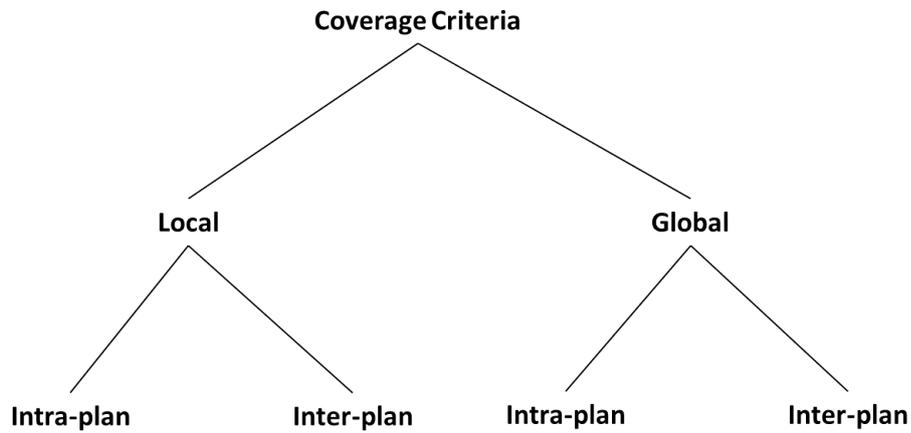


Figure 6.1: Our Proposed Coverage Criteria

6.1.1 Preliminaries

In the following, SC refers to an abstract service composition. Abstract services of SC are indexed by i , and a_i refers to an abstract service. Execution plans are indexed by j and $EPL = \{epl_j\}$ refers to a set of execution plans of SC . SC_j refers to a concrete version of SC using execution plan epl_j .

For each SC_j , the term *entity* indicates either an *operation* or a *branch* depending on what needs to be covered. E refers to the set of entities of SC using all execution plans, E_j refers to the set of entities of SC_j , e_j refers to one such *entity*, and $e_{i,j}$ refers to an entity of a_i in a concrete version SC_j . For instance, the operation bound to a_i or the branch going to a_i in SC_j .

Concerning execution traces, ET refers to a set of execution traces of SC , and ET_j refers to a set of execution traces of SC_j , $et_{i,j}$ refers to execution

trace traversing *entity* of a_i in SC_j . E_{et} refers to the entities traversed by the execution trace et .

Concerning coverage, we will use *satisfies* to indicate that 100% coverage is achieved.

Table 6.1 provides a summary of the introduced notations.

Table 6.1: Notation Summary

Group	Symbol	Description
<i>Service Composition</i>	SC	Service Composition
	i	Abstract service index
	a_i	i -th abstract service
<i>Execution Plan</i>	EPL	Set of execution plans
	j	Execution plan index
	epl_j	j -th execution plan
	SC_j	The concrete service composition using epl_j
<i>Entity</i>	E	Set of entities of SC using all execution plans
	E_j	Set of entities of SC_j
	E_i	Set of entities of a_i using all execution plans
	$e_{i,j}$	An entity of a_i in SC_j
<i>Execution Trace</i>	ET	Set of execution traces of SC
	ET_j	Set of execution traces of SC_j
	$et_{i,j}$	Execution trace traversing the <i>entity</i> of a_i in SC_j
	E_{et}	Set of entities traversed by et
<i>Coverage</i>	<i>satisfies</i>	100% coverage

6.1.2 The Local Criteria

The local coverage criteria focus on the coverage of one particular abstract service in a service composition. These criteria allow us to assess intra-plan coverage and inter-plan coverage of the entities of an individual abstract service. Figure 6.2 provides an illustration of the local criteria for both the intra-plan operation coverage and the inter-plan operation coverage.

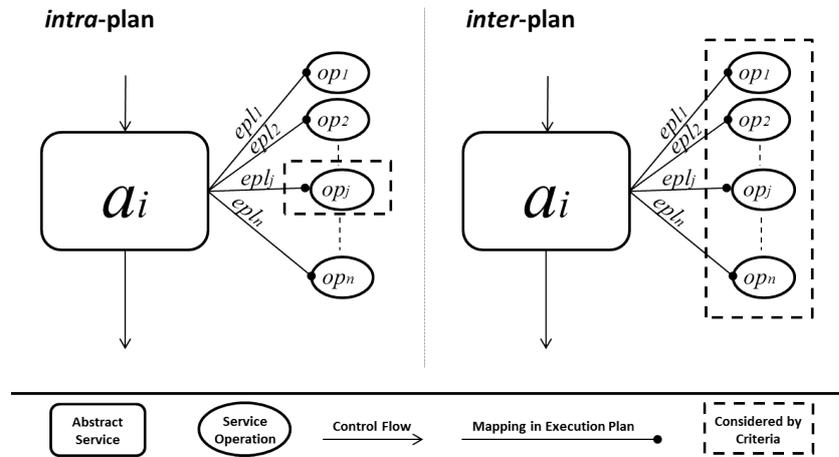


Figure 6.2: Illustration of the local criteria for *operation* coverage

The Intra-plan-local Coverage (Intra-LC) Criterion

The Intra-LC criterion is the simplest and least demanding criterion among the ones we propose. The Intra-LC criterion targets the *entity* of a particular abstract service in a service composition SC when SC is realized using a particular execution plan.

The Intra-LC criterion is defined for an individual abstract service a_i and an individual execution plan ep_j .

Definition 6.1 (Intra-plan-local Coverage Criterion). A set of execution traces ET satisfies the intra-plan-local criterion for an abstract service a_i and execution plan epl_j if and only if there is at least one execution trace et such that et traverses $e_{i,j}$ and $et \in ET_j$.

Formally:

Intra-LC. ET satisfies Intra-LC(a_i, epl_j) $\iff \exists et : e_{i,j} \in E_{et} \wedge et \in ET_j$.

The definition of the Intra-LC criterion requires that the *entity* of the abstract service in a service composition, realized using the execution plan, to be traversed by at least one execution trace of the concrete service composition.

Algorithm 4 provides the pseudocode for checking whether or not a given set of execution traces satisfies the Intra-LC criterion for a given abstract service and a given execution plan.

Algorithm 4 Checking the Satisfaction of the Intra-LC Criterion

Input: ET ▷ The set of target execution traces
Input: a_i ▷ The target abstract service
Input: epl_j ▷ The target execution plan
Output: $satisfied$ ▷ An indicator whether the criterion is satisfied

- 1: $satisfied \leftarrow FALSE$
- 2: $ET_j \leftarrow getPlanTraces(ET, epl_j)$ ▷ Select the execution traces of SC_j
- 3: $e_{i,j} \leftarrow getPlanEntity(a_i, epl_j)$ ▷ The entity of a_i in epl_j
- 4: **for all** $et \in ET_j$ **do**
- 5: **if** $e_{i,j} \in E_{et}$ **then**
- 6: $satisfied \leftarrow TRUE$
- 7: stop iteration
- 8: **end if**
- 9: **end for**
- 10: **return** $satisfied$

Algorithm 4 takes three input parameters: (1) The set of execution traces (ET) against which coverage will be assessed; (2) The abstract service (a_i) for which coverage will be assessed; (3) The execution plan (epl_j) for which the

coverage will be assessed. The algorithm returns *satisfied*; indicating whether *ET* satisfies the Intra-LC criterion w.r.t. a_i and epl_j .

Initially, the algorithm initializes *satisfied* with *FALSE*. The algorithm uses the function $getPlanTraces(ET, epl_j)$ to select the subset of execution traces ET_j of the execution plan epl_j , and the function $getPlanEntity(a_i, epl_j)$ to get the entity $e_{i,j}$ matching with the abstract service a_i and the execution plan epl_j ($e_{i,j}$ is thus the entity a_i in SC_j).

The algorithm then iterates over the set of execution traces ET_j to check if there exists at least one trace which traverses the entity $e_{i,j}$. As soon as such a trace is found, the algorithm updates *satisfied* with *TRUE* and stops the iteration. Otherwise, it continues the iteration. Once the iteration is finished, the algorithm returns *satisfied*. In case no execution trace traverses the entity $e_{i,j}$, the returned value of *satisfied* will be *FALSE*.

The Inter-plan-local Coverage (Inter-LC) Criterion

The Inter-LC criterion extends the Intra-LC criterion to consider the coverage of one abstract service of a service composition for each execution plan defined in a set of execution plans. Thus, the Inter-LC criterion targets the entities of the abstract service when using all execution plans of the service composition. Therefore, the Inter-LC criterion is defined with respect to an individual abstract service a_i of a service composition and a set execution plans EPL of the service composition.

Since the Inter-LC criterion extends the Intra-LC criterion to consider more than one execution plan, we can define the Inter-LC criterion using the definition of the Intra-LC criterion, as follows.

Definition 6.2 (Inter-plan-local Coverage Criterion). A set of execution traces ET satisfies the inter-plan-local criterion for an abstract service a_i and a set of execution plans EPL if and only if for each execution plan $epl_j \in EPL$, ET satisfies the Intra-LC criterion for the abstract service a_i .

Formally:

Inter-LC. $ET \text{ satisfies Inter-LC}(a_i, EPL) \iff \forall epl_j \in EPL, ET \text{ satisfies Intra-LC}(a_i, epl_j).$

The definition of the Inter-LC criterion requires that for each concrete service composition of an execution plan in EPL , the corresponding *entity* of the abstract service is covered by at least one execution trace.

The Inter-LC criterion targets all entities of abstract service a_i . Therefore, in addition to checking whether or not a set of execution traces satisfies the Inter-LC, we can also assess the percentage of entities of a_i covered by the execution traces, according to this criterion. To this end, Algorithm 5 provides a pseudocode for assessing the percentage of coverage achieved by a given set of execution traces, according to the Inter-LC criterion.

Algorithm 5 Inter-plan-local Coverage (Inter-LC) Assessment

Input: ET ▷ The target set of execution traces
Input: a_i ▷ The target abstract service
Input: EPL ▷ The target set of all execution plans
Output: $coverage$ ▷ The percentage of covered entities

- 1: $covered \leftarrow 0$
- 2: $coverage \leftarrow 0\%$
- 3: $E_i \leftarrow getServiceEntities(a_i, EPL)$ ▷ Get the entities of a_i
- 4: $all \leftarrow |E_i|$
- 5: **for all** $e \in E_i$ **do**
- 6: $EPL_e \leftarrow getPlans(e)$ ▷ The execution plans that use the entity e
- 7: **for all** $et \in ET$ **do**
- 8: $EPL_{et} \leftarrow getPlans(et)$ ▷ The execution plans traversed by et
- 9: **if** $(e \in E_{et}) \ \& \ (EPL_e \cap EPL_{et} \neq \phi)$ **then**
- 10: $covered \leftarrow covered + 1$
- 11: stop iteration over ET
- 12: **end if**
- 13: **end for**
- 14: **end for**
- 15: $coverage \leftarrow covered/all$
- 16: **return** $coverage$

Algorithm 5 takes three input parameters: (1) The target set of execution traces (ET) against which coverage will be assessed; (2) The target abstract service (a_i) for which coverage will be assessed; (3) The target set of execution

plans EPL . The algorithm returns the percentage of the achieved coverage (*coverage*) in the range $[0\% - 100\%]$, where 0% indicates that no entity of the abstract service is covered and 100% indicates that all entities are covered, and thus, the criterion is satisfied.

Initially, the algorithm initializes *covered* to 0 and *coverage* to 0% . The algorithm uses the function $getServiceEntities(a_i, EPL)$ to get the set of entities E_i of the abstract service a_i . The algorithm defines *all* as the total number of entities to be covered according to the Inter-LC criterion. The value of *all* in this case is the size of the set E_i .

The algorithm then iterates over the set of target entities E_i . The algorithm uses the function $getPlans(e)$ to get the set of execution plans which use the entity e . Then, the algorithm iterates over the set of execution traces ET . The algorithm uses the function $getPlans(et)$ to get the set of execution plans when the service composition is traversed by the execution trace. The algorithm then checks if the execution trace et traverses the entity e in the same concrete service composition in which e exists. In the positive case, the algorithm increments *covered* with 1 and stops the iteration over the execution traces for the entity e . Otherwise, the algorithm continues the iteration over the remaining execution traces of ET .

Once the iteration over E_i is finished, the algorithm computes the value of *coverage* by dividing *covered* over *all* and returns the value of *coverage* as a result.

6.1.3 The Global Criteria

The second set of coverage criteria is concerned with the coverage of the *overall* service composition. These criteria allow to assess intra-plan coverage and inter-plan coverage of the entities of service composition. Figure 6.3 provides an illustration of the global criteria for both the intra-plan operation coverage and the inter-plan operation coverage.

The Intra-plan-global Coverage (Intra-GC) Criterion

The Intra-GC criterion is concerned with the coverage of entities of a service composition using one execution plan. Similar to the Intra-LC criterion, the

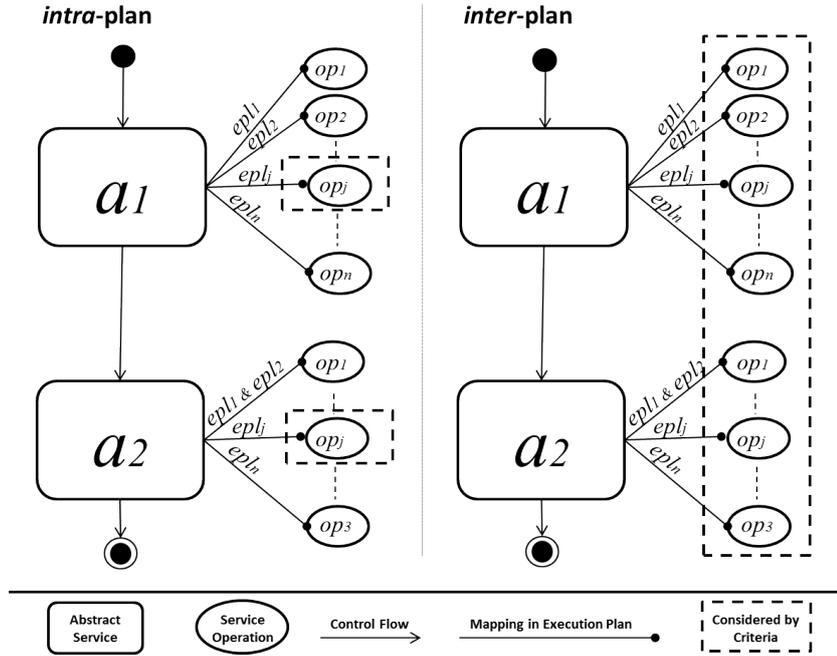


Figure 6.3: Illustration of the global coverage criteria using the entity type *operation*

Intra-GC criterion is defined with respect to a particular execution plan, as follows.

Definition 6.3 (Intra-plan-global Coverage Criterion). A set of execution traces ET satisfies the intra-plan-global criterion for execution plan epl_j if and only if for each entity of SC using epl_j , there exists at least one execution trace et which traverses the entity in SC_j .

Formally:

Intra-GC. ET satisfies Intra-GC(epl_j) $\iff \forall e \in E_j, \exists et \in ET_j : e \in E_{et}$.

Definition 6.3 requires that for one execution plan each entity of a concrete service composition is covered by at least one execution trace.

Like in the Intra-LC criterion, we can assess the percentage of entities covered by a given set of execution traces according to Intra-GC criterion, using the pseudocode in Algorithm 6.

Algorithm 6 takes two input parameters: (1) The set of target execution traces (ET); (2) The target execution plan (epl_j), and returns the percentage

Algorithm 6 Intra-plan-global Coverage (Intra-GC) Assessment

Input: ET ▷ The target set of execution traces
Input: epl_j ▷ The target execution plan
Output: $coverage$ ▷ The percentage of covered entities

- 1: $covered \leftarrow 0$
- 2: $coverage \leftarrow 0\%$
- 3: $ET_j \leftarrow getPlanTraces(ET, epl_j)$ ▷ Select the execution traces of SC_j
- 4: $E_j \leftarrow getPlanEntities(epl_j)$ ▷ Get the entities of SC_j
- 5: $all \leftarrow |E_j|$
- 6: **for all** $e \in E_j$ **do**
- 7: **for all** $et \in ET_j$ **do**
- 8: **if** $e \in E_{et}$ **then**
- 9: $covered \leftarrow covered + 1$
- 10: stop iteration over ET_j
- 11: **end if**
- 12: **end for**
- 13: **end for**
- 14: $coverage \leftarrow covered/all$
- 15: **return** $coverage$

of the achieved coverage (*coverage*).

Initially, the algorithm initializes *covered* to 0 and *coverage* to 0%. The algorithm uses the function $getPlanTraces(ET, epl_j)$ to select the subset of execution traces ET_j of SC_j . The algorithm uses the function $getPlanEntities(epl_j)$ to get the entire set of entities E_j of SC_j . The algorithm then assigns to *all* the size of the set.

For each entity in E_j , the algorithm iterates over the set of execution traces ET_j and checks if there exists a trace in this set which traverses the entity. As soon as such a trace is found, the algorithm increments *covered* with 1 and stops the iteration over ET_j . Otherwise, it continues the iteration over the remaining execution traces in ET_j .

Once all entities in E_j are visited, the algorithm computes the value of *coverage* by dividing *covered* over *all* and returns the result.

The Inter-plan-global Coverage (Inter-GC) Criterion

The Inter-GC criterion is the most comprehensive and most demanding criterion, among the ones we propose. The Inter-GC criterion is concerned with the coverage of entities of a service composition using a set of execution plans.

Definition 6.4 defines the Inter-GC criterion. The definition of the Inter-GC criterion is based on the definition of the Intra-GC criterion.

Definition 6.4 (Inter-plan-global Coverage Criterion). A set of execution traces ET satisfies the inter-plan-global criterion if and only if for each execution plan epl_j in the set of execution plans EPL , ET satisfies the Intra-GC criterion.

Formally:

Inter-GC. ET satisfies Inter-GC(EPL) $\iff \forall epl_j \in EPL, ET$ satisfies Intra-GC (epl_j).

Definition 6.4 requires using each execution plan in EPL , each entity of the concrete service composition to be covered by at least one execution trace in the same concrete service composition. Algorithm 7 provides the pseudocode for assessing the coverage achieved by a given set of execution traces, according to the Inter-GC criterion.

Algorithm 7 Inter-plan-global Coverage (Inter-GC) Assessment

Input: ET ▷ The target set of execution traces
Input: EPL ▷ The target set of execution plans
Output: $coverage$ ▷ The percentage of covered entities

- 1: $covered \leftarrow 0$
- 2: $coverage \leftarrow 0\%$
- 3: $E \leftarrow getEntities(EPL)$ ▷ Get the entities of SC in EPL
- 4: $all \leftarrow |E|$
- 5: **for all** $e \in E$ **do**
- 6: $EPL_e \leftarrow getPlans(e)$ ▷ The execution plans that use the entity e
- 7: **for all** $et \in ET$ **do**
- 8: $EPL_{et} \leftarrow getPlans(et)$ ▷ The execution plans of SC when traversed by et
- 9: **if** $(e \in E_{et}) \ \& \ (EPL_e \cap EPL_{et} \neq \phi)$ **then**
- 10: $covered \leftarrow covered + 1$
- 11: stop iteration over ET
- 12: **end if**
- 13: **end for**
- 14: **end for**
- 15: $coverage \leftarrow covered/all$
- 16: **return** $coverage$

Algorithm 7 takes two input parameters: (1) The target set of execution traces (ET); (2) The target set of execution plans entities (EPL), and returns the percentage of achieved coverage ($coverage$).

Initially, the algorithm initializes $covered$ with 0 and $coverage$ with 0%. The algorithm defines all as the total number of entities to be covered according to the Inter-GC criterion. This is equal to the size of the set E .

For each entity in E , the algorithm uses the function $getPlans(e)$ to get the set of execution plans that use the entity e , and iterates over the set of execution traces ET . The algorithm uses the function $getPlans(et)$ to get the set of execution plans of the service composition when traversed by the execution trace et . The algorithm checks if the execution trace et traverses the entity e in the concrete service composition in which the entity e exists. In the positive case, the algorithm increments $covered$ with 1 and stops the iteration over ET . Otherwise, the algorithm continues the iteration over the remaining execution traces of ET .

Once all entities are visited, the algorithm computes the value of $coverage$ by dividing $covered$ over all and returns the result.

6.1.4 Subsumption Relations

In software testing, coverage criterion X subsumes coverage criterion Y if every test suite satisfying X also satisfies Y [111]. Similarly, coverage criterion X for execution traces subsumes coverage criterion Y if every set of execution traces satisfying X also satisfies Y . Figure 6.4 shows the subsumption relation between our proposed coverage criteria.

The Intra-LC criterion is the weakest criterion among the others. This is because the coverage scope of this criterion is the entity of an individual abstract service using one execution plan. Therefore, the Intra-LC criterion does not subsume any other criterion.

A set of execution traces satisfying the Inter-LC criterion covers all entities of an individual abstract service when using all execution plans. Thus, the entity required by Intra-LC criterion is part of the super set required by the Inter-LC criterion. Therefore, the Inter-LC criterion subsumes the Intra-LC criterion. Moreover, since the Inter-LC criterion focuses on one individual abstract service and not the whole SC , the Inter-LC criterion does not subsume

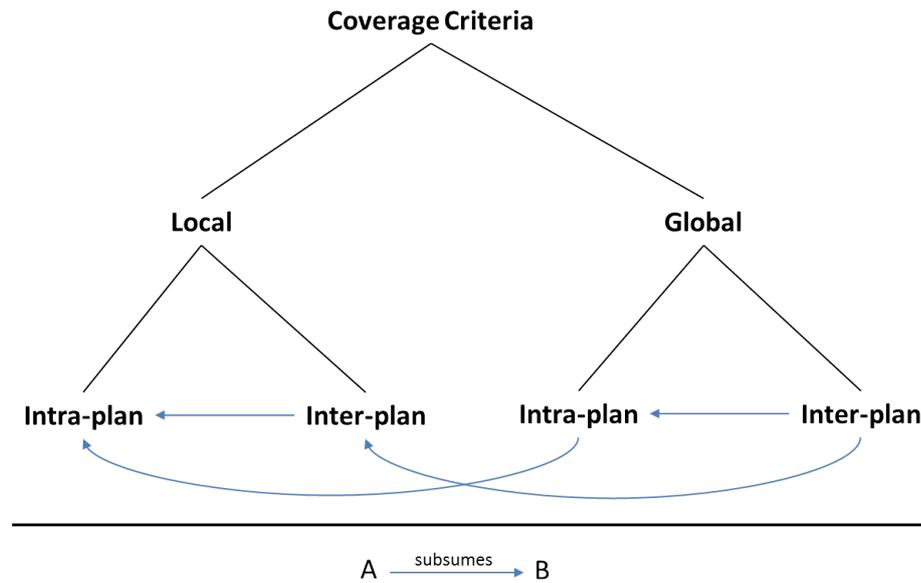


Figure 6.4: Subsumption Relation between the Coverage Criteria

any of the global criteria.

The Intra-GC criterion is weaker than its peer the Inter-GC criterion. The scope of coverage required by the Intra-GC criterion is only one execution plan. Whereas, the Inter-GC criterion goes beyond one execution plan and scopes out for the whole set of execution plans. A set of execution traces satisfying the Inter-GC criterion covers all entities of a service composition when using all execution plans. Therefore, the Inter-GC criterion subsumes the Intra-GC criterion. However, the converse does not hold. The Intra-GC criterion does not subsume the Inter-GC criterion.

Since the global criteria consider the whole service composition, both the Intra-GC criterion and the Inter-GC criterion subsume the Intra-LC criterion. Moreover, a set of execution traces satisfying the Inter-GC criterion covers all entities of a particular abstract service, thereby subsuming the Inter-LC criterion for all abstract services of the service composition. However, since the Intra-GC criterion focuses only on a single execution plan, it neither subsumes the Inter-LC criterion, nor subsumes the Inter-GC criterion.

6.2 Summary

This chapter presented our coverage criteria for self-adaptive service compositions. The proposed criteria exploit execution plans of a service composition and thus include criteria for one execution plan (i.e., intra-plan) and criteria for all execution plans (i.e., inter-plan). The proposed criteria scopes out for one abstract service and for the whole service composition. Additionally, we presented algorithms for measuring percentage of achieved coverage according to each of the criteria. Finally, we discussed the subsumption relation between the criteria.

Chapter 7

Online Test Case Selection and Prioritization

This chapter presents our approach for online test case selection and prioritization techniques. The intention is to select and prioritize test cases for re-execution in cases where coverage of a service composition is insufficient.

Our test case prioritization goal is to achieve coverage rate of the coverage criteria presented in Chapter 6 faster by combining runtime monitoring and online testing. In order to consider potential runtime monitoring coverage, our approach exploits information about the time it takes to execute test cases and the usage model of service composition.

7.1 Online Test Case Selection

In cases where the achieved coverage of the selected coverage criteria is insufficient, test cases are selected from a repository of test cases (see Chapter 4), and are prioritized for re-execution.

Test case prioritization is concerned with finding an order for the execution of test cases to achieve a certain goal. According to Rothermel et al. [95], there are many possible goals of prioritization including fast achievement of coverage which we target in our approach, Rothermel et al. state:

“Testers may wish to increase the coverage of coverable code in the system under test at a faster rate, thus allowing a code coverage criterion to be met earlier in the test process”.

Following this goal, prioritization techniques aim to schedule the test cases of a test suite such that code coverage is achieved at the fastest possible rate to reach 100% coverage soonest [65]. Alternatively, prioritizations aim to schedule the test cases of a test suite in a way to ensure the maximum possible coverage is achieved by a pre-defined cut-off point.

An increased rate of coverage provides earlier feedback and evidence about the quality of the system or the service composition (i.e., whether quality goals have been met). An increased rate of coverage is typically correlated with an increased rate of fault detection, which allows triggering adaptation of the service composition earlier.

There are several criteria for measuring the coverage rate of a prioritized test suite, the metrics used depend on the coverage criterion selected. For measuring the coverage rate of a prioritized test suite, we adapt the metrics defined by Li et al. [65] for block, decision, and statement coverage. For instance, considering statement coverage, Li et al. used APSC (Average Percentage Statement Coverage). APSC measures the rate at which a prioritized test suite covers the statements. Li et al. define APSC as the weighted average of the percentage of statement coverage over the life of the suite, as follows. Let a test suite T containing n test cases that covers a set S of m statements. Let TS_i be the first test case in the order T' of T that covers statement i . The APSC for order T' is given by the equation:

$$APSC = 1 - \frac{TS_1 + TS_2 + \dots + TS_n}{nm} + \frac{1}{2n},$$

where APSC values range from 0 to 100 and higher values imply faster (better) coverage rates.

7.2 Information Used for Test Case Prioritization

Our approach prioritizes test cases executed during the normal operation of service composition. Parallel execution of the test cases increases the load on the service composition and its constituent services. The increased load can lead to undesired side effects such as downtimes of the service composition (see [20] for more details on this issue). Therefore, we consider sequential

execution of test cases to minimize these side effects, and propose prioritization techniques for sequential execution of test cases.

An important factor in the effectiveness of a prioritization technique is the information (i.e., evidences) it utilizes [81]. Test cases and their corresponding execution traces posse the following valuable information which we utilize to achieve our prioritization goal:

- **Achievable Test Coverage.** When executed on a service composition, each test case achieves a percentage of coverage according to a certain coverage criterion. We refer to this as *achievable test coverage*. Intuitively, the test cases with a higher test coverage can contribute more to achieving faster coverage rate than the test cases with a lower test coverage. We use this information for our test case prioritization. As a basis for coverage assessment, we use the coverage criteria proposed in Chapter 6. Details of our test case prioritization are presented in Section 7.3.1.
- **Potential Monitoring Coverage.** During the online execution of a test suite for a service composition, the service composition may in addition be used by its users. By this normal usage, some parts of the service composition may be covered. I.e., normal usage contributes to the overall achievement of the coverage criteria of a service composition. We will refer to this type of coverage as *potential monitoring coverage* because it is not guaranteed that the service composition will be used during a session of online testing. The potential monitoring coverage is obtained in parallel to the test coverage achieved through online testing. Thus, combining both the achievable test coverage and the potential monitoring coverage of service composition can help achieving faster coverage rate, as some online test cases may not need to be executed, thereby reducing the overall time required to achieve the desired coverage.

The following information of the test cases can help us consider the potential monitoring coverage of a service composition:

- **Execution Time:** Test case execution time indicates the duration for completing the online execution of the test case. Using execution time, we can prioritize test cases to increase the chance for

potential monitoring coverage. Executing the test cases with the longest execution time first, expands the period in which potential monitoring coverage of other test cases might be obtained. This increases the chance for more potential monitoring coverage to be obtained. As the potential monitoring coverage occurs in parallel to the test coverage, faster coverage rate of the service composition might be achieved. Therefore, we use information about the test cases execution time in our test case prioritization. Details are presented in Section 7.3.2.

- **Path Probability:** The usage model (also called operational profile) of a service composition encodes information about the usage patterns of the service composition or parts of it. Runtime monitoring history of a service composition constitutes a rich source for building the service composition’s usage model. Using the usage model, we can compute the probabilities for the paths of the service composition potentially covered through runtime monitoring [57]. We refer to this information as *path probability*. Using path probability, we can prioritize test cases to consider the potential runtime monitoring coverage of those paths, thereby achieving coverage rate faster. Details are presented in Section 7.3.3.

In the following, we use this information for our test case prioritization techniques for service compositions.

7.3 Online Test Case Prioritization Techniques

Given any test case prioritization goal one or more prioritization techniques can be defined [95]. In this section we present test case prioritization techniques for our test case prioritization goal, which we classify into four groups: coverage-based, time-based, usage-based, and hybrid (combining information used in the other groups).

Table 7.1 provides an overview of all test case prioritization techniques presented in this chapter.

Table 7.1: Overview of Test Case Prioritization Techniques

Category	Technique	Description
Coverage-based	C1	Descending order of test cases achievable test coverage according to the Inter-plan-local coverage criterion, and pseudorandomly to resolve the tie.
	C2	Descending order of test cases achievable test coverage according to the Intra-plan-global coverage criterion, and pseudorandomly to resolve the tie.
	C3	Descending order of test cases achievable test coverage according to the Inter-plan-global criterion, and pseudorandomly to resolve the tie.
Time-based	E	Descending order of test cases execution time, and pseudorandomly to resolve the tie.
Usage-based	U	Ascending order of test cases concrete path probabilities, and pseudorandomly to resolve the tie.
Hybrid	H1	Descending order of test cases achievable test coverage. In case of a tie, descending order of test cases execution time. In case of a tie, ascending order of test cases concrete path probability.
	H2	Descending order of test case potential, and pseudorandomly to resolve the tie.

7.3.1 Coverage-based Test Case Prioritization

The coverage-based prioritization techniques that we propose are based on the coverage criteria presented in Chapter 6. The Intra-plan local coverage criterion (see Section 6.1.2) can be satisfied using only one test case. We thus propose test case prioritization techniques only for the remaining three coverage criteria.

C1: Descending Inter-plan-local Coverage

This technique is proposed for the inter-plan-local coverage criterion (see Section 6.1.2). The inter-plan-local coverage criterion is concerned with the coverage of entities of one particular abstract service using all execution plans.

The technique sorts the test cases in a descending order of their inter-plan-local coverage, and pseudorandomly in cases of tie. Test cases which cover the largest number of entities of a particular abstract service of a service composition using all execution plans, are assigned the highest priorities.

C2: Descending Intra-plan-global Coverage

This technique is proposed for the intra-plan-global coverage criterion (see Section 6.1.3). The intra-plan-global coverage criterion is concerned with the coverage of the overall service composition within one particular execution plan.

The technique sorts the test cases in descending order of their achievable intra-plan-global coverage, and pseudorandomly in cases of tie. Test cases which cover the largest number of entities of the service composition using a particular execution plan are assigned the highest execution priorities.

C3: Descending Inter-plan-global Coverage

This technique is proposed for the inter-plan-global coverage criterion (see Section 6.1.3). The inter-plan-global coverage criterion is concerned with the coverage of the overall service composition using all execution plans.

The technique prioritizes test cases according to their achievable inter-plan-global coverage, and pseudorandomly in cases of tie. Test cases that cover largest number of entities of the service composition using all execution plans, are assigned the highest priorities.

7.3.2 Time-based Test Case Prioritization

The time-based test case prioritization category is based on the execution time of test cases. For a given test case, the execution time can be estimated by summing up the response times of the concrete services along the path traversed by the test case. As we motivated in Section 7.2, test cases execution time takes into account potential runtime monitoring coverage when prioritizing test cases.

E: Descending Order of Test Case Execution Time

This technique sorts test cases in descending order of their execution time, and pseudorandomly in cases of ties. That is, the test cases which have the longest execution time will be assigned the highest execution priority.

The rationale behind this prioritization is that while executing the test cases with the longest execution time, the time in which runtime monitoring coverage can be considered increases. This facilitates the consideration of more runtime monitoring coverage.

7.3.3 Usage-based Test Case Prioritization

The usage-based test case prioritization category is based on the usage model of a service composition.

Typically, either Markov chains or flat operational profiles are used to represent usage models [57]. A Markov chain represents application states and transitions between those states, together with probabilities for those state transitions [102]. Flat operational profile is defined as a set of application operations and their occurrence probabilities [82].

Since a service composition consists of a workflow of tasks (or abstract services), a Markov chain is more appropriate for representing the usage model of a service composition than the flat operational profile. Therefore, we use a Markov chain to represent the usage model of a service composition. In this usage model, states represent the tasks (or abstract services) and transitions between the states represent the transitions between the abstract services. The transition probability between two nodes in the workflow is computed as the ratio of the number of times a transition is traversed to the total number of times all possible transitions between the nodes are traversed [57]. These

traversal counts can directly be computed from the runtime monitoring data of the service composition.

With respect to all execution plans defined for a service composition, we additionally define *execution plan probability* for a service composition as the probability that the execution plan is used to realize the service composition. Execution plan probability is computed as the ratio of the number of times the execution plan is used to the total number of times all execution plans are used. The number of times execution plans are used can directly be computed from the runtime monitoring data of a service composition.

Figure 7.1 provides an example for a usage model of a service composition with three execution plans and their probabilities.

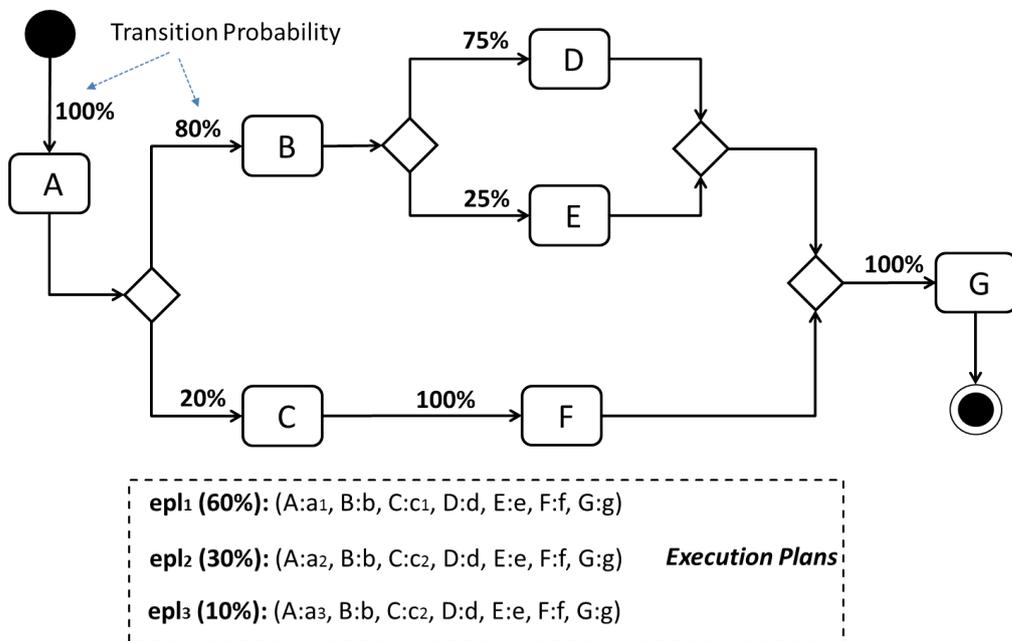


Figure 7.1: Example Usage Model and Execution Plans of a Service Composition

The example in Figure 7.1 includes 7 abstract services labelled *A* to *G*. Each transition in Figure 7.1 is annotated with a percentage indicating the transition probability. For example, the transition between the start node and the abstract service *A* has 100% probability. Another example, the transition probabilities for the transitions between the first decision node and the abstract services *B* and *C* are 80% and 20%, respectively. Moreover, the example in

Figure 7.1 has 3 execution plans epl_1 , epl_2 , and epl_3 . Each execution plan has execution plan probability denoted by $epl_x(\%)$. For example, the probability of execution plan epl_1 is 60%. In execution plan epl_1 , the abstract services A and B are bound to the concrete services a_1 and b denoted by $A : a_1$ and $B : b$, respectively.

Using the usage model and the execution plan probability, we can derive the following parameters which we use for test case prioritization:

- **Abstract Path Probability (P_{ap})** is defined as the probability that a given abstract path in the service composition's workflow is invoked. Abstract path probability is computed as the arithmetic multiplication of the transition probabilities along the abstract path [57]. For example, the probability of the abstract path (A, C, F, G) in Figure 7.1 is $100\% \times 20\% \times 100\% \times 100\% = 20\%$. Table 7.2 provides the probabilities for the three abstract paths of the example workflow.

Table 7.2: Abstract Path Probabilities in Figure 7.1

Abstract Path	Probability
$ap1$: A B D G	60%
$ap2$: A B E G	20%
$ap3$: A C F G	20%

- **Concrete Path Probability (P_{cp})**. A concrete path is a path where each abstract service is bound to a concrete service. Concrete path probability is defined as the probability that a given concrete path in the service composition's workflow is invoked. Concrete path probability is computed as follows:

$$P_{cp} = P_{ap} \times P_{epl}$$

For example, the probability of the concrete path (A:a1, C:c1, F:f, G:g) in Figure 7.1 is $20\% \times 60\% = 12\%$. Table 7.3 provides the probabilities of all concrete paths in Figure 7.1.

The idea behind the usage-based test case prioritization technique is to execute those test cases during online testing first which cover concrete paths

Table 7.3: Concrete Path Probabilities in Figure 7.1

Execution Plan	Concrete Path	Probability
epl1	<i>cp1</i> : (A:a1, B:b, D:d, G:g)	36%
	<i>cp2</i> : (A:a1, B:b, E:e, G:g)	12%
	<i>cp3</i> : (A:a1, C:c1, F:f, G:g)	12%
epl2	<i>cp4</i> : (A:a2, B:b, D:d, G:g)	18%
	<i>cp5</i> : (A:a2, B:b, E:e, G:g)	6%
	<i>cp6</i> : (A:a2, C:c2, F:f, G:g)	6%
epl3	<i>cp7</i> : (A:a3, B:b, D:d, G:g)	6%
	<i>cp8</i> : (A:a3, B:b, E:e, G:g)	2%
	<i>cp9</i> : (A:a3, C:c2, F:f, G:g)	2%

with least probable runtime monitoring coverage. We propose the following test case prioritization technique to achieve this goal.

U: Ascending Order of Concrete Path Probability

This technique sorts test cases in ascending order of the probabilities of the concrete paths they cover, and pseudorandomly in cases of ties. That is, the test cases which cover concrete paths with the lowest probability of their coverage during system usage (runtime monitoring) will be assigned the highest execution priority.

7.3.4 Hybrid Test Case Prioritization

Test case prioritization techniques using more than one piece of information may perform better than the techniques using only one piece of information [81]. The previous sections presented isolated prioritization techniques for each of the information presented in Section 7.2. In this section, we propose hybrid prioritization techniques which use all pieces of information together.

H1: Descending Achievable Coverage, Descending Execution Time, Ascending Concrete Path Probability

This technique uses achievable test coverage¹, execution time, and concrete

¹Achievable test coverage according to the coverage criteria presented in Chapter 6

path probability of test cases. The technique sorts the test cases in 3 steps ordered by certainty of the information available from achievable test coverage, execution time, and concrete path probability. In particular, the technique uses achievable test coverage first since the achievable test coverage reflects a *guaranteed coverage* of a service composition, while execution time and concrete path probability reflect *potential coverage* of the service composition. The technique uses the execution time before the concrete path probability because the execution time is a *measurement* (see Section 7.3.2) and the concrete path probability is an *estimation* (see Section 7.3.3).

Therefore, the technique initially sorts test cases in a descending order of their achievable test coverage. Then, the technique uses the execution time of test cases to break the tie when test cases have equal achievable test coverage. The technique sorts test cases in descending order of their execution times. Lastly, to break the tie when test cases have both equal achievable coverage and execution time, the technique sorts the test cases in ascending order of their path probability.

H2: Descending Test Case Potential

Another way for prioritizing test cases using all available information is to combine the information into a single model and to use the result for prioritization. The model we propose combines the achievable test coverage, execution time, and the concrete path probability into a single value which we call the *test case potential*.

The test case potential is computed as follows:

$$tc_p = \frac{C \times E}{P}, \text{ where}$$

- C is the achievable test coverage of the test case,
- E is the ratio of the test case execution time to the test suite execution time, and
- P is the path probability of the test case.

Since higher achievable test coverage and longer execution time might lead to faster coverage rate, tc_p is proportional to C and to E . Likewise, as low values of concrete path probability of a test case might lead to a faster coverage rate, tc_p is inversely proportional to P .

Based on the test case potential, the test case prioritization technique sorts test cases in descending order of their resulting tc_p values.

Despite the fact that both technique *H1* and technique *H2* use the same information (i.e., achievable coverage, execution time, and concrete path probability) for test case prioritization, technique *H1* and technique *H2* differ in how they use the information. Rather than using the information directly for prioritization as performed by technique *H1*, technique *H2* turns the information into a single value and uses the result for prioritization.

7.4 Summary

This chapter presented our techniques for online test case selection and prioritization for service compositions. The goal of the test case prioritization is to achieve coverage of service composition at a faster rate, using the coverage criteria presented in Chapter 6. To this end, the techniques use the following relevant pieces of information to prioritize test cases of a service composition: (1) the achievable coverage of the test cases; (2) the execution time of the test cases; (3) the usage model of the service composition.

Chapter 8

Online Testing and Monitoring Framework

This chapter presents our online testing and runtime monitoring framework called PROSA. PROSA introduced the idea of exploiting synergies between runtime monitoring and online testing, thereby achieving better coverage of service compositions [99]. PROSA was designed for Quality of Service monitoring and testing of constituent services of service compositions. In this Chapter, we adapt PROSA to support functional online testing and runtime monitoring of service compositions.

8.1 The PROSA Framework

PROSA constitutes three main modules as depicted in Figure 8.1¹:

- The **Runtime Monitoring Module** passively collects execution data of service compositions and their constituent services. The execution data include performance, usage, and coverage data.
- The **Online Testing Module** actively collects execution data of service compositions and their constituent services by initiating the execution of online tests.

¹For the notation used to model the architecture; see: <http://www.fmc-modeling.org/download/fmc-and-tam/SAP-TAM%5FStandard.pdf>

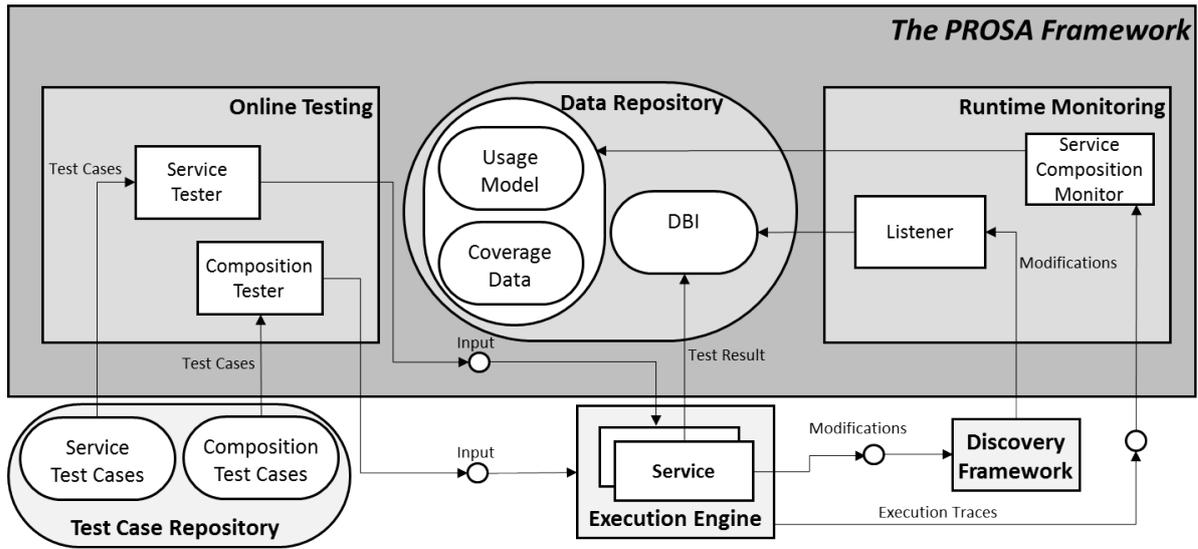


Figure 8.1: The PROSA Framework

- The **Data Repository Module** manages the storage and the retrieval of collected execution data.

In the following, we present each module in detail.

8.2 Runtime Monitoring Module

This module is responsible for the runtime monitoring activities of the PROSA framework. The Runtime Monitoring Module has two components: the Service Composition Monitor and the Listener.

8.2.1 Service Composition Monitor

This component is responsible for collecting and storing execution traces for a service composition. As defined in Section 5.1.1, an execution trace contains input, output, and traversed entities of the service composition.

The Service Composition Monitor can collect execution traces from different sources. One source could be the probes instrumented in the service composition [45, 9]. The service composition is instrumented to record the relevant entities (e.g., operations or branches) traversed by each execution. It should be

noted that, a heavy use of probes – as might be needed for collecting coverage at a detailed level – may degrade the performance of the service composition.

Another source are the events generated by the execution engine while executing a service composition [45]. Some service composition execution engines (e.g., ActiveBPEL² and Apache ODE³) emit events whenever a change in the service composition's execution state occurs. Execution traces can thus be obtained by analysing the events generated for those changes. Collecting execution traces using this approach is less invasive as these events are emitted by the execution engine anyway.

Execution traces may be obtained using different execution plans for the service composition. Hence, execution traces need to be associated with execution plans. Execution traces are therefore stored with identifiers for the used execution plans (one or more plan) or null to indicate an arbitrary use of the service composition.

The Service Composition Monitor stores the collected execution traces in the Data Repository Module.

8.2.2 Listener

The Listener component of the Runtime Monitoring Module collects information regarding modifications in the operations of the services bound in the service compositions. Observing modifications in the operations of bound services is required to determine valid execution traces (see Chapter 5).

The Listener can collect information about such modifications from the following sources:

- *Service Registry* [86]. A service registry refers to an infrastructure that supports service providers to publish descriptions about their offered services and potential users to search for services. Service registries regularly monitor the registered services to detect potential changes in their behaviour or Quality of Service. In case some of the constituent services are published in a Service Registry, the Listener subscribes to the registry to receive notifications about relevant changes of services pushed by the Service Registry.

²<http://www.activebpel.org/>

³<http://ode.apache.org/>

- *Service Discovery Framework [70]*. Service discovery frameworks are used by service integrators for the identification of services that can substitute constituent services in service compositions. Service discovery frameworks typically include listeners which query the used service registries about possible changes of registered services. Additionally, some runtime service discovery frameworks [70] may employ runtime monitors which intercept the SOAP messages exchanged between the service composition and its constituent services for identifying changes in their behaviour. Such runtime monitors can also be used to identify changes in the constituent services.

In either case, the Listener updates the Dynamic Binding Information (DBI) in the Data Repository to reflect observed modifications of services and service bindings.

8.3 Online Testing Module

This module is responsible for testing service compositions and their constituent services. Test execution is performed online i.e., in parallel to the normal operation of the service composition. As advocated by our approach, online testing of service compositions is used to complement runtime monitoring. The data collected by the Online Testing Module is similar to the data collected by the Runtime Monitoring Module.

For performing online testing activities, the Online Testing Module has two components, the Service Composition Tester for testing service compositions, and the Service Tester for testing constituent services.

8.3.1 Service Composition Tester

For coverage assessment, our approach uses (online) testing execution traces in combination with runtime monitoring traces. In this view, online testing can be triggered in cases when coverage of the service composition falls below a predefined threshold. The assessment of coverage can be performed using appropriate coverage criteria for service compositions. In Chapter 6, we proposed coverage criteria for service compositions.

For testing a service composition, test cases need to be selected. In this version of PROSA, we assume the test cases are available and stored in a Test Case Repository. The selection of test cases depends on the coverage criteria used for the service composition. The following are techniques for the coverage criteria presented in Chapter 6:

- *The Intra-plan-local Coverage Criterion (see Section 6.1.2):* The test case used for covering the target abstract service using the target execution plan will be selected in case the execution trace of the test case is invalid.
- *The Inter-plan-local Coverage (see Section 6.1.2):* The test cases used to cover the abstract service using all execution plans will be selected in case the execution traces of these test cases are invalid.
- *The Intra-plan-global Coverage Criterion (see Section 6.1.3):* The test cases used to cover the service composition using the target execution plan will be selected in case the execution traces of these test cases are invalid.
- *The Inter-plan-global Coverage Criterion (see Section 6.1.3):* The test cases used to cover the service composition using all execution plans will be selected in case the execution traces of the test cases are invalid.

For executing the selected test cases on the service composition, the Service Composition Tester sends the input defined in the test cases to the service composition execution engine, which then executes the service composition. The results of online testing (i.e., execution traces) are collected by the Service Composition Monitor in the same manner of collecting execution traces from normal usage of the service composition.

Using the Service Composition Monitor to collect both types of traces (i.e., test and monitoring) requires distinguishing the collected traces as either coming from monitoring normal system usage or from online testing. Technically, this can be achieved by attaching a tag when sending a request for testing purposes. The tag identifying the source of request is stored along with the execution trace. The default value for the tag will be identifying a monitoring

request. In case of testing, the value will be set for identifying a testing request as such.

For test case selection, it is necessary to know which execution trace belongs to which test case. As we defined execution traces, each execution trace includes input. Using the combination of input and testing activity identifier (i.e., the tag), we can identify test cases by looking for execution traces of particular input and identified as testing.

8.3.2 Service Tester

In addition to collecting execution traces for the entire service composition, the Online Testing Module is responsible for actively collecting observations about the constituent services. This activity is performed by the Service Tester component which complements the activities performed by the Listener. Therefore, the Service Tester invokes the operations of the services to identify potential modifications in the behaviour or the performance of used services. To this end, the Service Tester invokes the services using the input defined in the test cases stored in the Test Case Repository. The Service Tester can be configured to invoke services at predefined rates. For example, it can be configured to invoke a service whenever the usage rate of the service drops below a certain threshold.

The test execution can be implemented using existing test execution environments. In earlier versions of PROSA [99], we have used an existing services monitoring framework, SALMon, for performing online testing of constituent services. SALMon is designed following SOA principles, which allows easier integration in other frameworks and allows deploying SALMon functionality as third-party services. Moreover, SALMon combines both testing and monitoring in a single framework.

SALMon provides the Monitor Service (offered as a Web service) as the central access point for: (1) configuring monitoring and online testing for the different services in a service composition, and (2) retrieving Monitoring Data.

The Monitor Service has a testing component which invokes the services using the provided test input and test rate. To collect Monitoring Data from monitoring or online testing, the Monitor Service creates one or more measure instruments which implement the logic required to compute concrete quality

metrics of the service.

Once the monitor is configured, it dynamically activates a concrete set of Measure Instruments depending on the required quality metrics to measure. Monitoring is performed through an Enterprise Service Bus (ESB). That is, instead of directly invoking the services, all requests and responses are sent through the ESB which in turn feeds the Measure Instruments.

The results of testing are observed and compared with expected results as defined in the test cases. The Dynamic Binding Information in the Data Repository is updated to reflect the observed modifications.

8.4 Data Repository Module

This module stores the execution plans of the service composition. Execution plans can be thought of as configuration files (e.g., XML files) storing for each abstract service in a service composition, the operation of the bound service. These documents can be accessed when necessary, e.g., during service composition deployment or at runtime.

Additionally, the Data Repository Module is responsible for managing the data collected by the Online Testing and Runtime Monitoring Modules, such as data storage and retrieval. In particular, the repository stores the collected execution traces of service compositions. Execution traces are stored according to the format defined in Section 5.1.1, along with identifiers for the execution plans and the source of invocation (i.e., monitoring or testing). The Dynamic Binding Information stores the observed modifications of the used services.

Two types of data about the service composition and its constituent services can be derived from collected execution traces: *coverage data* and *usage data* of the service composition. While the coverage data is concerned with a single traversal of the entities, the usage data is concerned with their entire occurrence. The Coverage Data and the Usage Model are used by our proposed coverage criteria (see Chapter 6) and online test case selection and prioritization (see Chapter 7).

8.4.1 Coverage Data

Coverage data indicates which parts of the service composition (e.g., operations or branches) have been executed by which execution trace. Coverage data is stored in the Coverage Data repository.

In software testing, coverage data is compiled into a coverage matrix that associates each test execution with the parts of the software that the execution traverses [93]. The coverage matrix has two dimensions. One dimension is for the test cases. The other dimension is for the relevant parts of the test object. For instance, for edge-coverage data, the coverage matrix will associate test executions with the relevant edges in the test object.

We create the same representation for coverage data of service compositions. That is, we create coverage matrix associating each execution trace (of both testing and runtime monitoring) with the entities of the service composition contained by the trace. Since a service composition can be realized using several execution plans, we create an instance of the coverage matrix for each used execution plan. Each instance of the coverage matrix will associate the entities of the service composition using the respective execution plan, with the execution traces of the execution plan. Additionally, we create an instance of the coverage matrix for the coverage data of execution traces not based on any execution plan.

Using this representation for coverage data and the representation of execution traces, several information required by the thesis contributions can be derived. For example, we can determine which are the entities of the used execution plans and which are the entities traversed by existing execution traces. We can also determine which of the execution plan entities are covered by which execution trace.

8.4.2 Usage Model

Usage data indicates the usage patterns of the parts of the service composition. Usage data is recorded using the Usage Model. The Usage Model is used by the thesis contribution of online test case selection and prioritization (see Chapter 7). The Data Repository is responsible for maintaining the usage models for the deployed service compositions.

As discussed in Chapter 7, we use the Markovian representation for service compositions usage model. Key to maintaining the usage model of a service composition is computing the transition probability for all transitions in the model. It is important to note that the Usage Model is only updated by using execution traces obtained from actual service invocations and not by using the invocations which are triggered by online testing. If invocations for online testing would be considered, the data in the Usage Model would not reflect the usage of the system.

8.4.3 Dynamic Binding Information (DBI)

The Dynamic Binding Information (or DBI for short) is also stored in the Data Repository.

As discussed in Section 5.2, the DBI is a table for bookkeeping the modifications with respect to the bound operations of each abstract service in the service composition. For each candidate service binding, DBI stores the operations used in the service composition and whether the operations were modified. Any new operations used in the service composition are added in the DBI.

8.5 Summary

In this Chapter we presented the PROSA framework. The PROSA framework constitutes (1) The Runtime Monitoring Module; (2) The Online Testing Module; (3) The Data Repository Module. The Runtime Monitoring Module, and its components, are responsible for the passive collection of execution data of service compositions and their constituent services. The Online Testing Module, and its components, are responsible for the active collection of execution data of service compositions and their constituent services. The Data Repository Module is responsible for managing the collected data, including storage and retrieval. The modules of the PROSA framework provide the technical support for the thesis contributions introduced earlier.

Chapter 9

Evaluation

In this chapter, we evaluate the first three contributions of the thesis. The PROSA framework – the last contribution which focuses on the technical aspects of the contributions evaluated in this Chapter – has been evaluated in another context, and the results are published in [99, 79].

We use the **Goal Question Metric (GQM)** paradigm [67] for defining evaluation goals, refining goals into metrics, and interpreting the resulting data. We carry out the evaluation by means of controlled experiments. A controlled experiment provides us with a higher degree of control compared to other types of evaluation such as case study or questionnaire.

Thus for each evaluation, in addition to the goals, questions, and metrics, we present the experimental plan including objects, design, and execution, followed by analysis of the experimental results.

9.1 The Goal Question Metric Paradigm

The idea behind the GQM Paradigm is that measurement should be based on goals [67]. Goals provide rationale for the data collection and interpretation activities carried out in the evaluation.

The **Goal** states the particular aspects of a contribution that will be studied by the evaluation. A goal can be characterized with respect to the following aspects: *object*, *purpose*, *quality*, *viewpoint*, and *context* of the study [67], and can be constructed by means of a *goal template*, as follows:

Analyse *<Object(s) of study>*
 for the purpose of *<Purpose>*
 with respect to their *<Quality focus>*
 from the point of view of the *<Perspective>*
 in the context of *<Context>*.

The *object* is the studied entity in the experiment, e.g., model, metric, theory, etc. The *purpose* defines what the experiment is intended to do, e.g., to evaluate, to characterize, etc. The *quality focus* is the effect under study, e.g., efficiency, effectiveness, cost, etc. The *perspective* defines from which viewpoint the results are interpreted, e.g., developer, researcher, etc. Finally, the *context* is the environment in which the experiment is conducted. Context defines the involved personnel (subjects) and the used software artefacts (objects) in the experiment.

The **Question** formulates the questions required to be answered for meeting the goal(s). Finally, the **Metric** defines the measures, and thus the data to be collected, for answering the questions.

9.2 Evaluation of Determining Valid Execution Traces

9.2.1 Goals, Questions, and Metrics

For the evaluation of determining valid execution traces we define goal 1.

Goal 1. Analyse the *extended graph-walk algorithm*
 for the purpose of *evaluation*
 with respect to its *execution time*
 from the point of view of the *researcher*
 in the context of *self-adaptive service compositions*.

Goal 1 is concerned with performance evaluation of our graph-walk algorithm in terms of the algorithm's execution time. The asymptotic analysis performed in Section 5.2.3 indicated the critical parameters by which the runtime of the algorithm can be bounded. The goal is to support the analysis with

empirical results for the runtime of the algorithm, while considering the impact of the bounding parameters. While the result of the asymptotic analysis allows estimating the runtime of the algorithm for any value of the parameters, the evaluation will provide results for concrete values of the parameters, selected based on values used in the service-oriented computing literature. The bounding parameters are:

1. *Number of nodes*: this parameter indicates the number of nodes in the control-flow graph of a service composition. The asymptotic analysis of the algorithm indicated a linear relationship between the execution time of the algorithm and the number of control-flow graph nodes.
2. *Number of bindings*: this parameter indicates the number of candidate service bindings per abstract service in a service composition. The asymptotic analysis of the algorithm indicated a linear relationship between the execution time and the number of bindings.
3. *Number of traces*: this parameter indicates the number of execution traces considered by the algorithm when searching for invalid traces. The asymptotic analysis of the algorithm indicated a linear relationship between the execution time of the algorithm and the number of execution traces.

Therefore, meeting goal 1 will require us to answer the following questions.

Q1.1: What effect can the number of nodes of the control-flow graph have on the execution time of the algorithm?

Q1.2: What effect can the number of bindings in the control-flow graph have on the execution time of the algorithm?

Q1.3: What effect can the number of execution traces of the service composition have on the execution time of the algorithm?

These questions will be answered by collecting data for the following metric:

M1.1: The amount of time the algorithm required to determine invalid traces.

9.2.2 Experimental Plan

Experimental Objects

As objects for our experiments, we use realistic service compositions frequently used in the service-oriented computing research topics related to our contributions such as adaptive service composition and service testing. Such example service compositions are used in the service-oriented computing literature [75, 76, 77, 104, 106, 52, 23, 20, 23] to compensate for the lack of open source service compositions to be used for evaluation purposes.

Table 9.1: Characteristics of the service compositions used in the experiments

Service Composition			Control-flow Graph		
Application	Domain	#services	#nodes	#trans.	#branches
<i>Loan Approval</i>	Banking	2	9	13	4
<i>DSL Service</i>	E-commerce	3	10	11	2
<i>Supply Chain</i>	E-commerce	6	8	7	0
<i>Trip Planning</i>	Tourism	6	10	10	2
<i>Image Processing</i>	Computing	6	12	13	4

In particular, we use: a *Loan Approval* service composition [75, 76, 77, 106], a *DSL Service* service composition [75, 76, 77], a *Supply Chain* service composition [104, 106], a *Trip Planning* service composition [52, 23], and an *Image Processing* service composition [20, 23]. Table 9.1 summarizes the characteristics of each service composition.

Design and Execution

To meet goal 1, we design our experiment as follows. The independent variables are the parameters on which the execution time of the algorithm can strongly depend. The dependent variable is the execution time of the algorithm which we would like to measure. We vary the values for the independent variables as follows:

1. *Number of nodes*: we consider the control-flow graphs of the service compositions in Table 9.1, thereby covering a various number of control-flow graph nodes and abstract services.
2. *Number of bindings*: We vary the number of bindings from 10 to 50 with steps of 10, covering the values which have been considered by several other studies [69, 23, 65, 55, 59, 108, 22].
3. *Number of traces*: We vary the values for this parameter from 100 to 1000 with steps of 100, and from 1000 to 5000 with steps of 500. Despite the small size of the service compositions used in the experiments, considering a large number of service bindings allows having a large number of unique execution traces.

In the performed experiments, we combine the independent variables as follows. We vary the values of the independent variable of interest and hold all the other parameters fixed at certain values. The values at which the other independent variables are fixed do not effect the analysis of the results as we use the same values when evaluating the parameters. In particular, we vary the values of the *number of bindings* while fixing the *number of traces* to 1000 for each of the service compositions. We vary the values of the *number of traces* while fixing the *number of bindings* to 50 for each of the service compositions. We consider all the service compositions in Table 9.1, which have various *number of nodes* (8,9,10,12).

Our experiments cover the whole set of combinations resulting from the aforementioned design. Each experiment run involves the following steps. Initially, we generate a set of execution traces by simulating various executions of the studied service composition, randomly choosing one path over the control-flow graph of the service composition. For evaluating adequate monitoring (see Chapter 3 for details), Bertolino et al. [16] similarly simulated execution traces randomly choosing one path over the behavioural model of the service compositions. Then, we generate a modified version of the service composition simulating that a number service bindings is modified. We achieve this by setting in the DBI (see Chapter 5 and Chapter 8) the status of the service binding to *modified*. Finally, using the control-flow graphs of the original and modified versions of the service composition, we run our algorithm to determine the

invalid execution traces from the set of execution traces which we generated and record the execution time.

We have conducted the experiments on a machine with a 2.93 GHz Intel Core i7 processor, 8 GB 1333 MHz DDR3 memory, the OS X Yosemite operating system, and Java JRE 1.6. For obtaining the CPU time, we have used the package `java.lang.management`¹. This package provides the management interface for monitoring and management of the Java virtual machine as well as the operating system on which the Java virtual machine is running. Following other studies [48, 112, 61, 23, 106], we repeat each run 100 to avoid bias introduced by randomness, e.g., from simulating execution traces and simulating modifications in service compositions.

9.2.3 Results

The results for goal 1 are obtained from applying M1.1 and are summarized in two main figures. Figure 9.1 shows the effect of the *number of bindings* on the execution time. Figure 9.2 shows the effect of the *number of traces* on the execution time. Thus, the x-axis of the figures includes the values for the independent variable and the y-axis includes the average execution time in ms (for 100 repetitions). The results are differentiated for all the service compositions in Table 9.1.

The Effect of Number of Bindings on Execution Time

Figure 9.1 shows that – for all the considered service compositions – the average execution time of the algorithm is affected by the *number of bindings*, when holding the *number of traces* fixed. The execution time increases as the number of service bindings used by the service composition increases. The relationship between the execution time and the number of bindings appears to be linear for the design considered in our experiments. For the *Image Processing*, *Trip Planning*, and *Supply Chain*, which have the largest number of nodes, the execution time is the longest along all the values of the *number of bindings*.

The Effect of Number of Traces on Execution Time

¹<http://docs.oracle.com/javase/6/docs/api/java/lang/management/package-summary.html>

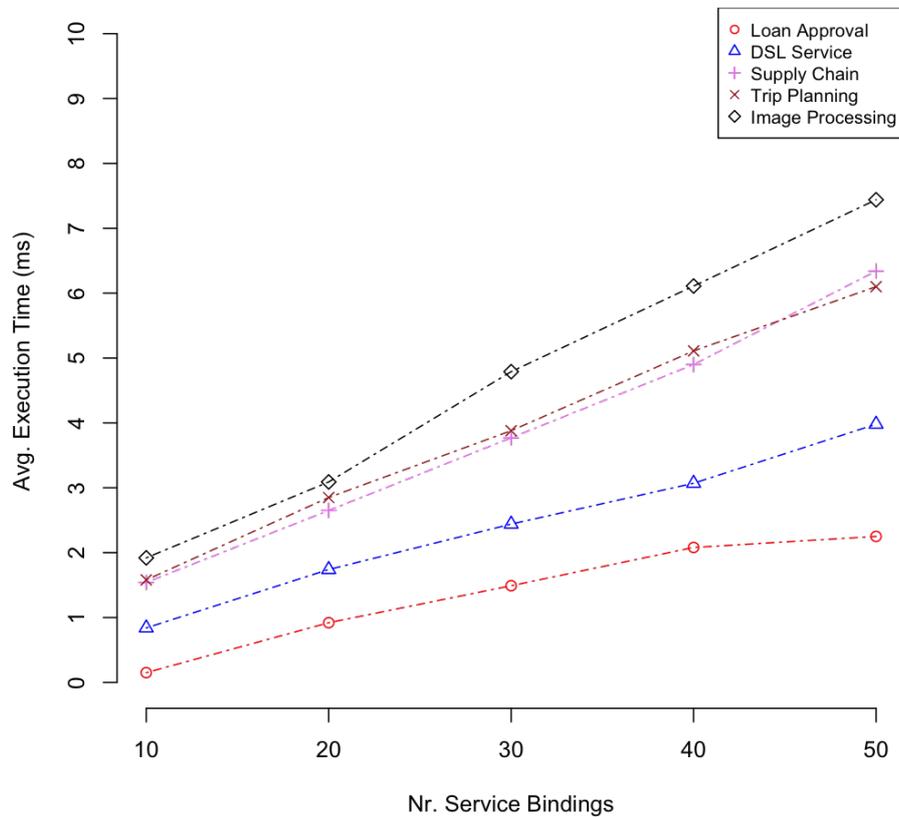


Figure 9.1: The average execution time of the algorithm vs. the *number of bindings* of the service compositions (Q1.2)

Similar conclusions can be obtained from Figure 9.2 regarding the effect of the *number of traces* on the execution time. For all the considered service compositions, the average execution time of the algorithm is affected by the *number of execution traces*, when holding the *number of bindings* fixed. The execution time increases as the number of execution traces considered by the algorithm increases. The relationship between the execution time and the number of execution traces appears to be linear for the design considered in our experiments. The execution time is longer for service compositions with larger number of nodes.

The Effect of Number of Nodes on Execution Time

Despite the fact the number of nodes in the studied service composition has

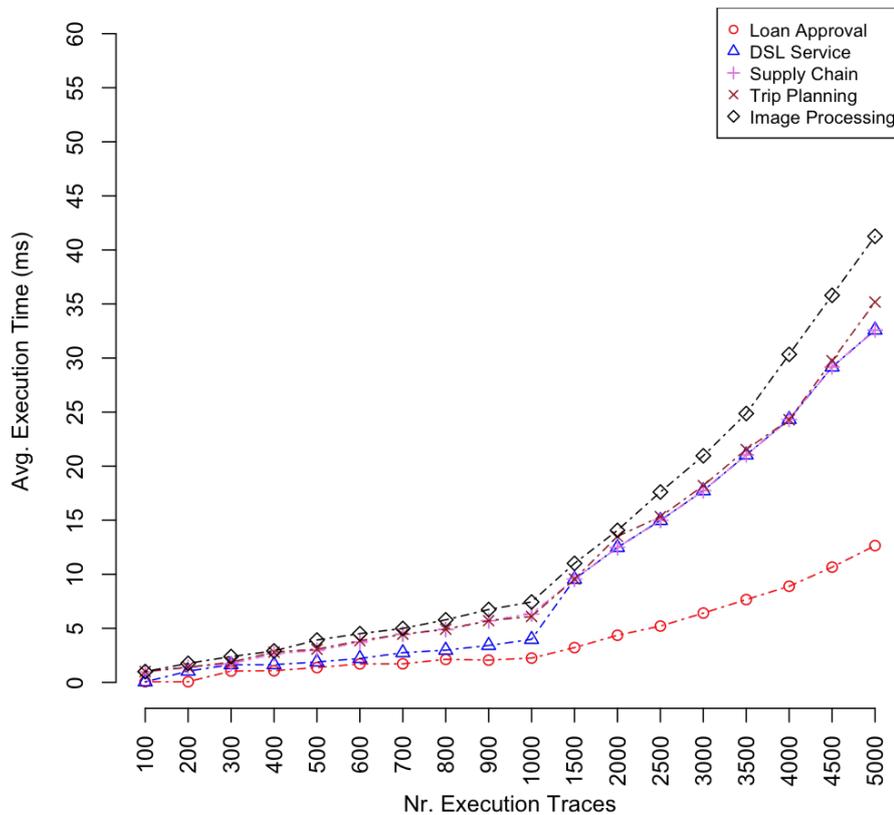


Figure 9.2: The average execution time of the algorithm vs. the *number of traces* of the service compositions (Q1.3)

not largely varied (from 9 – 12), both Figure 9.1 and Figure 9.2 suggest that the execution time is affected by the number of nodes in the service composition. The higher the number of nodes (as for Supply Chain, Trip Planning, and Image Processing), the longer is the execution time of the algorithm.

Conclusion

For the considered cases in this experiment, the results show the bounding parameters identified by asymptotic analysis effect the execution time of the algorithm. Additionally, the results suggest that the execution time of our algorithm is not high (1 – 40 ms). The algorithm can be used at runtime to support timely decision making.

9.3 Evaluation of Coverage Criteria

9.3.1 Goals, Questions, and Metrics

For the evaluation of coverage criteria we define goal 2 and goal 3.

Goal 2. Analyse the *global coverage criteria* for the purpose of *evaluation* with respect to *avoiding unnecessary operation (resp. branch) coverage* from the point of view of the *researcher* in the context of *runtime coverage assessment, self-adaptive service compositions*.

Goal 2 is concerned with the unnecessary operation (resp. branch) coverage assessment. In coverage assessment for a service composition, unnecessary operation (resp. branch) coverage can be computed. Unnecessary coverage can be computed when considering candidate service bindings not used in the execution plans defined for the service composition. Our coverage criteria focus rather on the coverage of operations (resp. branches) used in the execution plans defined for the service composition.

Meeting goal 2 will require us to answer the following questions.

Q2.1: To what extent does the global operation coverage of a service composition differ when: (1) *considering* the execution plans defined for the service composition and (2) *not considering* the execution plans defined for the service composition?

Q2.2: To what extent does the global branch coverage of a service composition differ when: (1) *considering* the execution plans defined for the service composition and (2) *not considering* the execution plans defined for the service composition?

These questions will be answered by collecting data for the following metrics.

M2.1: The inter-plan-global operation coverage of a service composition considering the execution plans defined for the service composition. It is measured using algorithm 7 where in this case *entity = operation*.

M2.2: The inter-plan-global operation coverage of a service composition considering all candidate service bindings of the service composition. It is measured using algorithm 7 where in this case *entity = operation* and the set of

execution plans EP is the set of all possible combinations of candidate service bindings of the service composition.

M2.3: The arithmetic difference between M2.1 and M2.2 (i.e., the result of the arithmetic subtraction $M2.1 - M2.2$).

M2.4: The inter-plan-global branch coverage of a service composition considering the execution plans defined for the service composition. It is measured using algorithm 7 where in this case $entity = branch$.

M2.5: The inter-plan-global branch coverage of a service composition considering all candidate service bindings of the service composition. It is measured using algorithm 7 where in this case $entity = branch$ and the set of execution plans EP is the set of all possible combinations of candidate service bindings of the service composition.

M2.6: The arithmetic difference between M2.4 and M2.5 (i.e., the result of the arithmetic subtraction $M2.4 - M2.5$).

Goal 3 is similar to goal 2 but concerns our local coverage criteria rather than our global coverage criteria. Thus, we define goal 3 using the same template of goal 2 replacing *global* with *local*.

Goal 3. Analyse the *local coverage criteria* for the purpose of *evaluation* with respect to *avoiding unnecessary operation (resp. branch) coverage* from the point of view of the *researcher* in the context of *runtime coverage assessment, self-adaptive service compositions*.

Meeting goal 3 will require us to answer the following questions.

Q3.1: To what extent does the local operation coverage of a service composition differ when: (1) *considering* the execution plans defined for the service composition and (2) *not considering* the execution plans defined for the service composition?

Q3.2: To what extent does the local branch coverage of a service composition differ when: (1) *considering* the execution plans defined for the service composition and (2) *not considering* the execution plans defined for the service composition?

These questions will be answered by collecting data for the following metrics.

M3.1: The inter-plan-local operation coverage of a service composition considering the execution plans of the service composition. It is measured using algorithm 5 where in this case $entity = operation$.

M3.2: The inter-plan-local operation coverage of a service composition considering all candidate service bindings of the service composition. It is measured using algorithm 5 where in this case $entity = operation$ and the set of execution plans EP is the set of all possible combinations of candidate service bindings of the service composition.

M3.3: The arithmetic difference between M3.1 and M3.2 (i.e., the result of the arithmetic subtraction $M3.1 - M3.2$).

M3.4: The inter-plan-local branch coverage of a service composition considering the execution plans of the service composition. It is measured using algorithm 5 where in this case $entity = branch$.

M3.5: The inter-plan-local branch coverage of a service composition considering all candidate service bindings of the service composition. It is measured using algorithm 5 where in this case $entity = branch$ and the set of execution plans EP is the set of all possible combinations of candidate service bindings of the service composition.

M3.6: The arithmetic difference between M3.4 and M3.5 (i.e., the result of the arithmetic subtraction $M3.4 - M3.5$).

9.3.2 Experimental Plan

Experimental Objects

As objects for our experiments, we use the control-flow graphs of the service compositions in Table 9.1.

Design and Execution

Meeting goal 2 and goal 3 requires execution plans and execution traces for the service compositions to apply the metrics M2.1 – M2.6 and M3.1 – M3.6. The dependent variable in our experiment is the measured coverage. The independent variables are the number of execution plans and execution traces.

To obtain execution plans for a service composition, we proceed as follows. We assign candidate service bindings for each abstract service in the service

composition’s control-flow graph. Then, we generate all possible plans for the control-flow graph by considering all possible combinations of candidate service bindings. For each control-flow graph, we use a number of service bindings per abstract service from the values covered in Section 9.2.2², such that we obtain the same number of plans for all control-flow graphs. The resulting number of plans for all control-flow graphs is 4096. From these plans, we randomly select a set of plans to represent the execution plans of the control-flow graph. We vary the number of selected execution plans by considering percentages in the range 10% – 100% from all plans.

To obtain execution traces, we follow the same approach described in Section 9.2. That is, we simulate executions of a service composition, randomly selecting one complete path over the control-flow graph of the service composition. However, instead of randomly selecting a service binding for an abstract service, we randomly use an execution plan, from the selected execution plans, to define the service bindings for the abstract services in a control-flow graph. As different numbers of execution traces could achieve different coverage results, we vary the number of execution traces from 1000 to 5000 for all runs of the experiment.

Finally, we apply the metrics M2.1 – M2.6 and M3.1 – M3.6. In our experiments, randomness is used in simulating and selecting execution traces and execution plans. In our pre-tests, we observed that the results obtained for multiple repetitions of the experiment did not vary much because of this randomness. Therefore, we repeat each run of the experiment 5 times.

9.3.3 Results

Goal 2 Results

Concerning the global coverage criteria (goal 2), Figure 9.3 and Figure 9.4 show the results measured with metric M2.3 and metric M2.6. Each figure contains 5 plots (a-e) showing the measured results for the different numbers of execution traces considered in the experiments (i.e., 1000 – 5000). The *x*-axis represents the number of execution plans used. The *y*-axis represents

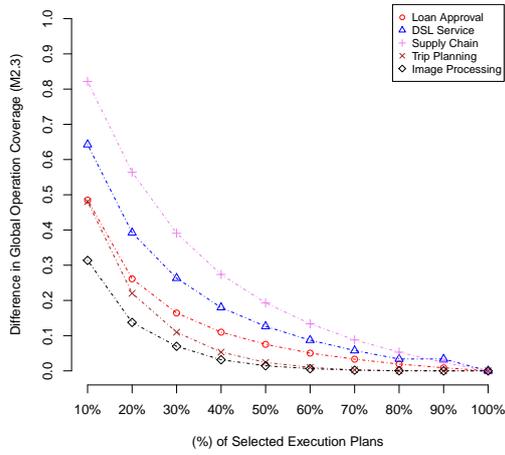
²Loan Approval has only 2 abstract services, therefore we use a larger number of service bindings (63)

the average of the measured results over 5 experiment repetitions (see Section 9.3.2). The shown results are differentiated for all the service compositions in Table 9.1.

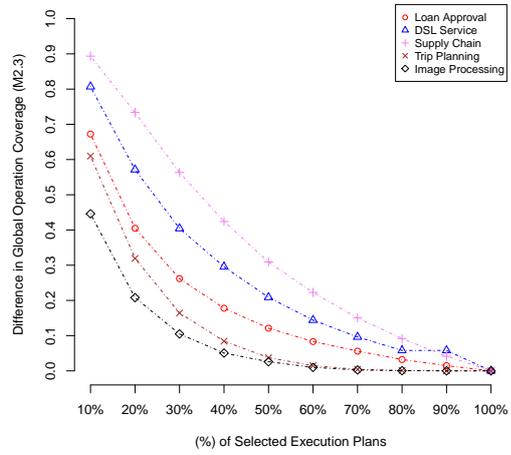
The results suggest that, a higher inter-plan-global coverage of service compositions' operations (resp. branches) is achieved when using execution plans compared to using all candidate service bindings. These differences are quantified using the results measured with metric M2.3 and metric M2.6. The rationale behind this is that, using execution plans, a smaller number of operations (resp. branches) is subject to coverage assessment compared to using all combinations of candidate service bindings.

As we can see from the figures, the results measured with metric M2.3 and metric M2.6 decrease as the number of execution plans increase. This means that, the inter-plan-global coverage of service compositions' operations (resp. branches) decreases as more execution plans are used. The rationale behind this observation is that, using more execution plans, more operations (resp. branches) become subject to coverage assessment. Therefore, the achieved coverage decrease. Moreover, using all candidate service bindings, the execution traces could cover more operations (resp. branches). Therefore, the achieved coverage increase. Consequently, the results measured with metric M2.3 and metric M2.6 decrease.

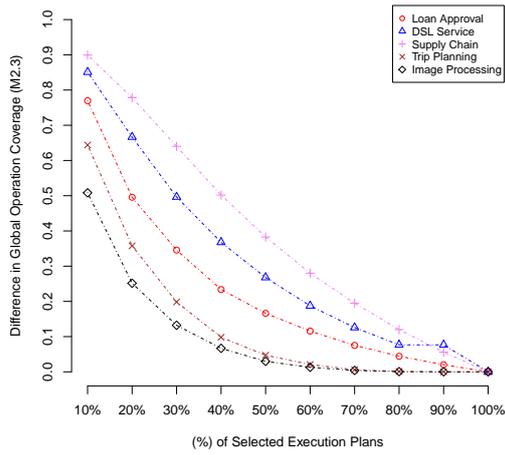
Concerning the effect of the number of execution traces on the measured coverage, the results measured with metric M2.3 and metric M2.6 increase as the number of execution traces increase. As one would expect, more execution traces could achieve higher inter-plan-global coverage of service composition's operations (resp. branches) when using execution plans and when using all candidate service bindings. As using execution plans a fewer operations (resp. branches) could be subject to coverage assessment compared to using candidate service bindings, the increase in coverage using more execution traces could be higher than using candidate service bindings. Therefore, the differences in the achieved inter-plan-global coverage, as quantified by metric M2.3 and metric M2.6, increase as more execution traces are used.



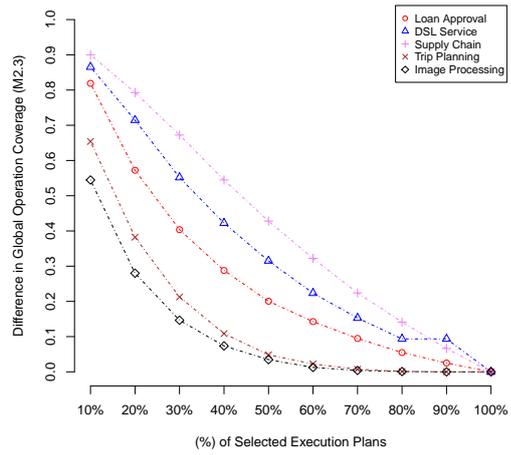
(a) 1000 Execution Traces



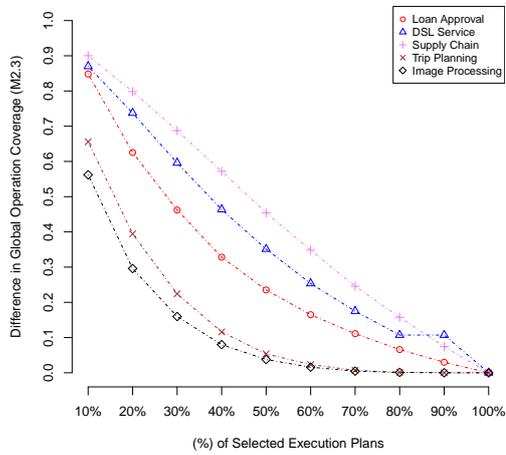
(b) 2000 Execution Traces



(c) 3000 Execution Traces

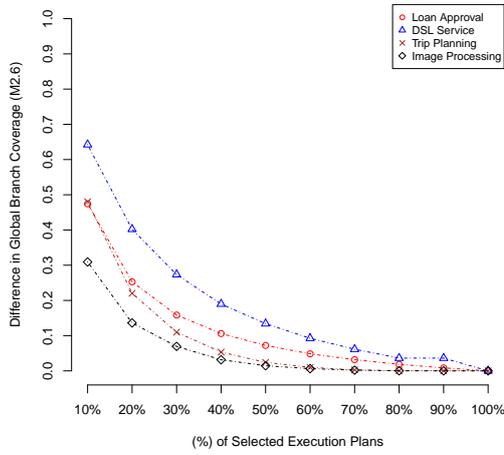


(d) 4000 Execution Traces

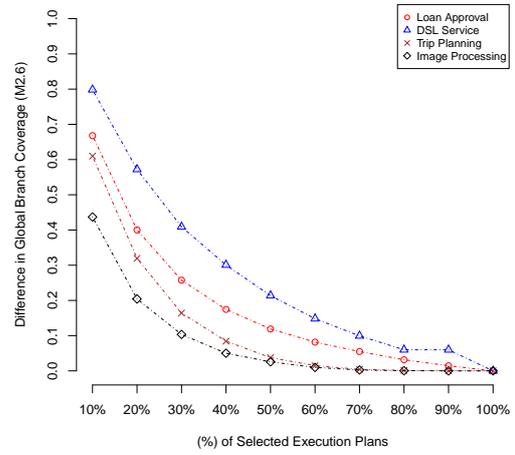


(e) 5000 Execution Traces

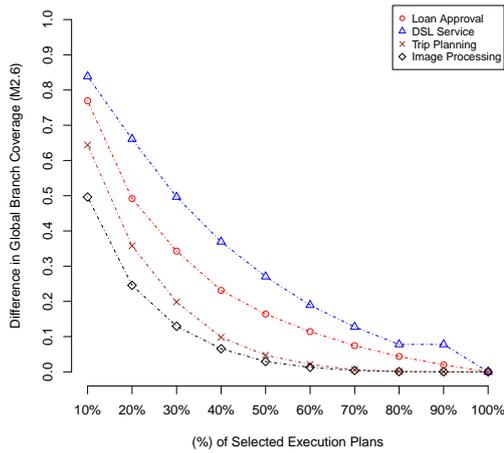
Figure 9.3: Results measured with metric M2.3.



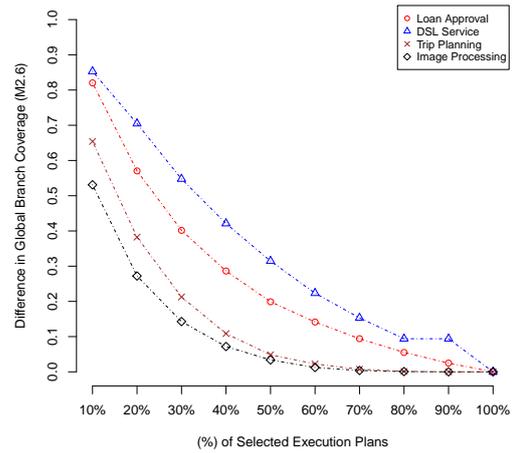
(a) 1000 Execution Traces



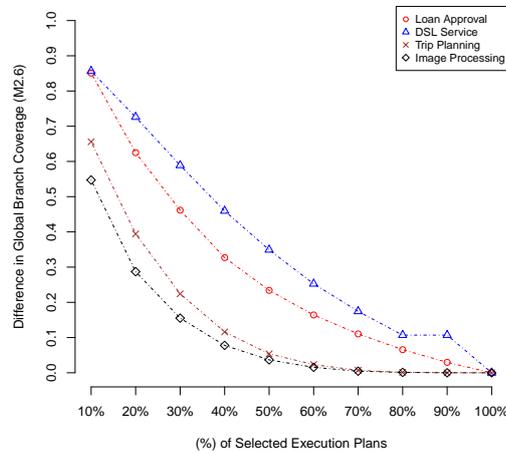
(b) 2000 Execution Traces



(c) 3000 Execution Traces



(d) 4000 Execution Traces



(e) 5000 Execution Traces

Figure 9.4: Results measured with metric M2.6. The service composition Supply Chain has no branches.

Goal 3 Results

Concerning the local coverage criteria (goal 3), Figure 9.5 and Figure 9.6 show the results measured with metric M3.3 and metric M3.6. The results measured with metric M3.3 and metric M3.6. for all abstract services in a service composition are very much similar. Therefore, we only show results for one abstract service of each service composition to keep the presentation compact. The same observations and conclusions apply for all abstract services of the service composition. The comprehensive results for all abstract services in the service compositions are available in Appendix A.

As we can see from Figure 9.5 and Figure 9.6, the same observations made for goal 2 also hold for goal 3. The results measured with M3.3 and M3.6 indicate that using execution plans higher inter-plan-local coverage of service composition's operations (resp. branches) could be achieved compared to using all candidate service bindings. The results measured with metric M3.3 and metric M3.6 decrease as more execution plans are used. Using more execution traces leads to higher results measured with metric M3.3 and metric M3.6.

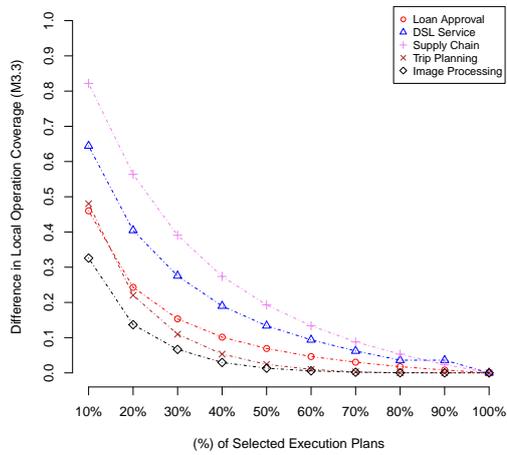
9.4 Evaluation of Online Test Case Selection and Prioritization

9.4.1 Goals, Questions, and Metrics

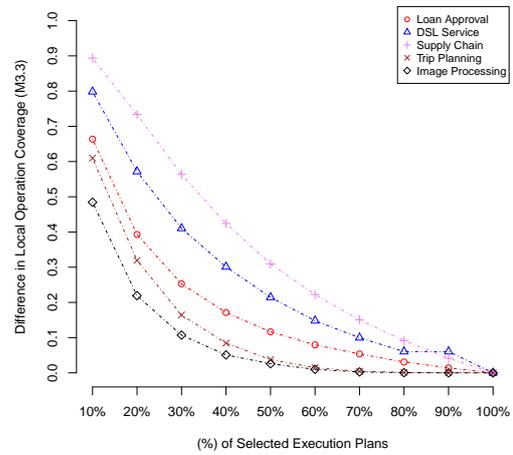
For evaluating the online test case selection and prioritization we define goal 4.

Goal 4. Analyse the *test case prioritization techniques* for the purpose of *understanding, comparison* with respect to their operation (resp. branch) *coverage rate* from the point of view of the *researcher* in the context of *online test case prioritization, self-adaptive service compositions*.

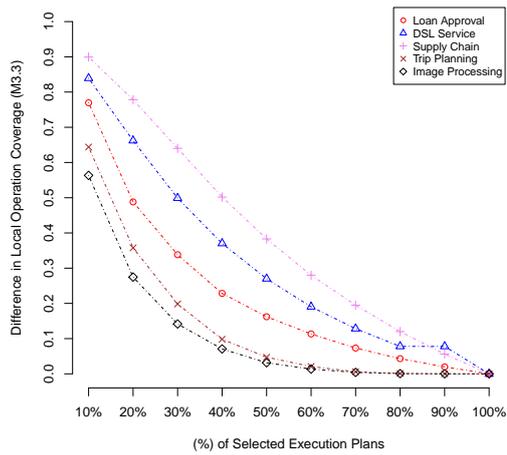
Goal 4 is concerned with the effectiveness of the proposed prioritization techniques in terms of operation (resp. branch) coverage rate. To this end, we



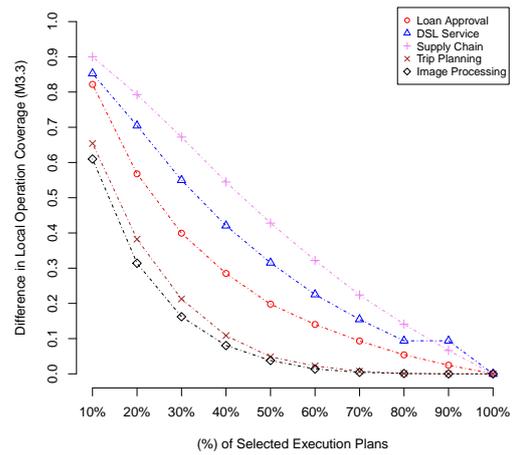
(a) 1000 Execution Traces



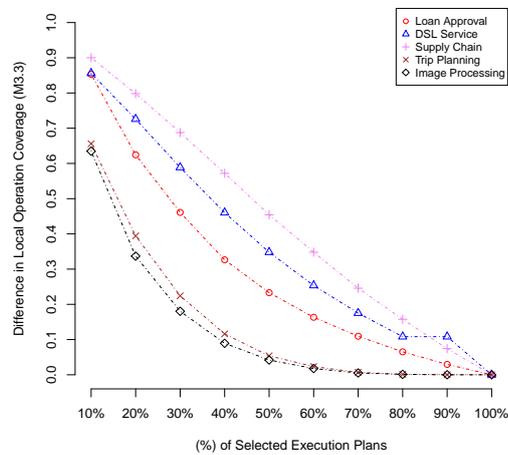
(b) 2000 Execution Traces



(c) 3000 Execution Traces

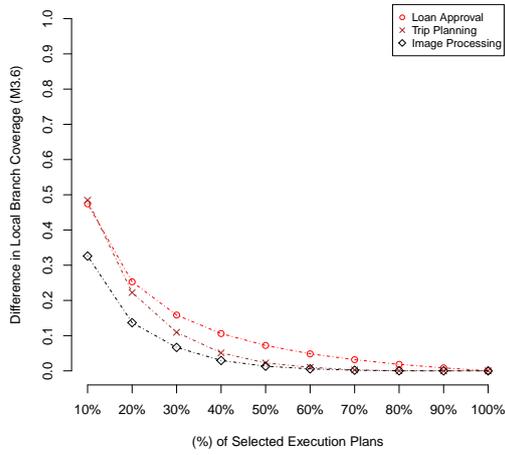


(d) 4000 Execution Traces

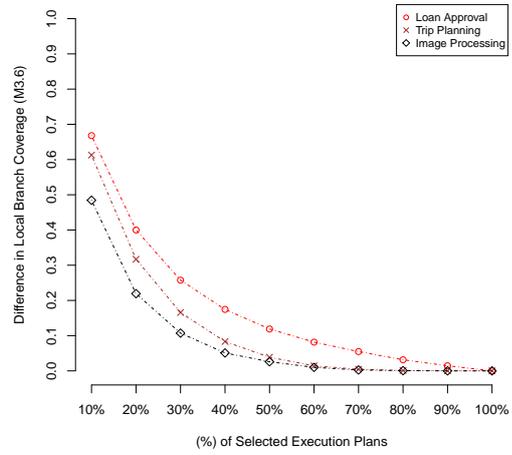


(e) 5000 Execution Traces

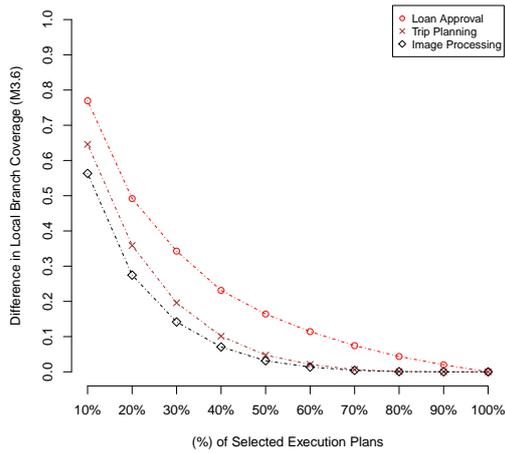
Figure 9.5: Results measured with metric M3.3.



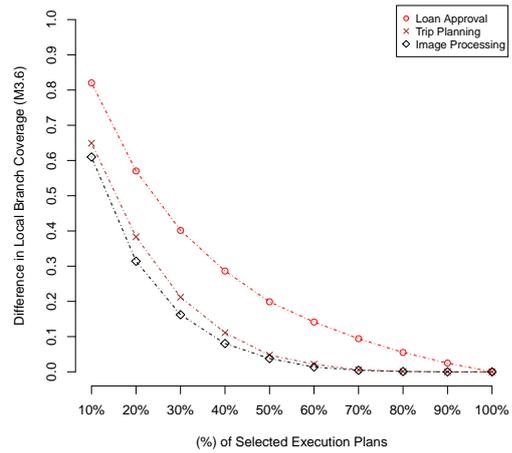
(a) 1000 Execution Traces



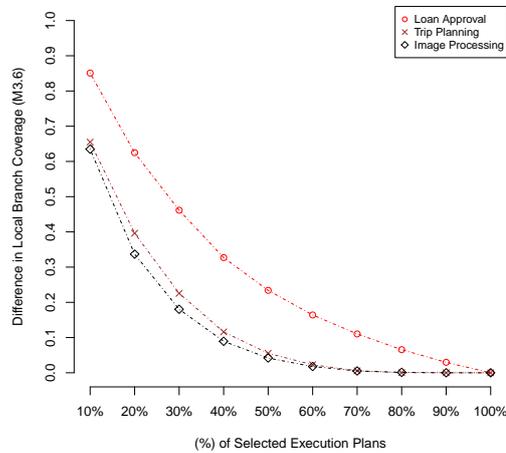
(b) 2000 Execution Traces



(c) 3000 Execution Traces



(d) 4000 Execution Traces



(e) 5000 Execution Traces

Figure 9.6: Results measured with metric M3.6. Supply Chain has no branches. DSL Service has no abstract service where local branch coverage is computed.

compare our prioritization techniques with: (1) random prioritization, which randomly orders a test suite; (2) no prioritization, which does not perform any ordering of the test suite. The difference between random prioritization and no prioritization is that the order of the test cases will remain the same in case of no prioritization whereas it can be different as a result of random ordering in case of random prioritization.

Furthermore, our prioritization techniques are divided into techniques that use coverage information, execution time information, and usage information. It is interesting to know whether one type of the information is more effective for prioritization than the others.

To study goal 4, we answer the following questions.

Q4.1 Which technique is most effective in terms of operation coverage rate for online test case prioritization?

Q4.2 Which technique is most effective in terms of branch coverage rate for online test case prioritization?

To answer these questions, we collect information for the following metrics.

M4.1 The Average Percentage Operation Coverage (APOC for short) which measures the rate at which a prioritized test suite covers operations. Let a test suite T containing n test cases that covers a set O of m operations. Let TO_i be the first test case in the order T' of T that covers operation i . The APOC for order T' is given by the equation

$$APOC = 1 - \frac{TO_1 + TO_2 + \dots + TO_n}{nm} + \frac{1}{2n},$$

where APOC values range from 0 to 100 and higher values imply faster (better) coverage rates.

M4.2 The Average Percentage Branch Coverage (APBC for short) which measures the rate at which a prioritized test suite covers branches. Let a test suite T containing n test cases that covers a set B of m branches. Let TB_i be the first test case in the order T' of T that covers branch i . The APBC for order T' is given by the equation

$$APBC = 1 - \frac{TB_1 + TB_2 + \dots + TB_n}{nm} + \frac{1}{2n},$$

where APBC values range from 0 to 100 and higher values imply faster (better) coverage rates.

The metric APOC and the metric APBC are based upon the metrics proposed by Zheng et al. [65] (see Section 7.1), which are variants of the famous metric APFD (Average of the Percentage of Faults Detected). The metric APFD measures the weighted average of the percentage of faults detected over the life of the test suite. Instead of using faults exposed by a test case, which cannot be estimated before testing has taken place, the metric APOC and metric APBD use coverage as a surrogate measure. The APOC and the APBD metrics are thus suitable for evaluating our test case prioritization techniques.

9.4.2 Experimental Plan

Experimental Objects

As objects for our experiments, we use the control-flow graphs of the service compositions in Table 9.1.

Design and Execution

Goal 4 requires test suites for the service compositions, and prioritizing test cases in the test suites using the prioritization techniques to be evaluated.

To obtain a test suite for a control-flow graph of a service composition, we generate test cases traversing random paths in the control-flow graph. Thus, each test case contains coverage information about the traversed path in the control-flow graph. In order to select the test cases of the test suite, we followed the approach used in an infrastructure designed to support controlled experimentation with software testing and regression testing [36]. The same approach is also used by [65, 106]. A test case is generated at random and is added to the suite only if the test case increases the cumulative coverage of the entire service composition³. We use both operation coverage and branch coverage. This is repeated until the test suite achieves full operation (resp. branch) coverage.

Using the generated test suite, we order the test cases using the coverage-based prioritization and random prioritization techniques. Then, we measure the coverage rate of each technique using the metrics M4.1 and M4.2.

³Another approach for generating large test suites is to add each randomly generated test case to the suite.

The time-based prioritization technique requires information about test case execution time. For a given test suite for a service composition, we compute the execution time for the test suite as a summation of the execution time of each test case in the test suite. As we generate a logical test case and not actual test case, we compute the execution time of the test case by summing up the execution times of each operation traversed by the test case. This requires us to have execution time information for each operation in the service composition. To this end, we employ publicly-available QWS2 dataset [1] which is frequently used by other researchers for the same purpose [103, 51, 4, 3].

The QWS2 dataset comprises measurements of 9 Quality of Service attributes (including response time) for 2,507 real-world Web services. These Web services were collected from public sources on the Web, including UDDI registries, search engines and service portals, and their Quality of Service values were measured using commercial benchmark tools. More details about QWS2 dataset can be found in [1]. As the dataset provides response times of Web services, it suits our experimental plan.

Thus, following [103], for each operation in the control-flow graph of a service composition, we randomly select a candidate Web service from the 2,507 Web services. The same service may be selected for several abstract services in the control-flow graph. We retrieve the response time of the selected Web service and use it as the execution time for the corresponding Web service.

The usage-based prioritization technique requires a usage model for the service composition in order to compute the concrete path probability of each test case in the test suite. To this end, we derive a usage model for the control-flow graph of service composition from 10000 execution traces which we simulate to act as runtime monitoring history. We simulate monitoring execution traces following the same approach described in Section 9.2.2.

For measuring the coverage rate of the monitoring-based prioritization techniques⁴ using M4.1 and M4.2, we need to consider the coverage obtained from service composition monitoring. To this end, we simulate the execution of the service composition over a period of time. As monitoring coverage is considered during the execution of the prioritized test cases, we set the length of

⁴The monitoring-based prioritization techniques are: time-based, usage-based, and hybrid.

the simulation period to the test suite's execution time. To simulate service composition execution, we generate 1000 execution traces following the same approach described in Section 9.2.2. Additionally, we divide the simulation period to a number of time points equal to the length of the simulation period. Then, we randomly distribute the monitoring execution traces over the simulation period. The frequency at which we distribute each monitoring execution trace is based on the usage model of the service composition.

Finally, we compute the metrics M4.1 and M4.2. In our simulations, randomness is used in simulating and selecting execution traces, candidate service bindings from the dataset, and execution period. In our pre-tests, we observed that the results obtained for multiple repetitions of the experiment did not vary much because of this randomness. Therefore, we repeat each experiment 5 times.

9.4.3 Results

Figure 9.7 and Figure 9.8 show results for Q4.1, and Figure 9.9 and Figure 9.10 show results for Q4.2, using box plots. Figure 9.7 and Figure 9.9 show the results for each service composition individually, while Figure 9.8 and Figure 9.10 show the results for all service compositions together.

The studied test case prioritization techniques are labelled as follows:

- *Original*: refers to the original test suite before prioritization (i.e., no prioritization).
- *Random*: refers to random prioritization which randomly orders the test cases in the test suite.
- *Coverage*: refers to the coverage-based prioritization which orders the test cases in a test suite according to the achieved inter-plan global coverage of the test cases, as described in Section 7.3.1.
- *Time*: refers to the time-based prioritization which orders the test cases in a test suite according to the execution time of the test cases, as described in Section 7.3.2.

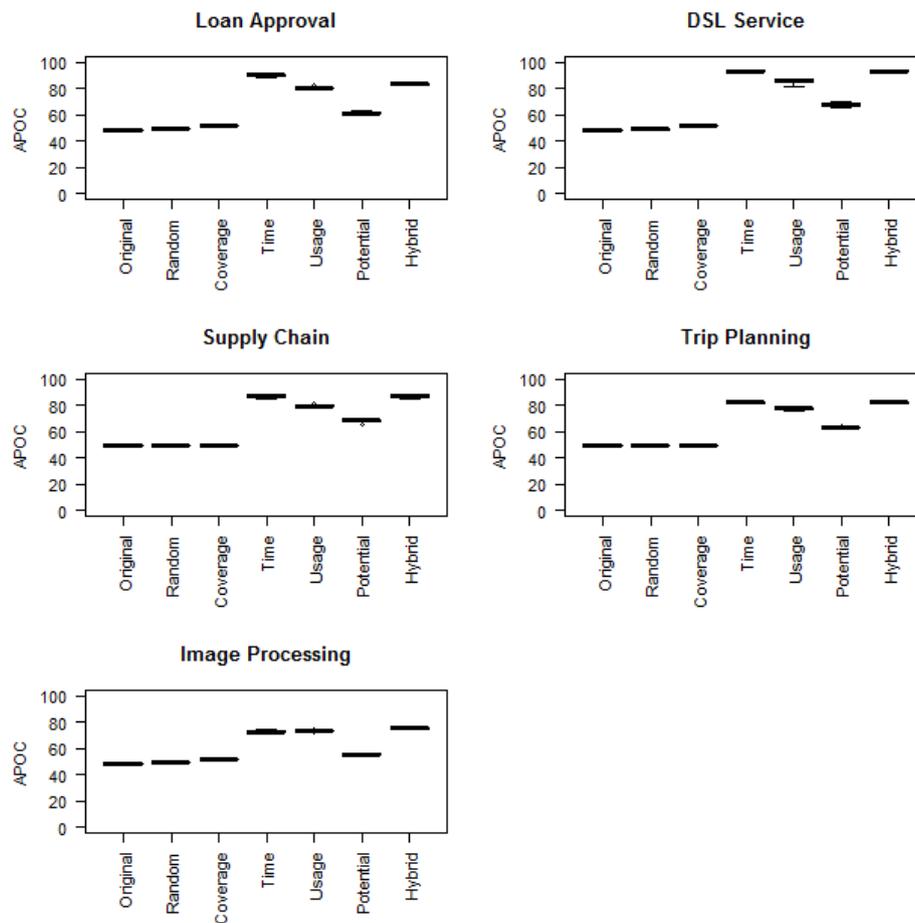


Figure 9.7: APOC results of all prioritization techniques for each service composition, individually.

- *Usage*: refers to the usage-based prioritization which orders the test cases in a test suite according to the concrete path probability of the test cases, as described in Section 7.3.3.
- *Potential*: refers to the hybrid prioritization which orders the test cases in a test suite according to the test case potential of the test cases, as described in Section 7.3.4.
- *Hybrid*: refers to the hybrid prioritization which orders the test cases in a test suite according to coverage, execution time and path probability of the test cases, as described in Section 7.3.4.

The results in Figure 9.7 – Figure 9.10 show that all our proposed priori-

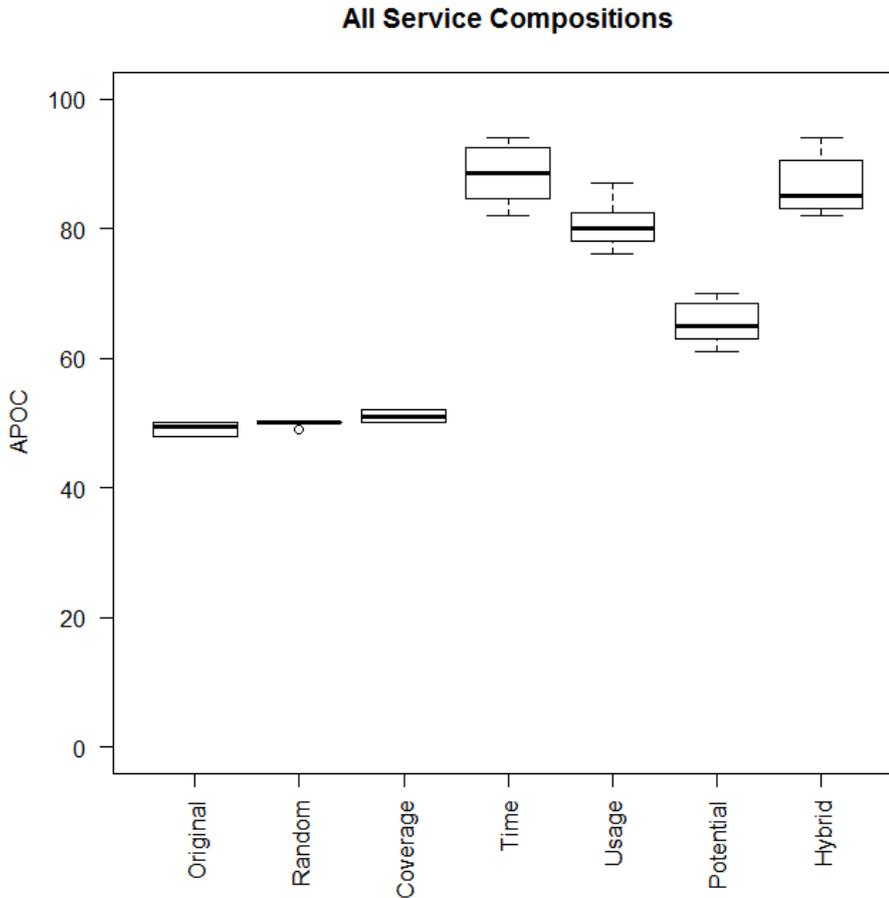


Figure 9.8: APOC results of all prioritization techniques for all service compositions, together.

zation techniques achieve higher coverage rate when compared to no prioritization and random prioritization. The time-based prioritization and the hybrid prioritization achieve the highest coverage rate when compared to the usage-based prioritization and the potential prioritization techniques, and coverage-based achieved the lowest coverage. Usage-based prioritization achieves higher coverage rate when compared to potential prioritization and coverage-based prioritization.

While coverage-based prioritization in general achieves higher coverage rate than random prioritization and no prioritization, the difference among the three techniques is subtle. Even more, in the case of *Supply Chain*, *Trip Planning*, and *Image Processing*, there are no differences at all (see Figure 9.7 and Figure 9.9). The rationale behind this is that for the considered cases in

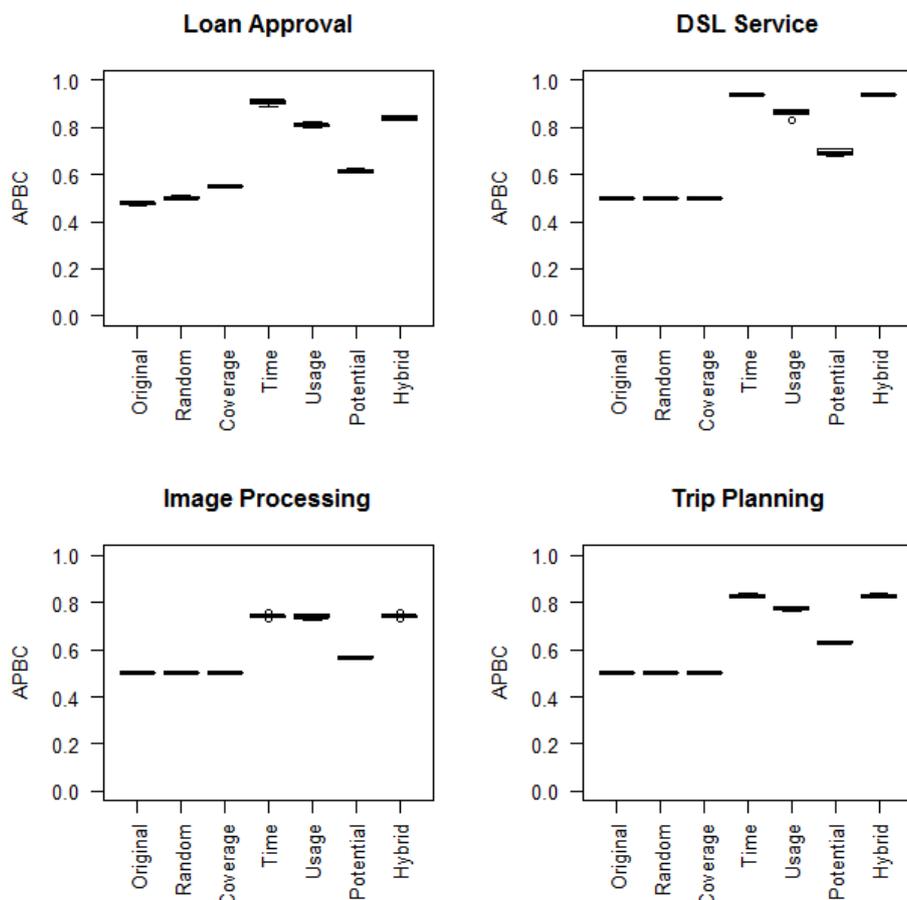


Figure 9.9: APBC results of all prioritization techniques for Loan Approval, DSL Service, Trip Planning, and Image Processing, individually. The service composition Supply Chain has no branches.

the experiment, the number of all operations (resp. branches) which need to be covered is large, making the contribution of each of the test cases small in terms of coverage. The difference between the coverage achieved by the test cases is subtle and in many cases there is no difference.

The time-based prioritization achieves the highest coverage rate as we observed that many operations (resp. branches) are covered by monitoring traces during the execution of the longest test cases. The same effect happens in the hybrid technique. Due to the subtle differences in coverage between the test cases, the test cases with longest execution time top the order of the test cases.

As coverage is low, the test case potential which results from the multiplication, is negatively affected, making the coverage rate of potential prioritization

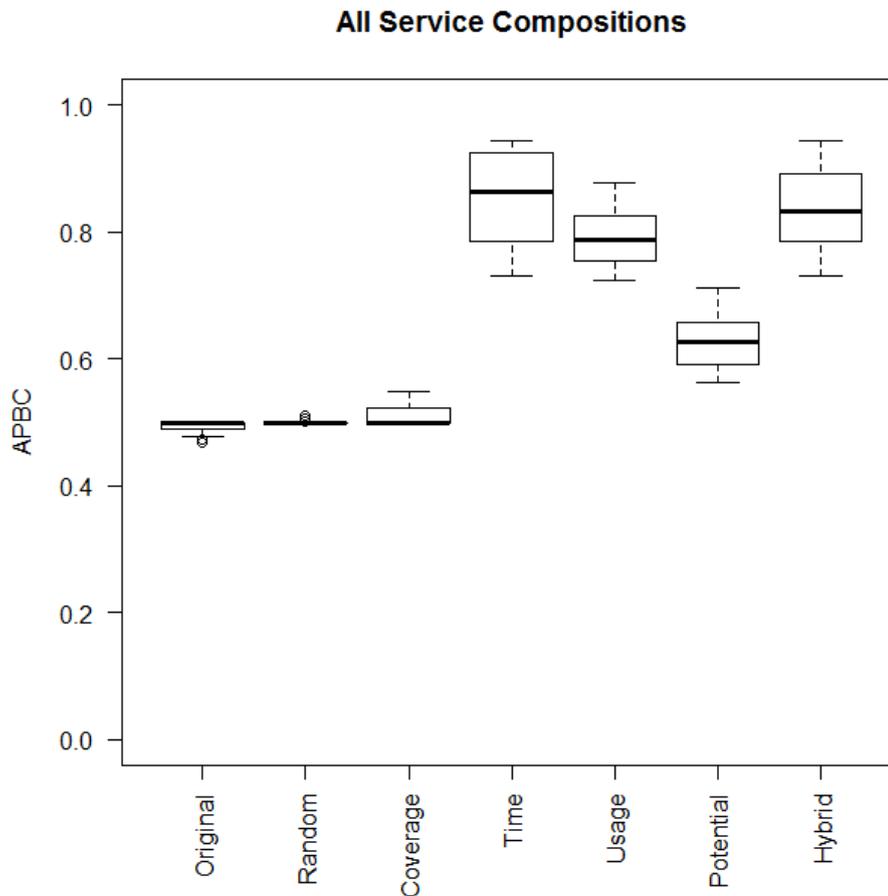


Figure 9.10: Average Percentage Branch Coverage results of all prioritization techniques for Loan Approval, DSL Service, Trip Planning, and Image Processing, together. The service composition Supply Chain has no branches.

less when compared to the other prioritizations which use it individually.

To conclude, the results of the cases considered in our experiments suggest that our proposed test case prioritization techniques can increase the operation (resp. branch) coverage rate achieved by a test suite against a service composition. However, due to the large number of operations (resp. branches) which need to be covered for dynamic service compositions, coverage information is least useful for test case prioritization. Information about test case execution time and the usage of the service composition appear to be substantial for test case prioritization.

9.5 Threats to Validity

Like any evaluation, the validity of our evaluation results are threatened by several factors. In the following, we discuss the main factors which threaten the validity of our results and conclusions.

9.5.1 Construct Validity

Construct validity concerns the variables and metrics used for measuring the target aspects. We tried to minimize this threat by using frequently used metrics for measuring similar aspects where appropriate.

9.5.2 Internal Validity

Internal validity concerns the way we designed our experiments and the influence of the design on the obtained results and conclusions. In our evaluation, this includes the simulation for obtaining execution plans, execution traces, test suite, modifications in service compositions, and the simulation period used for evaluating the test case prioritization techniques. We tried to minimize this threat by carefully designing and executing the experiments to mimic realistic situations where possible. We also used random values and repeated the experiments several times to avoid the effect of having the results by chance.

9.5.3 External Validity

Another concern is the generalization of the evaluation results. Our evaluation is not based on real service compositions. However, we tried to minimize this threat by using realistic service compositions frequently used in service-oriented computing literature covering various domains. The size of the service compositions is small when looking at the number of control-flow graph elements (i.e., nodes, transitions, and branches). However, except for the execution time analysis in Q1.1, the size of the control-flow graphs elements does not affect the results of the evaluation. In case of the execution time analysis, estimating the execution time for control-flow graphs of any size complexity can be obtained from the asymptotic analysis performed in Section 5.2.3. Finally, for obtaining response time of control-flow graph operations, we used the

publicly-available dataset QWS2 [1], which comprises measurements of Quality of Service attributes for 2,507 real-world Web services.

Chapter 10

Conclusion and Future Work

In this chapter, we summarize the research contributions and results of this thesis. We explain how the research results of the thesis contribute to advancing the state of the art analysed in Chapter 3. Furthermore, we revisit the research questions introduced in Section 1.3. We critically analyse how far those questions could be answered in this thesis. Finally, we conclude the thesis with an outlook on future research directions based on the remaining open issues.

10.1 Summary

The overall goal of the thesis is to investigate how to combine runtime monitoring and online testing to enhance coverage adequacy of self-adaptive service compositions at runtime. Towards achieving this goal, the thesis has provided five research contributions detailed in Chapter 4 – Chapter 9.

The first contribution of the thesis (Chapter 5) is an approach for determining valid execution traces for self-adaptive service compositions at runtime. The approach employs execution traces of both (online) testing and runtime monitoring. To compute coverage of a service composition, invalid execution traces are not considered. Therefore, the approach considers modifications which might result in invalid execution traces at two levels: *workflow* and *concrete services*. Where existing graph-walk algorithms employed the control-flow graphs of programs, we extended an algorithm to consider concrete service bindings of service compositions.

The second contribution of the thesis (Chapter 6) is coverage criteria for self-adaptive service compositions. The criteria consider the definition of execution plans of a service composition as a reference for coverage assessment. In addition to execution plans, the criteria consider coverage at two different scopes: *abstract service* and *overall service composition*. Combining execution plans and different coverage scopes, we have defined four new coverage criteria: (1) intra-plan-local, (2) inter-plan-local, (3) intra-plan-global, and (4) inter-plan-global.

The third contribution of the thesis (Chapter 7) is an approach for online-test-case prioritization. The goal is to achieve coverage of a service composition at a faster rate. In addition to actual test coverage which might be obtained from test suite execution against a service composition, the proposed test case prioritization approach considers the potential coverage of the service composition which might be obtained from runtime monitoring of actual usage of the service composition. The approach exploits the execution time of test cases as well as the usage profile of a service composition. Considering both the actual test coverage from test suite execution and potential coverage from runtime monitoring, we developed test case prioritization techniques classified into four main categories: (1) coverage-based, (2) time-based, (3) usage-based, and (4) hybrid.

The fourth contribution of the thesis (Chapter 8) is a framework for runtime monitoring and online testing of services and service compositions (called PROSA). PROSA exploits synergies between runtime monitoring and online testing in order to enhance coverage of service compositions. The modules of the PROSA framework provide the technical support for the aforementioned contributions of the thesis.

The last contribution of thesis (Chapter 9) is an empirical evaluation of the contributions of the thesis. The evaluation is performed by means of controlled experiments using five service compositions frequently used by other researchers for testing and dynamic composition of services. Where the response time of services is needed, the evaluation used the publicly-available dataset QWS2 [1]. The QWS2 dataset comprises measurements of Quality of Service attributes for 2,507 real-world Web services.

10.2 Revisiting Research Questions

In Section 1.3 we outlined the research questions addressed in this thesis. In the following, we revisit these research questions and explain the answers of our research contributions to these research questions. Additionally, we summarize limitations of our contributions.

Research Question I:

How to assess coverage of self-adaptive service compositions at runtime?

In general, coverage assessment involves two key elements: (1) a set of execution traces against which coverage is measured, (2) coverage adequacy criteria which define requirements on the execution traces to be considered adequate.

Regarding the first element (i.e., execution traces), the thesis proposed an approach for determining the execution traces to be considered for coverage assessment, taking into account the dynamic modifications at the runtime of a self-adaptive service composition. The evaluation results suggest that the execution time of our algorithm for determining the execution traces is not high (1 – 40 ms). Therefore, the algorithm can be used at runtime to support timely decision making.

Regarding coverage adequacy, the thesis introduced coverage adequacy criteria which indicate whether or not a set of execution traces of a service composition is adequate, taking into account the realizations of the service composition (i.e., execution plans). The coverage criteria consider relevant scopes of coverage based on the structure of the service composition. The evaluation results suggest that, using execution plans a faster coverage is achieved when compared to considering all possible plans for all candidate service bindings of the service composition.

One limitation concerning our approach for determining invalid traces is that it does not consider the internal structure of the bound services in a service composition. This is due to the fact that service providers tend not to reveal the internal structure of their services and offer only the interfaces of their services. However, considering the internal structure of the bound

services might allow to determine invalid traces at a more fine-grained level, thereby potentially increasing the accuracy.

Additionally, the contributions of the thesis related to research question I are limited to the assessment of structural coverage based on entities of the control-flow graph of a service composition. Although structural coverage is widely used in software testing, there exist also data-flow coverage and state coverage. Data-flow coverage is based on the data-flow graph of a tested program and the state coverage is based on the state model of the software. We did not consider data-flow coverage and state coverage of service compositions.

Research Question II:

How to combine runtime monitoring and online testing to enhance coverage adequacy at runtime?

Based on the results of coverage assessment, the achieved coverage of a service composition might turn out to be insufficient according to pre-defined coverage criteria and a coverage level. The idea followed in the thesis to enhance the coverage is to perform online testing to obtain additional execution traces. As testing is performed in parallel to the actual usage of the service composition, synergies with runtime monitoring are exploited to minimize the associated costs with online testing.

This thesis introduced techniques for online test case selection and prioritization with the goal to achieve coverage faster. Some of the proposed techniques consider the potential coverage which might be obtained from the normal usage of the service composition. The evaluation results indicate that some of the proposed techniques achieve a faster coverage compared with random test case selection. Additionally, the PROSA framework introduced in the thesis provides the technical support required for the thesis contributions.

A pre-requisite for applying our prioritization techniques is determining which test cases need to be used for testing the service composition. The thesis assumes that a repository of test cases for the service composition already exists from which one could select the test cases.

The thesis does not provide an approach for the cost-benefit analysis of online testing and the achieved coverage which might further support the deci-

sion of whether or not to perform online testing to enhance coverage in a more informed way.

10.3 Future Work

The discussion in Section 10.2 indicated some aspects related to the combined usage of runtime monitoring and online testing to enhance the coverage of service composition not addressed in the thesis. These aspects open up possibilities for future research:

- Data-flow coverage and state coverage of service composition: As discussed in Section 10.2, the contributions of the thesis are limited to the structural coverage of service composition. However, data-flow coverage and state-based coverage are also key in software testing. In particular, data-flow coverage is of importance for data-intensive service composition and state coverage is key for statefull service composition. There are research results for data-flow testing of service composition (see [75, 54, 55]) and for state-based testing of service composition (see [37]). It would be interesting to study how existing approaches can be used and/or extended for online testing and runtime monitoring.
- Internal coverage of third-party services: As discussed in Section 10.2, the contributions of thesis consider a coarse-grain coverage of third-party services (i.e., operation coverage). There exist solutions to obtain internal coverage information of third-party services (see [106, 12, 17, 11]). It would be interesting to study potential advantages obtained from considering a more fine-grained coverage information of third-party services.
- Synergies with runtime verification of service composition: As discussed in Section 1.1, the thesis focuses on exploiting synergies between two key runtime quality assurance techniques: runtime monitoring and online testing. Runtime verification of service composition is another key runtime quality assurance technique for service composition. It would be interesting to study potential synergies obtained from combining runtime verification of service composition with our proposed approach. For

example, online test cases could be generated from the results of runtime verification of service composition.

- **Cost-benefit analysis of online testing:** As discussed in Section 8, online testing can cause additional costs. Cost models which take into account factors related to the cost of online testing might support the decision when to execute online testing and when not. The costs of online testing need to be compared with the benefits of using online testing.
- **Evaluation:** The evaluation conducted in the thesis is based on experiments using example service compositions, frequently used in service-oriented computing research to compensate for the lack of open source real-world service compositions available for evaluation purposes. Therefore, the evaluation of the thesis contributions should be extended to consider real-world service compositions. Additionally, as discussed in Section 9, there exist other types of evaluation in software engineering than controlled experiments. These include for example case study and questionnaire. For instance, the evaluation of our proposed coverage criteria using questionnaire involving testing experts and practitioners would be valuable. Moreover, the results of the evaluation of the PROSA framework using a case study from the industry would be key for the uptake of PROSA.

Further research can be performed following these open issues.

Appendix A

Detailed Results from the Evaluation

We present detailed evaluation results for goal 3. In particular, for all abstract services in the service compositions we present the inter-plan local coverage results from the evaluation performed in Section 9.3.1.

A.1 Results for 1000 Execution Traces

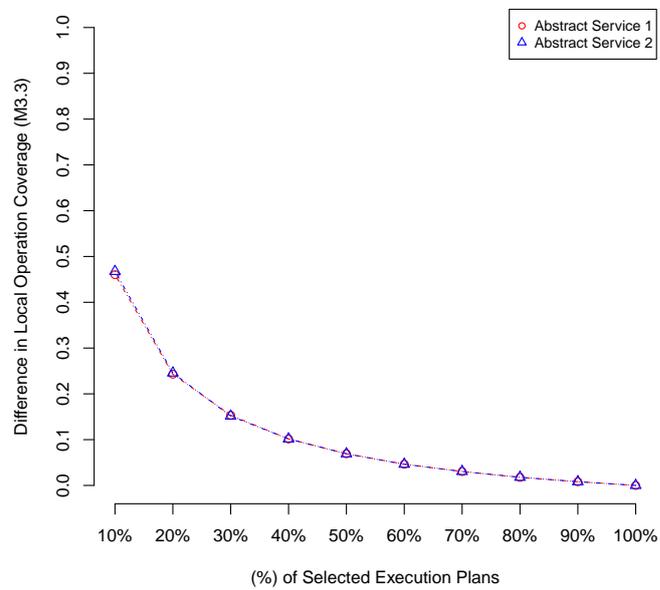


Figure A.1: Results measured with metric M3.3 for Loan Approval using all amounts of selected execution plans and 1000 execution traces.

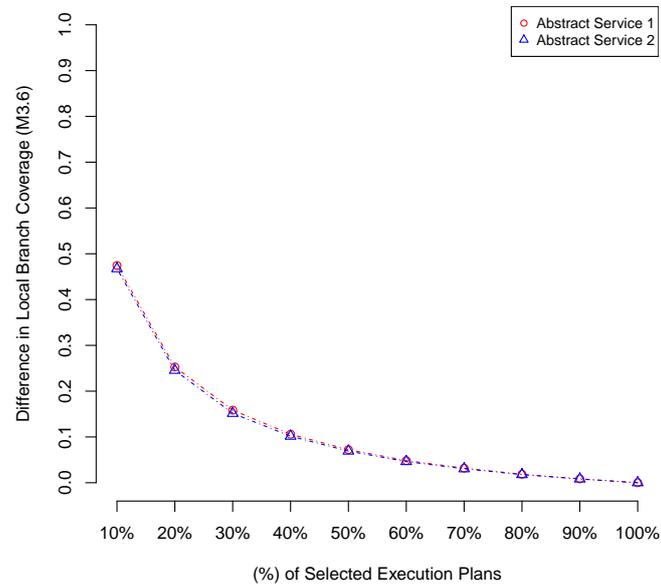


Figure A.2: Results measured with metric M3.6 for Loan Approval using all amounts of selected execution plans and 1000 execution traces.

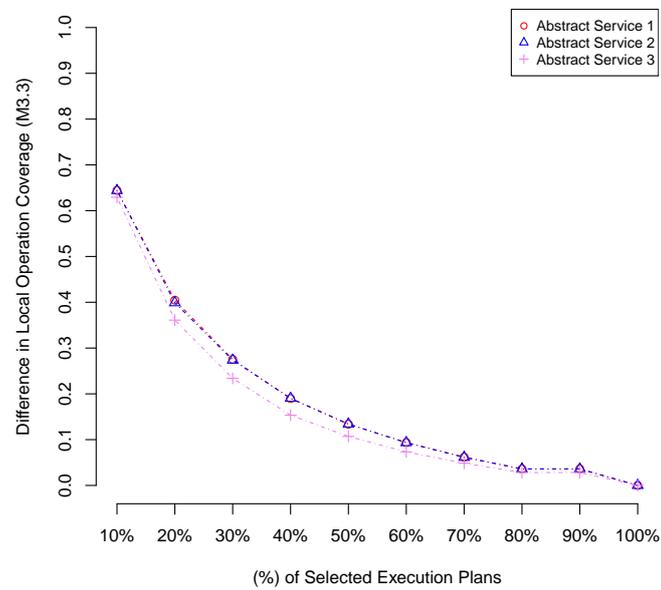


Figure A.3: Results measured with metric M3.3 for DSL Service using all amounts of selected execution plans and 1000 execution traces.

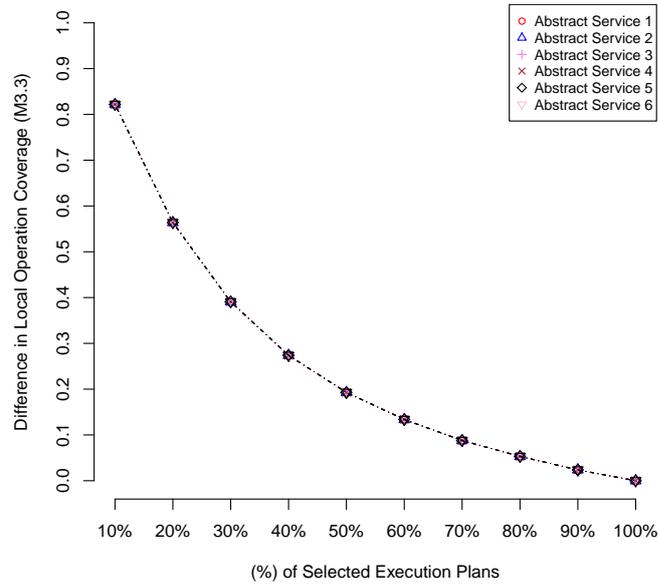


Figure A.4: Results measured with metric M3.3 for Supply Chain using all amounts of selected execution plans and 1000 execution traces.

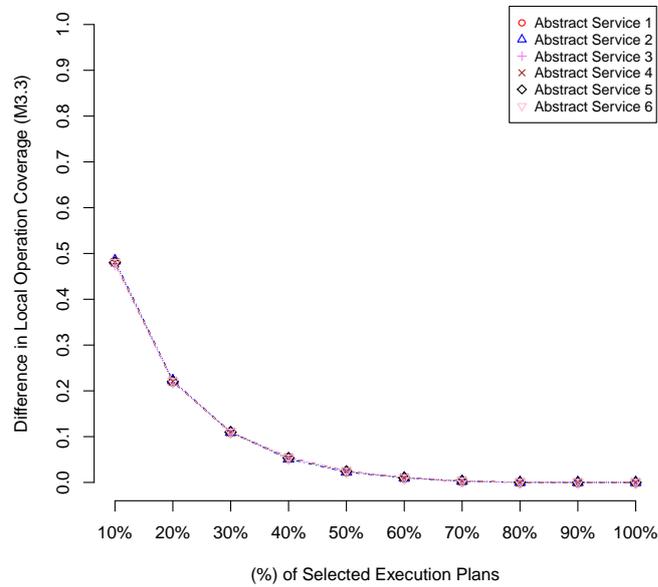


Figure A.5: Results measured with metric M3.3 for Trip Planning using all amounts of selected execution plans and 1000 execution traces.

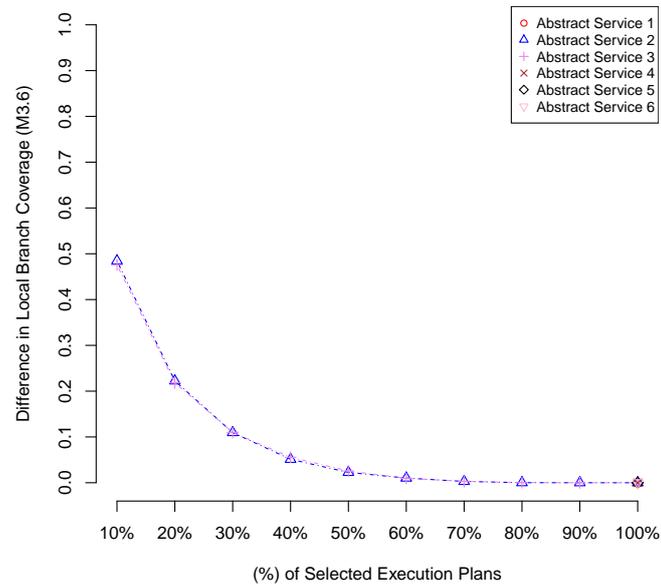


Figure A.6: Results measured with metric M3.6 for Trip Planning using all amounts of selected execution plans and 1000 execution traces.

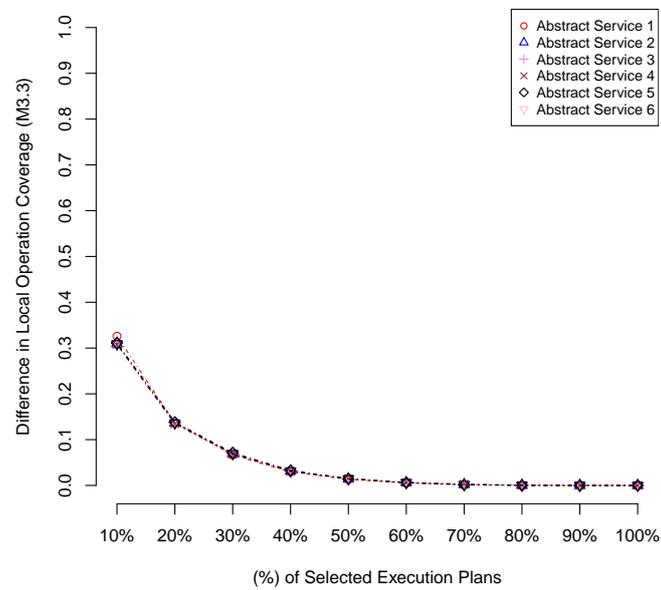


Figure A.7: Results measured with metric M3.3 for Image Processing using all amounts of selected execution plans and 1000 execution traces.

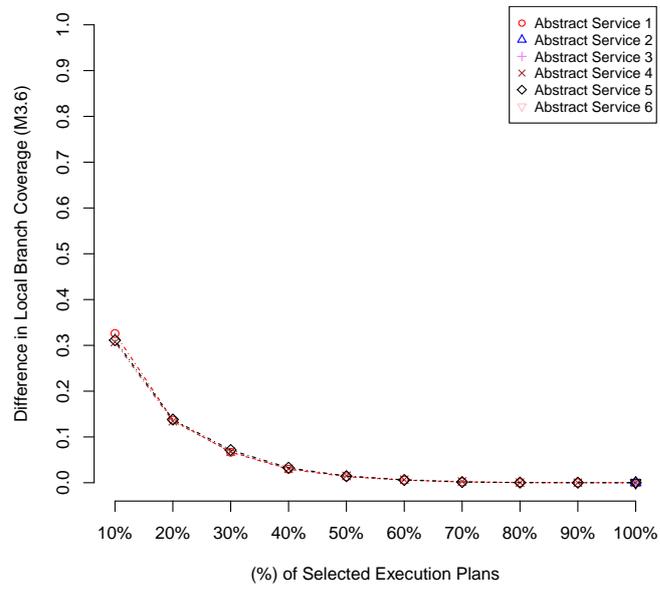


Figure A.8: Results measured with metric M3.6 for Image Processing using all amounts of selected execution plans and 1000 execution traces.

A.2 Results for 2000 Execution Traces

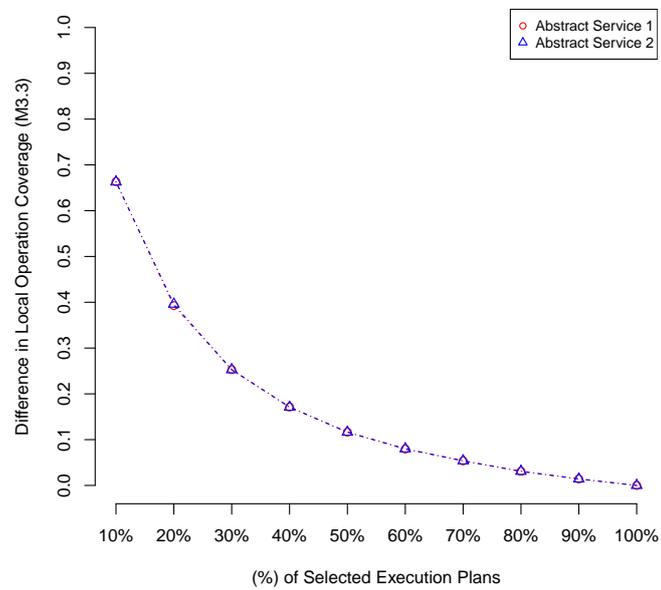


Figure A.9: Results measured with metric M3.3 for Loan Approval using all amounts of selected execution plans and 2000 execution traces.

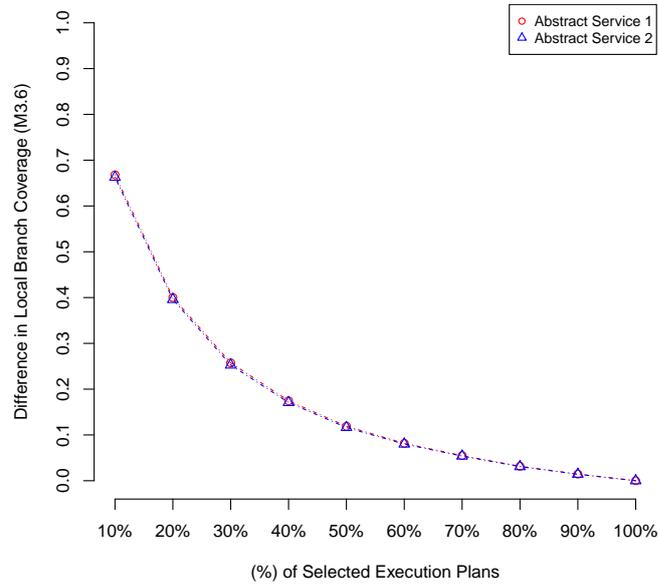


Figure A.10: Results measured with metric M3.6 for Loan Approval using all amounts of selected execution plans and 2000 execution traces.

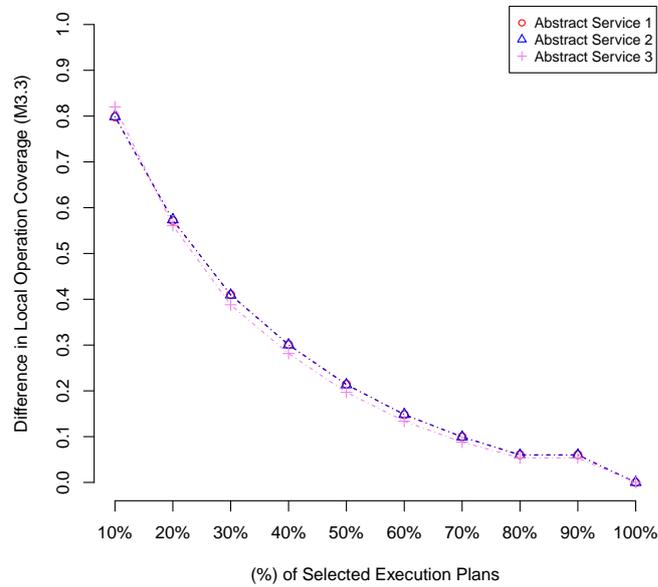


Figure A.11: Results measured with metric M3.3 for DSL Service using all amounts of selected execution plans and 2000 execution traces.

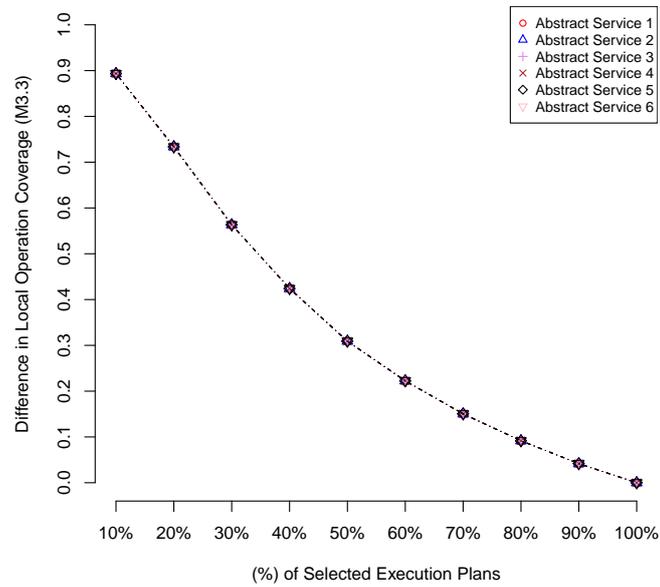


Figure A.12: Results measured with metric M3.3 for Supply Chain using all amounts of selected execution plans and 2000 execution traces.

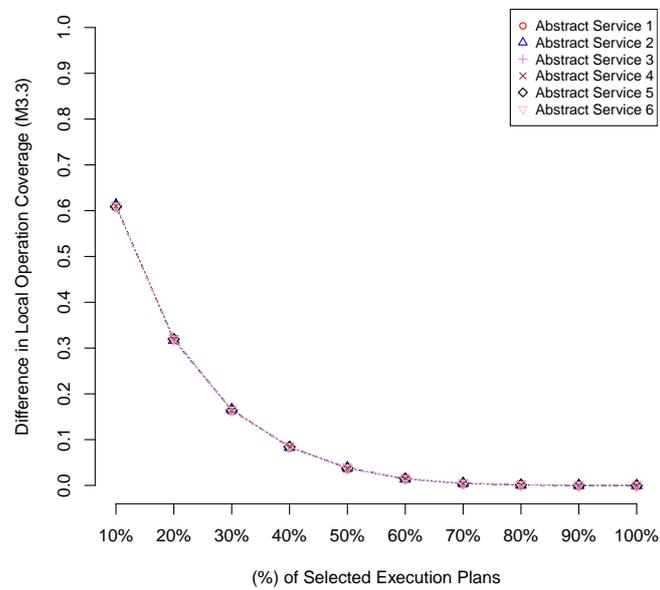


Figure A.13: Results measured with metric M3.3 for Trip Planning using all amounts of selected execution plans and 2000 execution traces.

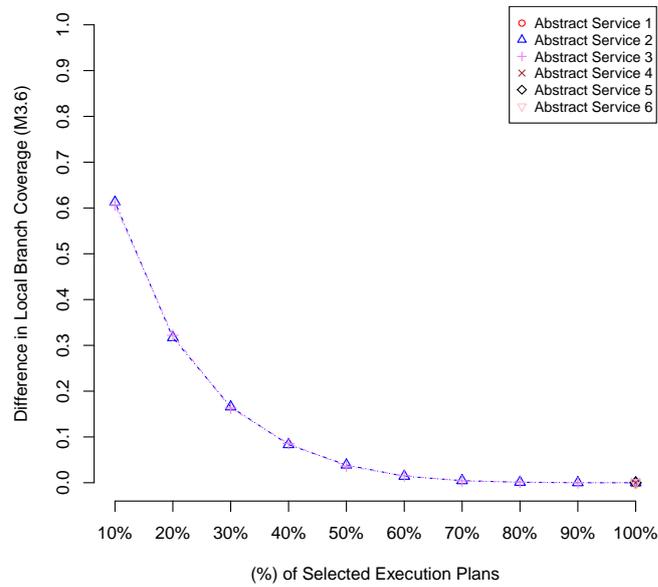


Figure A.14: Results measured with metric M3.6 for Trip Planning using all amounts of selected execution plans and 2000 execution traces.

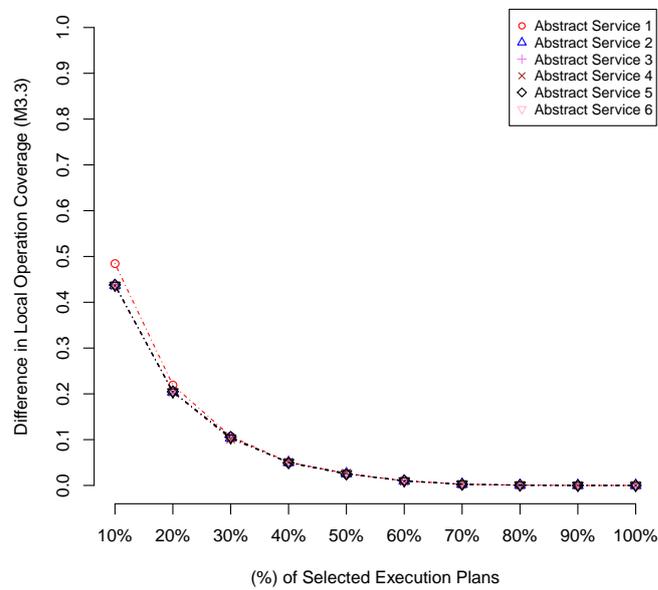


Figure A.15: Results measured with metric M3.3 for Image Processing using all amounts of selected execution plans and 2000 execution traces.

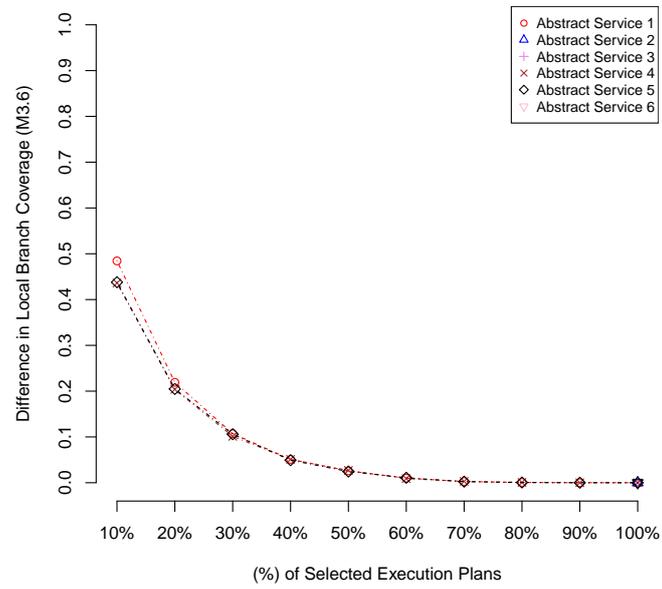


Figure A.16: Results measured with metric M3.6 for Image Processing using all amounts of selected execution plans and 2000 execution traces.

A.3 Results for 3000 Execution Traces

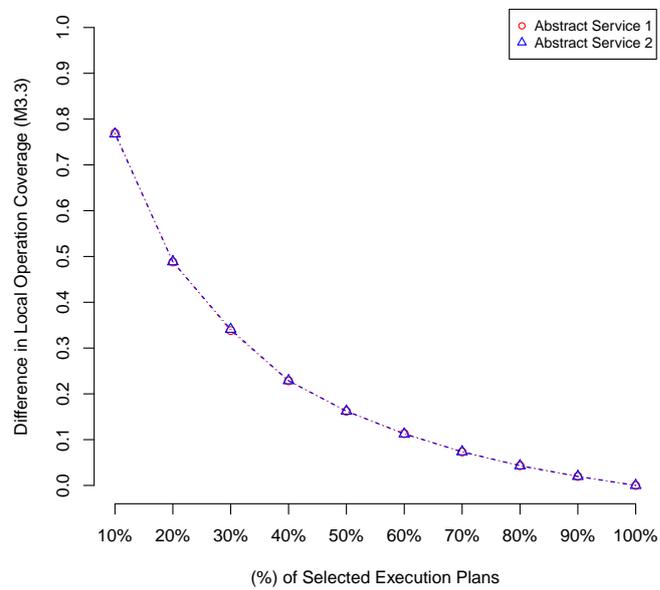


Figure A.17: Results measured with metric M3.3 for Loan Approval using all amounts of selected execution plans and 3000 execution traces.

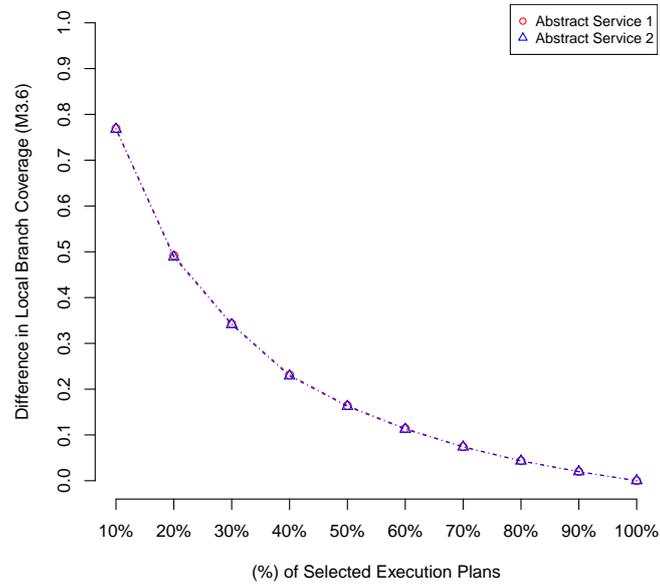


Figure A.18: Results measured with metric M3.6 for Loan Approval using all amounts of selected execution plans and 3000 execution traces.

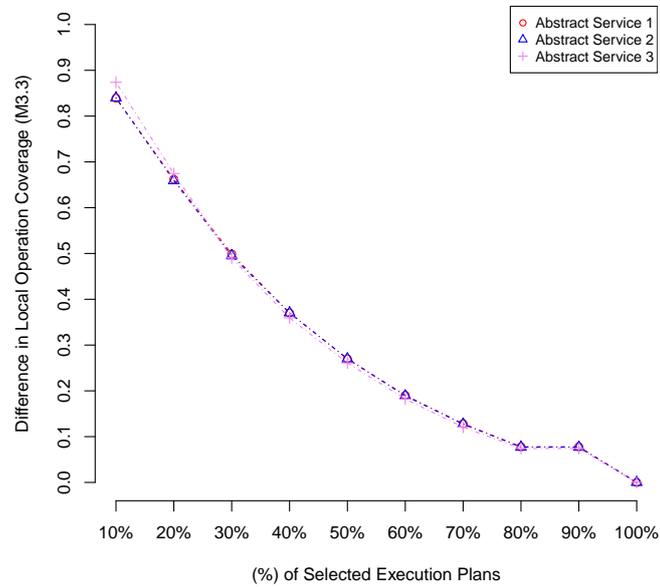


Figure A.19: Results measured with metric M3.3 for DSL Service using all amounts of selected execution plans and 3000 execution traces.

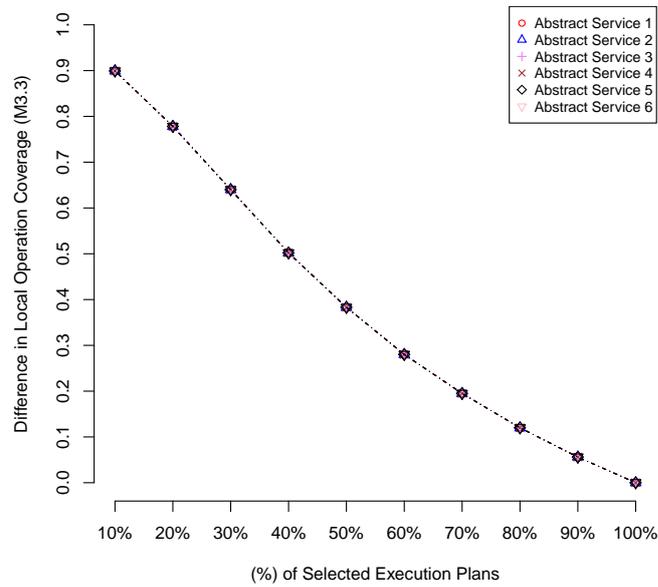


Figure A.20: Results measured with metric M3.3 for Supply Chain using all amounts of selected execution plans and 3000 execution traces.

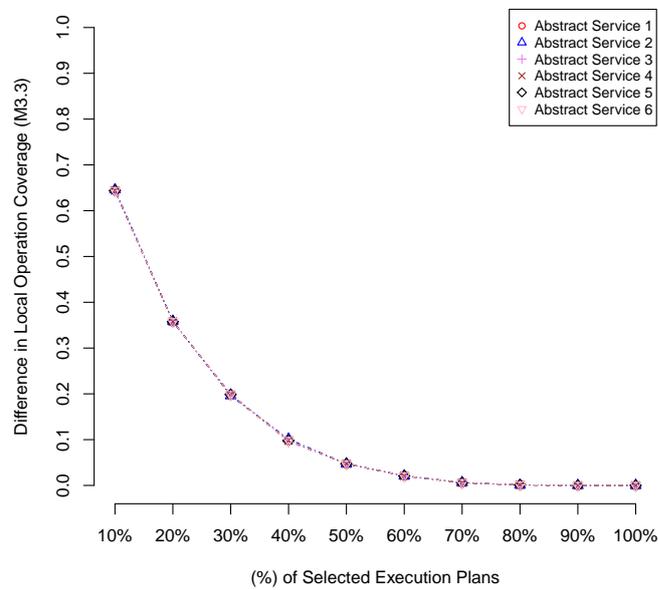


Figure A.21: Results measured with metric M3.3 for Trip Planning using all amounts of selected execution plans and 3000 execution traces.

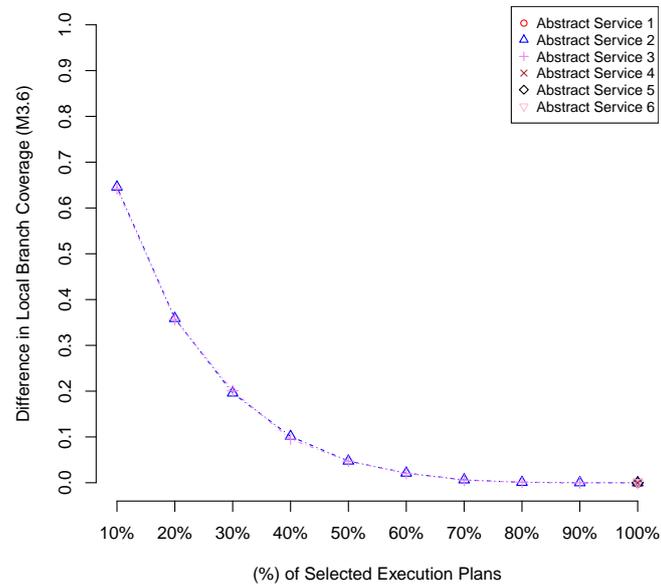


Figure A.22: Results measured with metric M3.6 for Trip Planning using all amounts of selected execution plans and 3000 execution traces.

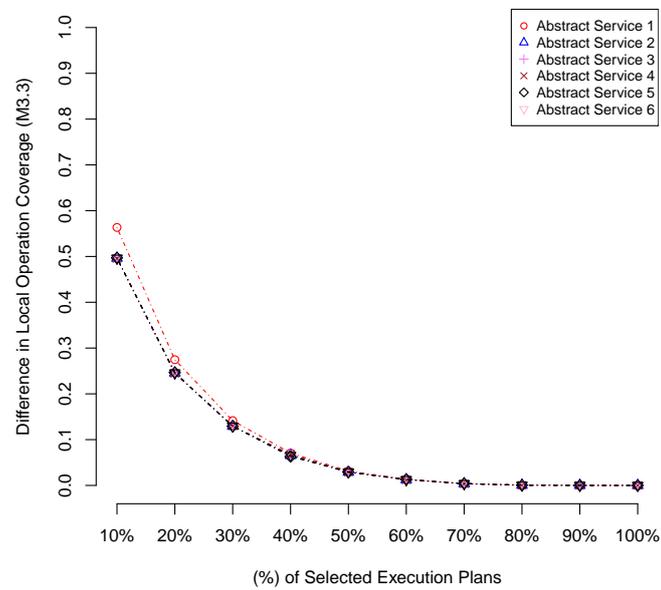


Figure A.23: Results measured with metric M3.3 for Image Processing using all amounts of selected execution plans and 3000 execution traces.

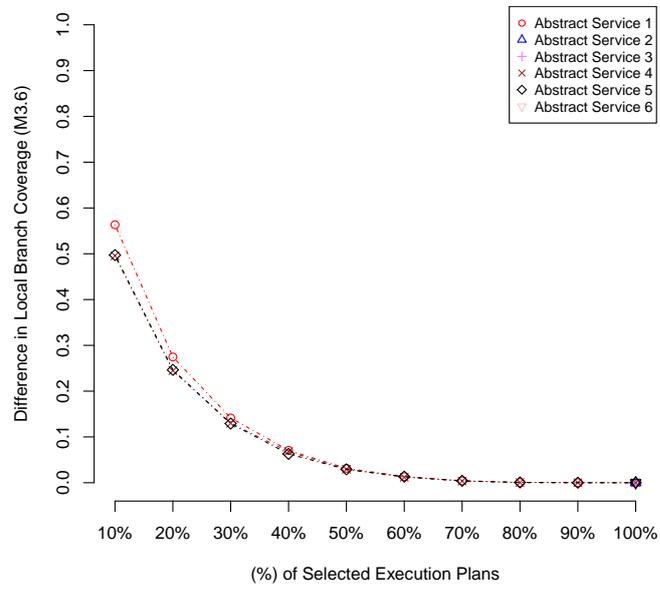


Figure A.24: Results measured with metric M3.6 for Image Processing using all amounts of selected execution plans and 3000 execution traces.

A.4 Results for 4000 Execution Traces

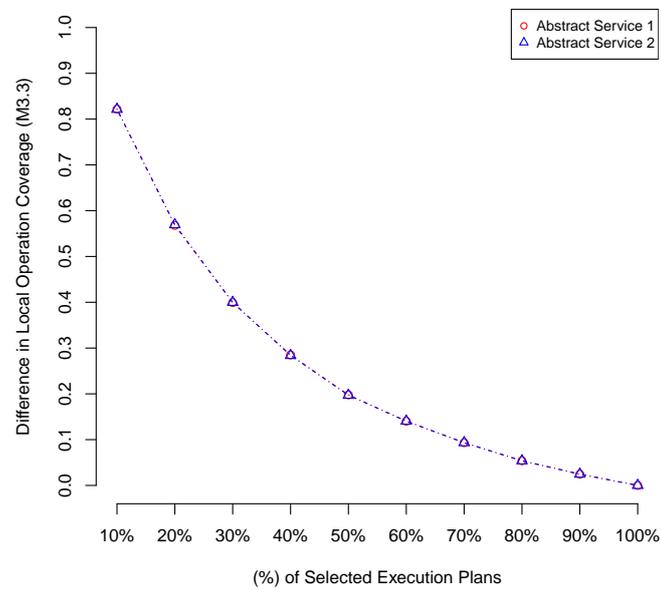


Figure A.25: Results measured with metric M3.3 for Loan Approval using all amounts of selected execution plans and 4000 execution traces.

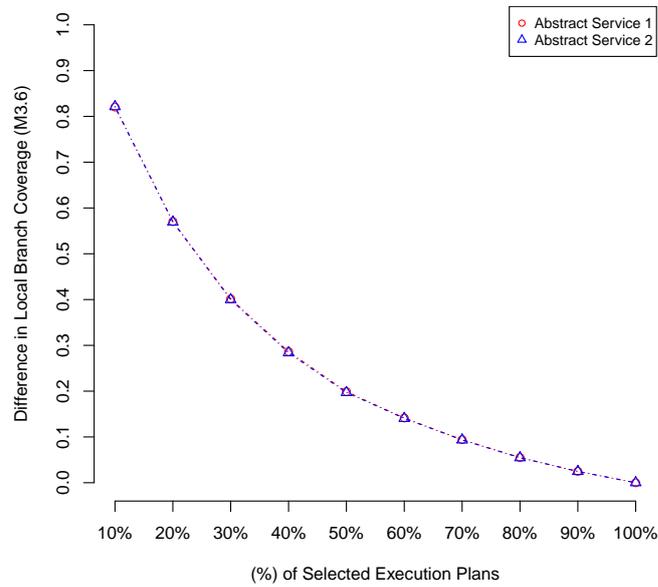


Figure A.26: Results measured with metric M3.6 for Loan Approval using all amounts of selected execution plans and 4000 execution traces.

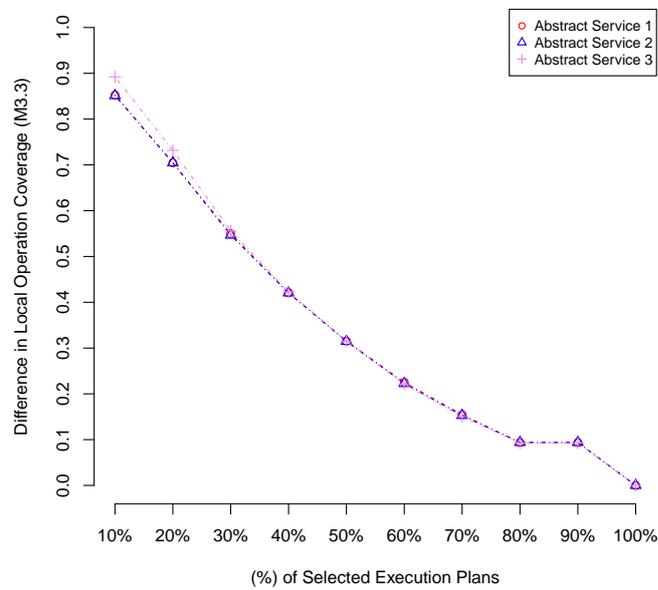


Figure A.27: Results measured with metric M3.3 for DSL Service using all amounts of selected execution plans and 4000 execution traces.

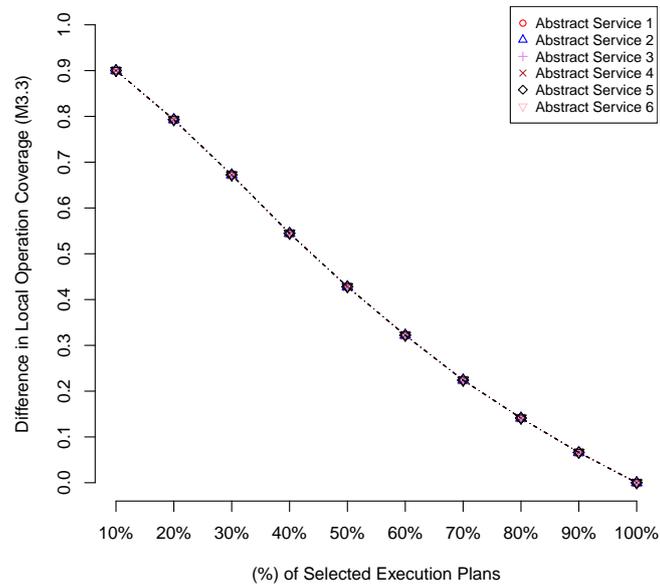


Figure A.28: Results measured with metric M3.3 for Supply Chain using all amounts of selected execution plans and 4000 execution traces.

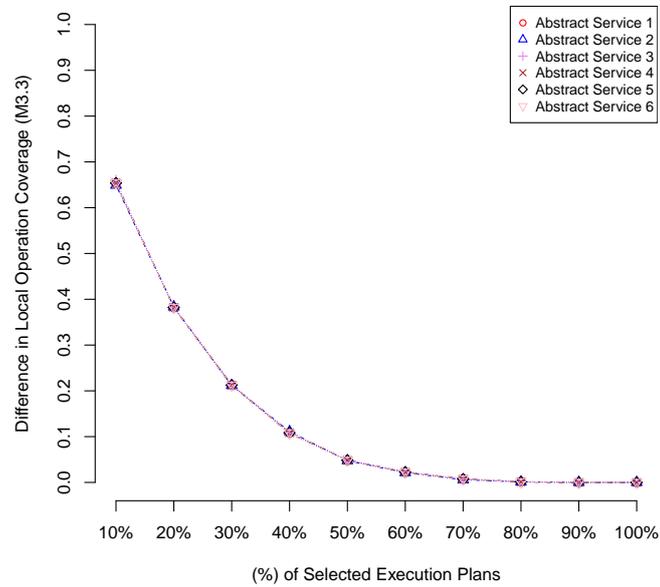


Figure A.29: Results measured with metric M3.3 for Trip Planning using all amounts of selected execution plans and 4000 execution traces.

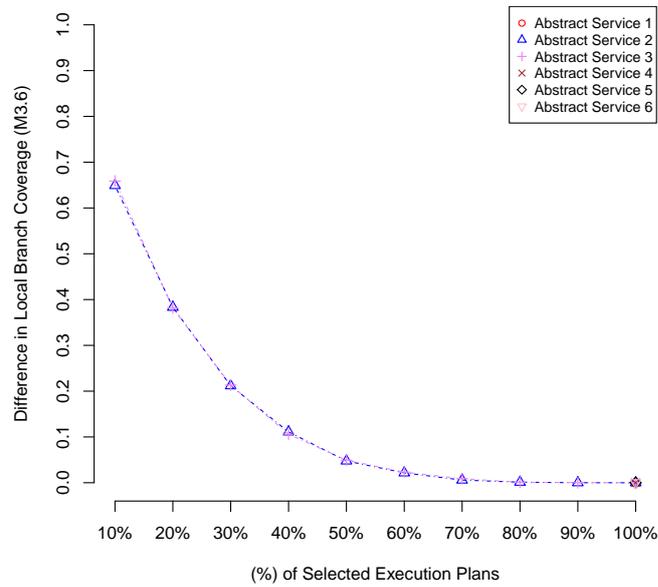


Figure A.30: Results measured with metric M3.6 for Trip Planning using all amounts of selected execution plans and 4000 execution traces.

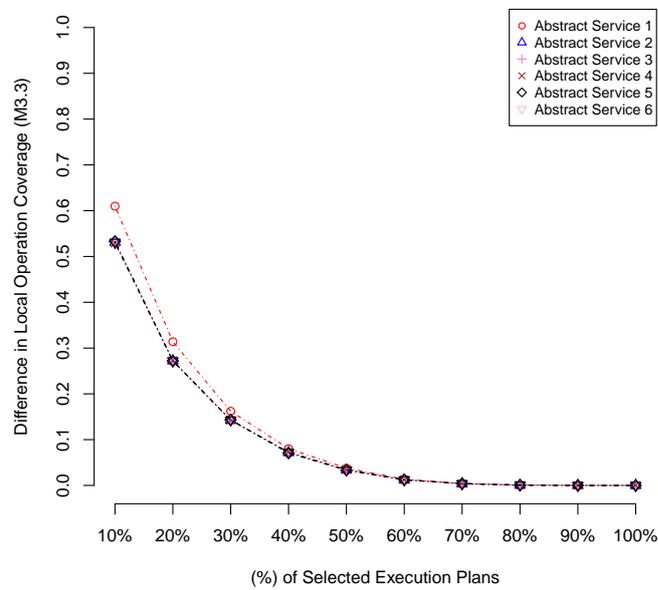


Figure A.31: Results measured with metric M3.3 for Image Processing using all amounts of selected execution plans and 4000 execution traces.

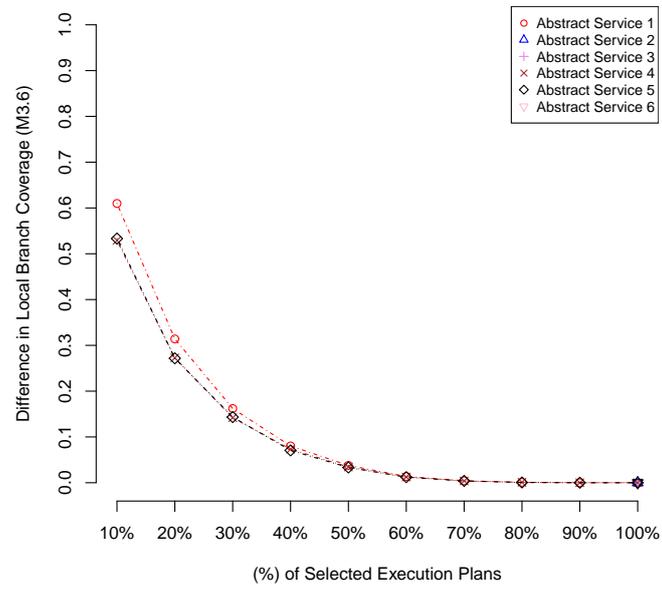


Figure A.32: Results measured with metric M3.6 for Image Processing using all amounts of selected execution plans and 4000 execution traces.

A.5 Results for 5000 Execution Traces

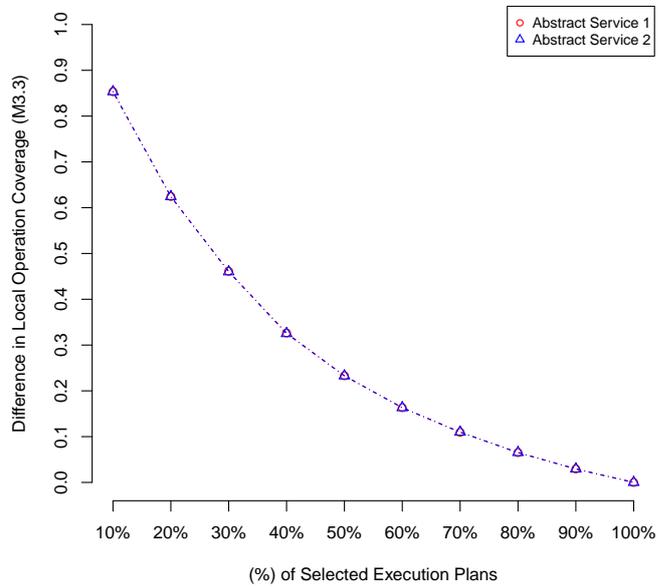


Figure A.33: Results measured with metric M3.3 for Loan Approval using all amounts of selected execution plans and 5000 execution traces.

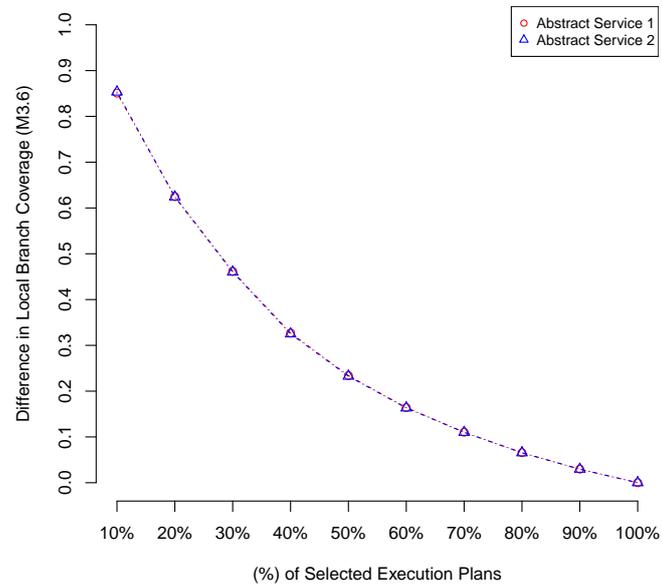


Figure A.34: Results measured with metric M3.6 for Loan Approval using all amounts of selected execution plans and 5000 execution traces.

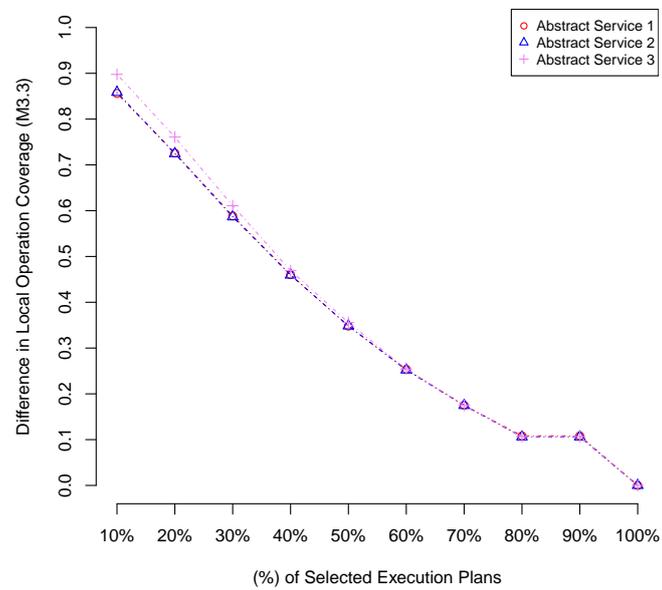


Figure A.35: Results measured with metric M3.3 for DSL Service using all amounts of selected execution plans and 5000 execution traces.

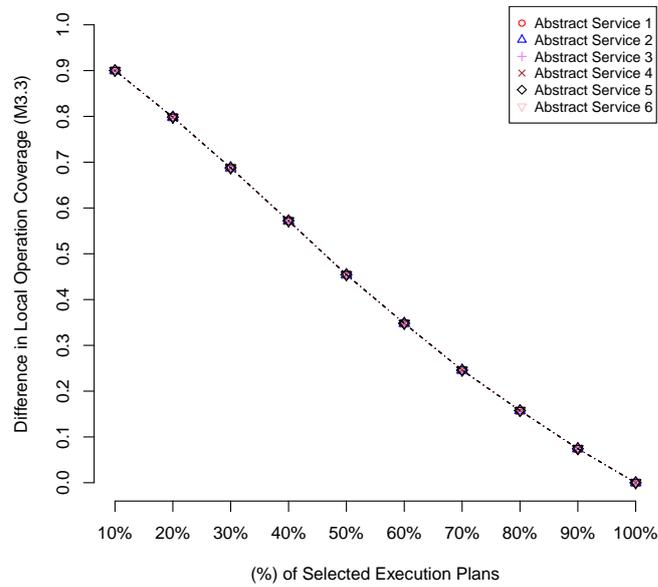


Figure A.36: Results measured with metric M3.3 for Supply Chain using all amounts of selected execution plans and 5000 execution traces.

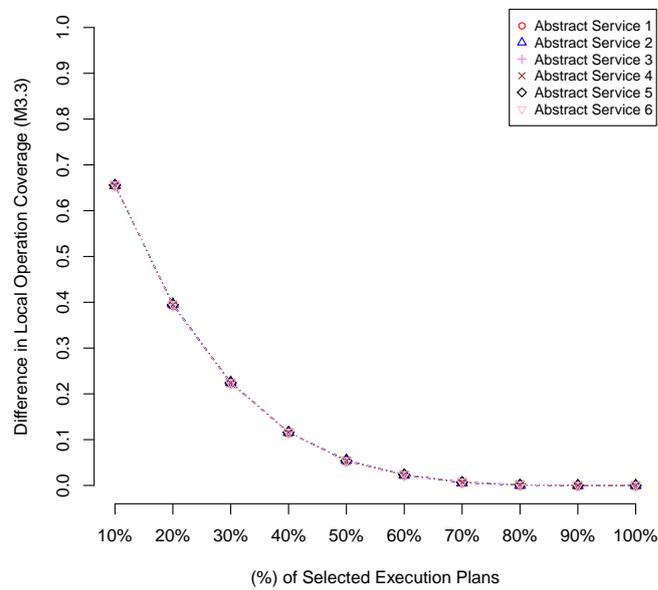


Figure A.37: Results measured with metric M3.3 for Trip Planning using all amounts of selected execution plans and 5000 execution traces.

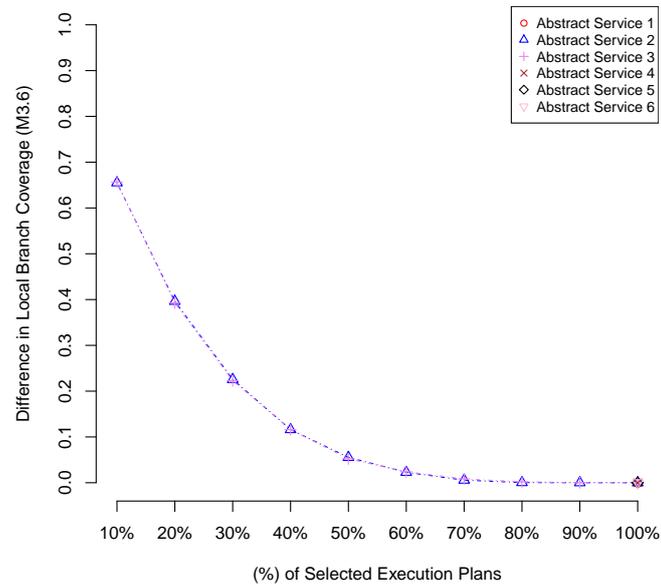


Figure A.38: Results measured with metric M3.6 for Trip Planning using all amounts of selected execution plans and 5000 execution traces.

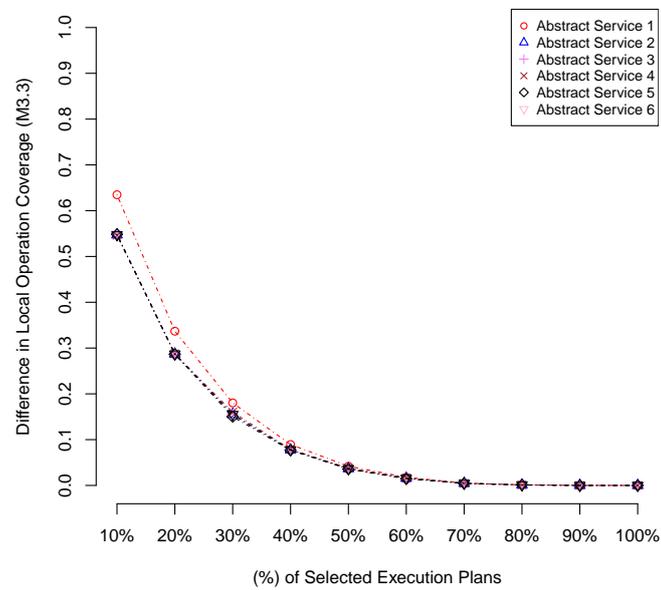


Figure A.39: Results measured with metric M3.3 for Image Processing using all amounts of selected execution plans and 5000 execution traces.

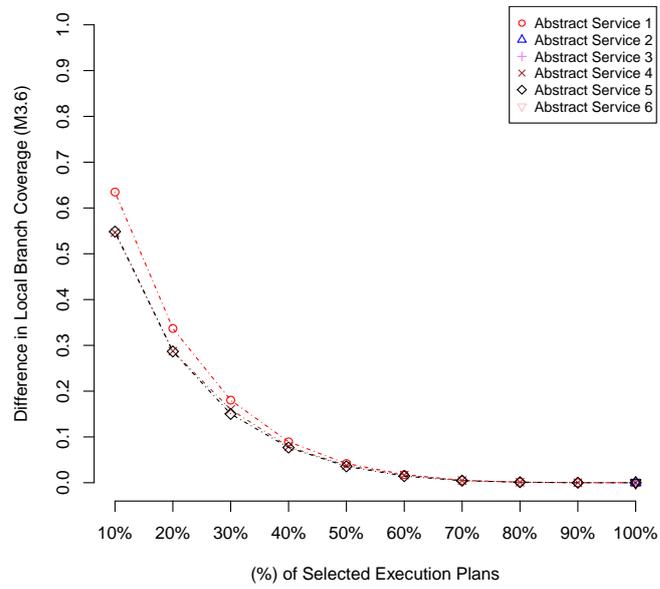


Figure A.40: Results measured with metric M3.6 for Image Processing using all amounts of selected execution plans and 5000 execution traces.

References

- [1] Eyhab Al-Masri and Qusay H. Mahmoud. Investigating web services on the world wide web. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 795–804, New York, NY, USA, 2008. ACM.
- [2] Midhat Ali, Antonia Bertolino, Francesco De Angelis, Guglielmo De Angelis, Daniele Fani, and Andrea Polini. An extensible framework for online testing of choreographed services. *Computer*, 47(2):23–29, February 2014.
- [3] Mohammad Alrifai, Thomas Risse, and Wolfgang Nejdl. A hybrid approach for efficient web service composition with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 6(2):7:1–7:31, June 2012.
- [4] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 11–20, New York, NY, USA, 2010. ACM.
- [5] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, June 2007.
- [6] Xiaoying Bai, Yinong Chen, and Zhongkui Shao. Adaptive web services testing. In *31st Annual International Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 233–236, July 2007.
- [7] Xiaoying Bai, Guilan Dai, Dezheng Xu, and Wei-Tek Tsai. A multi-agent based framework for collaborative testing on web services. In *Proceedings*

- of the *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA '06)*, SEUS-WCCIA '06, pages 205–210, Washington, DC, USA, 2006. IEEE Computer Society.
- [8] Xiaoying Bai, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. Ontology-based test modeling and partition testing of web services. In *Proceedings of the 2008 IEEE International Conference on Web Services, ICWS '08*, pages 465–472, Washington, DC, USA, 2008. IEEE Computer Society.
- [9] Xiaoying Bai, Yongli Liu, Lijun Wang, and Peide Zhong. Model-based monitoring and policy enforcement of services. *Simulation Modelling Practice and Theory*, 17(8):1399–1412, sep 2009.
- [10] Xiaoying Bai, Dezheng Xu, and Guilan Dai. Dynamic reconfigurable testing of service-oriented architecture. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01, COMPSAC '07*, pages 368–378, Washington, DC, USA, 2007. IEEE Computer Society.
- [11] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software*, 84(4):655–668, April 2011.
- [12] Cesare Bartolini, Antonia Bertolino, and Eda Marchetti. Introducing service-oriented coverage testing. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, 2008. ASE Workshops 2008*, pages 57–64, September 2008.
- [13] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering, FOSE '07*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.
- [14] Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. (role)CAST: A framework for on-line service testing. In *Proceedings*

of the 7th International Conference on Web Information Systems and Technologies (WEBIST 2011), pages 13–18. SciTePress, 2011.

- [15] Antonia Bertolino, Guglielmo De Angelis, Sampo Kellomaki, and Andrea Polini. Enhancing service federation trustworthiness through online testing. *Computer*, 45(1):66–72, January 2012.
- [16] Antonia Bertolino, Eda Marchetti, and Andrea Morichetta. Adequate monitoring of service compositions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 59–69, New York, NY, USA, 2013. ACM.
- [17] Antonia Bertolino and Andrea Polini. Soa test governance: Enabling service integration testing across organization and technology borders. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW '09*, pages 277–286, Washington, DC, USA, 2009. IEEE Computer Society.
- [18] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Suktaran. Regression test selection techniques: A survey. *Informatica*, 35:289, 2011.
- [19] Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, Gianpiero Esposito, and Valentina Mazza. Using test cases as contract to ensure service compliance across releases. In *Proceedings of the Third International Conference on Service-Oriented Computing, ICSOC'05*, pages 87–100, Berlin, Heidelberg, 2005. Springer-Verlag.
- [20] Gerardo Canfora and Massimiliano Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, March 2006.
- [21] Gerardo Canfora and Massimiliano Di Penta. *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, chapter Service-Oriented Architectures Testing: A Survey, pages 78–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

- [22] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, GECCO '05*, pages 1069–1075, New York, NY, USA, 2005. ACM.
- [23] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, October 2008.
- [24] Venkata U. B. Challagulla, Farokh B. Bastani, Raymond A. Paul, Wei-Tek Tsai, and Yinong Chen. A machine learning-based reliability assessment model for critical software systems. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01, COMPSAC '07*, pages 79–86, Washington, DC, USA, 2007. IEEE Computer Society.
- [25] Wing-Kwong Chan, Shing chi Cheung, and Karl Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.
- [26] Lin Chen, Ziyuan Wang, Lei Xu, Hongmin Lu, and Baowen Xu. Test case prioritization for web service regression testing. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE)*, pages 173–178, 2010.
- [27] Betty Cheng and et al. Software engineering for self-adaptive systems: A research roadmap. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 1–26. Springer, 2009.
- [28] Pavan Kumar Chittimalli and Mary Jean Harrold. Re-computing coverage information to assist regression testing. In *IEEE International Conference on Software Maintenance, 2007. ICSM 2007*, pages 164–173, October 2007.

- [29] Pavan Kumar Chittimalli and Mary Jean Harrold. Recomputing coverage information to assist regression testing. *IEEE Transactions on Software Engineering*, 35(4):452–469, jul 2009.
- [30] Rogerio de Lemos and et al. Software engineering for self-adaptive systems: A second research roadmap. In Rogerio de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems*, number 10431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.
- [31] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, December 2004.
- [32] Peter H. Deussen, George Din, and Ina Schieferdecker. An on-line test platform for component-based systems. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, Washington, DC, USA, 2002. IEEE Computer Society.
- [33] Massimiliano Di Penta, Marcello Bruno, Gianpiero Esposito, and et al. Web services regression testing. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 205 – 234. Springer, 2007.
- [34] Massimiliano Di Penta, Gerardo Canfora, Gianpiero Esposito, Valentina Mazza, and Marcello Bruno. 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, 2007. ACM.
- [35] Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani, Roberto Codato, Massimiliano Colombo, and Elisabetta Di Nitto. Ws binder: A framework to enable dynamic binding of composite web services. In *Proceedings of the 2006 International Workshop on Service-oriented Software Engineering, SOSE '06*, pages 74–80, New York, NY, USA, 2006. ACM.

- [36] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, October 2005.
- [37] Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. Enabling proactive adaptation through just-in-time testing of conversational services. In *Proceedings of the 3rd European Conference on Towards a Service-Based Internet (ServiceWave 2010)*, volume 6481 of *LNCS*, pages 63–75. Springer, 2010.
- [38] Sebastian Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 170–, Washington, DC, USA, 2001. IEEE Computer Society.
- [39] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.
- [40] Ilenia Epifani, Carlo Ghezzi, Raffaella Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 111–121, Washington, DC, USA, 2009. IEEE Computer Society.
- [41] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 341–350, New York, NY, USA, 2011. ACM.
- [42] Marc Fisher, II, Jan Wloka, Frank Tip, Barbara G. Ryder, and Alexander Luchansky. An evaluation of change-based coverage criteria. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, PASTE '11*, pages 21–28, New York, NY, USA, 2011. ACM.

- [43] Angelo Gargantini, Marco Guarnieri, and Eros Magri. Extending coverage criteria by evaluating their robustness to code structure changes. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems*, number 7641 in Lecture Notes in Computer Science, pages 168–183. Springer Berlin Heidelberg, January 2012.
- [44] Angelo Gargantini, Marco Guarnieri, and Eros Magri. Aurora: Automatic robustness coverage analysis tool. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST '13*, pages 463–470, Washington, DC, USA, 2013. IEEE Computer Society.
- [45] Carlo Ghezzi and Sam Guinea. Run-time monitoring in service-oriented architectures. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 237–264. Springer Berlin Heidelberg, January 2007.
- [46] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.
- [47] Alberto González, Éric Piel, and Hans-Gerhard Gross. A model for the measurement of the runtime testability of component-based systems. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2009)*, pages 19–28. IEEE Computer Society, 2009.
- [48] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 72–82, New York, NY, USA, 2014. ACM.
- [49] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Evaluation of online testing for services: A case study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2010)*, pages 36–42. ACM, 2010.

- [50] Mary Jean Harrold. Testing: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72, New York, NY, USA, 2000. ACM.
- [51] Qiang He, Jun Yan, Hai Jin, and Yun Yang. Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. *IEEE Transactions on Software Engineering*, 40(2):192–215, February 2014.
- [52] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet (ServiceWave 2008)*, volume 5377 of *LNCS*, pages 122–133. Springer, 2008.
- [53] Shan-Shan Hou, Lu Zhang, Tao Xie, and Jia-Su Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, pages 257–266, 2008.
- [54] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *4th International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 40–49, 2011.
- [55] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Testing of data-centric and event-based dynamic service compositions. *Software Testing, Verification and Reliability*, 23(6):465–497, 2013.
- [56] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.
- [57] Chaitanya Kallepalli and Jeff Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, November 2001.

- [58] James Keables, Katherine Roberson, and Anneliese von Mayrhauser. Data flow analysis and its application to software maintenance. In *Proceedings of the Conference on Software Maintenance, 1988*, pages 335–347, Oct 1988.
- [59] Jong Myoung Ko, Chang Ouk Kim, and Ick-Hyun Kwon. Quality-of-service oriented web service composition algorithm and planning architecture. *Journal of Systems and Software*, 81(11):2079 – 2090, 2008.
- [60] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. Runtime testing framework for improving quality in dynamic service-based systems. In *Proceedings of the 2013 International Workshop on Quality Assurance for Service-based Applications, QASBA 2013*, pages 17–24, New York, NY, USA, 2013. ACM.
- [61] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, prediction and prevention of SLA violations in composite services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2010)*, pages 369–376. IEEE Computer Society, 2010.
- [62] Hareton K. N. Leung and Lee White. Insights into regression testing [software testing]. In *Proceedings of the Conference on Software Maintenance, 1989.*, pages 60–69, Oct 1989.
- [63] Bixin Li, Dong Qiu, Shunhui Ji, and Di Wang. Automatic test case selection and generation for regression testing of composite service based on extensible bpel flow graph. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1 –10, sept. 2010.
- [64] Bixin Li, Dong Qiu, Hareton Leung, and Di Wang. Automatic test case selection for regression testing of composite service based on extensible bpel flow graph. *Journal of Systems and Software*, 85(6):1300–1324, June 2012.
- [65] Zheng Li, M. Harman, and R.M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

- [66] Hehui Liu, Zhongjie Li, Jun Zhu, and Huafang Tan. Business process regression testing. In *Proceedings of the 5th international conference on Service-Oriented Computing, ICSOC '07*, pages 157–168, Berlin, Heidelberg, 2007. Springer-Verlag.
- [67] Christopher M. Lott and H. Dieter Rombach. Repeatable software engineering experiments for comparing defect-detection techniques. *Empirical Software Engineering*, 1(3):241–277, 1996.
- [68] Daniel Lübke, Leif Singer, and Alex Salnikow. Calculating bpm test coverage through instrumentation. In *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada, May 18-19, 2009.*, pages 115–122, 2009.
- [69] Yue Ma and Chengwen Zhang. Quick convergence of genetic algorithm for qos-driven web service selection. *Computer Networks*, 52(5):1093 – 1104, 2008.
- [70] Khaled Mahbub, George Spanoudakis, and Andrea Zisman. A monitoring approach for runtime service discovery. *Automated Software Engineering*, 18(2):117–161, June 2011.
- [71] Michele Manciacchi. Consolidated and updated state of the art report on service-based applications (CD-IA-1.1.7). Technical report, S-Cube Network of Excellence, November 2011.
- [72] John D. McGregor and David A. Sykes. *A Practical Guide to Testing Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [73] Lijun Mei, W. K. Chan, T. H. Tse, and Robert G. Merkel. Tag-based techniques for black-box test case prioritization for service testing. In *Proceedings of the 2009 Ninth International Conference on Quality Software, QSIC '09*, pages 21–30, Washington, DC, USA, 2009. IEEE Computer Society.
- [74] Lijun Mei, W. K. Chan, T. H. Tse, and Robert G. Merkel. Xml-manipulating test case prioritization for xml-manipulating services. *Journal of Systems and Software*, 84(4):603–619, April 2011.

- [75] Lijun Mei, W.K. Chan, and T.H. Tse. Data flow testing of service-oriented workflow applications. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 371–380, New York, NY, USA, 2008. ACM.
- [76] Lijun Mei, Ke Zhai, Bo Jiang, W. K. Chan, and T. H. Tse. Preemptive regression test scheduling strategies: A new testing approach to thriving on the volatile service environments. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference, COMPSAC '12*, pages 72–81, Washington, DC, USA, 2012. IEEE Computer Society.
- [77] Lijun Mei, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th international conference on World wide web, WWW '09*, pages 901–910, New York, NY, USA, 2009. ACM.
- [78] Andreas Metzger and Elisabetta Di Nitto. Addressing highly dynamic changes in service-oriented systems: Towards agile evolution and adaptation. In Xiaofeng Wang, Nour Ali, Isidro Ramos, and Richard Vidgen, editors, *Agile and Lean Service-Oriented Development: Foundations, Theory and Practice*. IGI Global, 2012.
- [79] Andreas Metzger, Osama Sammodi, and Klaus Pohl. Accurate proactive adaptation of service-oriented systems. In Javier Camara, Rogerio Lemos, Carlo Ghezzi, and Antonia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 240–265. Springer Berlin Heidelberg, 2013.
- [80] Andreas Metzger, Osama Sammodi, Klaus Pohl, and Mark Rzepka. Towards pro-active adaptation with confidence: Augmenting service monitoring with online testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010)*, pages 20–28. ACM, 2010.
- [81] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *Proceedings of the 10th*

- International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 276–290, Berlin, Heidelberg, 2007. Springer-Verlag.
- [82] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, March 1993.
- [83] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
- [84] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 636–643, July 2011.
- [85] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, December 2008.
- [86] OASIS. UDDI: Universal description, dscovery, and integration. <http://uddi.xml.org>.
- [87] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 65–69, New York, NY, USA, 2002. ACM.
- [88] Leon Osterweil. Strategic directions in software quality. *ACM Computing Surveys*, 28(4):738–750, December 1996.
- [89] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, November 2007.
- [90] Mike Papazoglou, Klaus Pohl, Michael Parkin, and Andreas Metzger, editors. *Service Research Challenges and Solutions for the Future Inter-*

net: Towards Mechanisms and Methods for Engineering, Managing, and Adapting Service-Based Systems. Springer, 2010.

- [91] Cesare Pautasso and Gustavo Alonso. Flexible binding for reusable composition of web services. In *Proceedings of the 4th International Conference on Software Composition, SC'05*, pages 151–166, Berlin, Heidelberg, 2005. Springer-Verlag.
- [92] Ajitha Rajan, Michael W. Whalen, and Mats P.E. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 161–170, New York, NY, USA, 2008. ACM.
- [93] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.
- [94] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, April 1997.
- [95] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.
- [96] Michael Ruth, Sehun Oh, Adam Loup, Brian Horton, Olin Gallet, Marcel Mata, and Shengru Tu. Towards automatic regression test selection for web services. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 729–734, 2007.
- [97] Michael Ruth and Shengru Tu. A safe regression test selection technique for web services. In *Proceedings of the Second International Conference on Internet and Web Applications and Services*, Washington, DC, USA, 2007. IEEE Computer Society.
- [98] Michael E. Ruth and Shengru Tu. Empirical studies of a decentralized regression test selection framework for web services. In *Proceedings of the*

2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, TAV-WEB '08, pages 8–14, New York, NY, USA, 2008. ACM.

- [99] Osama Sammodi, Andreas Metzger, Xavier Franch, Marc Oriol, Jordi Marco, and Klaus Pohl. Usage-based online testing for proactive adaptation of service-based applications (short). In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2011)*, pages 582–587. IEEE Computer Society, 2011.
- [100] Anja Strunk. Qos-aware service composition: A survey. In *Proceedings of the 2010 Eighth IEEE European Conference on Web Services*, ECOWS '10, pages 67–74, Washington, DC, USA, 2010. IEEE Computer Society.
- [101] Abbas Tarhini, Haccene Fouchal, and Nashat Mansour. Regression testing web services-based applications. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, AICCSA '06, pages 163–170, Washington, DC, USA, 2006. IEEE Computer Society.
- [102] Carmen Trammell. Quantifying the reliability of software: statistical testing based on a usage model. In *Proceedings of the Second IEEE International Software Engineering Standards Symposium, 1995. (ISESS'95) 'Experience and Practice'*, page 208, Washington, DC, 1995. IEEE Computer Society.
- [103] Immanuel Trummer, Boi Faltings, and Walter Binder. Multi-objective quality-driven service selection – a fully polynomial time approximation scheme. *IEEE Transactions on Software Engineering*, 40(2):167–191, February 2014.
- [104] W.T. Tsai, Xinyu Zhou, Yinong Chen, and Xiaoying Bai. On testing and evaluating service-oriented software. *Computer*, 41(8):40–46, August 2008.
- [105] Di Wang, Bixin Li, and Ju Cai. Regression testing of composite service: An xbf-g-based approach. In *IEEE Congress on Services Part II, 2008. SERVICES-2*, pages 112–119, 2008.

- [106] Chunyang Ye and Hans-Arno Jacobsen. Whitening soa testing via event exposure. *IEEE Transactions on Software Engineering*, 39(10):1444–1465, October 2013.
- [107] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, March 2012.
- [108] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.
- [109] Ke Zhai, Bo Jiang, and W. K. Chan. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *IEEE Transactions on Services Computing*, 7(1):54–67, January 2014.
- [110] Ke Zhai, Bo Jiang, W. K. Chan, and T. H. Tse. Taking advantage of service selection: A study on the testing of location-based web services through test case prioritization. In *Proceedings of the 2010 IEEE International Conference on Web Services, ICWS '10*, pages 211–218, Washington, DC, USA, 2010. IEEE Computer Society.
- [111] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.
- [112] Hong Zhu and Yufeng Zhang. Collaborative testing of web services. *IEEE Transactions on Services Computing*, 5(1):116–130, Jan 2012.