Automated Fault Localization for Service-Oriented Software Systems

Automated Fault Localization for Service-Oriented Software Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
woensdag 27 mei 2015 om 10:00 uur

Door

Cuiting CHEN

Master of Computer Science
Dresden University of Technology, Germany
geboren te Fujian, China.

This dissertation has been approved by the

promotor: Prof. dr. A. van Deursen and
copromotor: Dr. A.E. Zaidman

Composition of the doctoral committee:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. A.E. Zaidman	Delft University of Technology, copromotor
Dr. H.-G. Gross	Esslingen University of Applied Sciences, Germany

Independent members:

Prof. dr. T. Xie	University of Illinois at Urbana-Champaign, USA
Dr. M.I.A. Stoelinga	University of Twente, The Netherlands
Prof. dr. C. Witteveen	Delft University of Technology
Prof. dr. E. Visser	Delft University of Technology

This work was carried out as part of the Jacquard ScaleItUp project. This project was supported by the Netherlands Organization for Scientific Research (NWO).

Published and distributed by: Cuiting Chen
E-mail: cuiting.c.chen@gmail.com
ISBN: 978-94-6186-471-0

Copyright © 2015 by Cuiting Chen
Cover: Image ’Bookshelf Spectrum 1.0’ by Pietro Bellini.
Printed and bound in The Netherlands by CPI Wörmann Print Service.

Dedicated to my beloved mama
献给我亲爱的妈妈

// Acknowledgments

In this special occasion, I would like to express my heartfelt appreciation to all people who have contributed to the success and happiness of my Ph.D. journey.

First of all, I would like to thank my promoter and my two daily-supervisors: Arie van Deursen, Hans-Gerhard Gross and Andy Zaidman. Arie, thank you for your kindness, your support, and your consideration. I always feel lucky to have the opportunity to pursue my PhD in your research group. In particular, I would also like to thank you for offering me the postdoc position, which has significantly reduced the pressure from both career and life for me. Gerd, thank you for your advice and supervision. You pointed me the directions when I did not know how to go further. This thesis would not have been completed like this without your advice, your support and your hard work. Andy, thank you for always being there to support me. Your patience and consideration are amazing. You always consider the needs of your students and do your best to supervise them. The advices and encouragements you have given to me are not only on work, but also on life. You are also in my list of the best people I have met.

Besides my promoter and two daily-supervisors, I would like to extend my gratitude to other members of my defense committee: Prof. Dr. Tao Xie from University of Illinois at Urbana-Champaign, Dr. Marielle Stoelinga from University of Twente, Prof. Dr. Cees Witteveen from TU Delft and Prof. Dr. Eleco Visser from TU Delft. Thank you for accepting to be in my committee, reviewing the dissertation and providing valuable feedback.

I would also like to thank Prof. Dr. Serge Demeyer. I was deeply encouraged when you came to talk to me, in order to complete our conversation happened two years ago during the banquet of ICSE'12. Your attitude to students shows me why you are a great professor.

My thanks also go to the current and former members of the Software Engineering Research Group. Cor-Paul and Tiago, you both are great officemates. Thank you for all amazing chats, for all generous sharing of technology and experience, and for all positive words when I was facing some difficulties. Danielle, my first conference trip was also shared with you, thank you so much for being the local guide in Milan, and also for all Italian style of compliments. To other members of the SERG group: Alberto Bacchelli, Ali, Anja, Bas, Eric Bouwers, Felienne (for translating the propositions and the summary), Fenia (for lots of baby stuff),

Georgios, Hans, Hennie, Kees, Martin, Michaela, Moritz, Nicolas (for teaching me driving and charging my car), and Rini: thank you all for your kindness, support, consideration, and for teaching me lots of western things.

I would like to thank the supporting staffs in the software technology department. Esther, Tamara, Ilse, and Rina, thank you for considerate administrative supports. Paulo, Munire, and Stephen, thank you for excellent technical supports, especially, when rescuing my laptop after the water-pouring accident.

Many thanks go to my friends who I met in Delft. Claudia and Xin, thank you so much for being my paranymphs, and Ang-Ang's emergency contacts in daycare. Also, thank you for always helping me and always being willing to help me. Eric Piel, I really appreciate your patience to answer lots of trivial questions from me. Adele, I enjoy all hangouts and chats with you very much, you are always able to give me considerate and positive feedbacks. Ke, thank you for all help, especially, for sharing the cover design of thesis. Qiaole, thank you for inviting me (a sofa-potato) for dinners, I enjoyed them and learnt various games. Alberto and Zhutian, I am deeply touched when you told me you are coming to my defense from UK. I am looking forward to reuniting with you and seeing your lovely daughter Diana. I am very thankful to all of you. Because of you, Delft lets me feel more like home than other cities.

I also owe special thanks to Paula, thank you for helping me to learn more about myself, and helping me to get out of the most difficult situation I have ever had in my entire life.

To my family and relatives:

致我的家人和亲人们：

首先，我要感谢我的父母和弟弟。爸爸妈妈：谢谢你们无私的爱，谢谢你们辛苦的养育，严格的教导，一直不变的支持、尊重和包容。我爱你们。弟弟：小时候，我经常因为小事情对你很凶，而你总是信任我。长大后，越来越多的是我向你询问做人做事的分寸。不管相隔咫尺或天涯，我们都是一家人。

我要感谢一直关心我爱护我的整个大家族。我感谢大伯一家：你和哥哥们一直关心我的学业和生活，在我困惑的时候为我指点迷津。我感谢外公外婆舅舅阿姨和姑奶奶们，谢谢你们在我求学过程中的各种支持，照顾和帮助。外公，我相信你一定在看着我拿到学位的那一刻，并为我感到高兴。我感谢上海叔公一家的关心和帮助，每次到上海你们的热情都让我感动。叔公，我托福培训的时候你每天为我送饭的情景让我至今难以忘怀。限于篇幅，请原谅我仅用这寥寥数语来向你们大家致以我心里最诚挚的谢意。

最后，我要感谢我现在的小家庭。安安她爸，谢谢你为这个小家庭做出的所有努力。安安，你是我的人间四月天。我感谢你来到我的生命里，跟你一起的每一天都是新的，充满了希望、惊喜和期待。是你让我明白并去努力做到要珍惜当下，要充实快乐地度过每一天。我爱你！

Contents

Acknowledgements **vii**

1 Introduction **1**

1.1 Background 2
1.2 Problem Statement 4
1.3 Research Methodology 7
1.4 Contributions 8
1.5 Related Work 9
1.6 Thesis Outline 10
1.7 Origin of Chapters 10

2 Research Infrastructure **13**

2.1 Monitoring for Service-Oriented Systems 13
2.2 Assessment Vehicles 17
2.3 Summary 25

3 Spectrum-based Fault Diagnosis for Service-Oriented Systems **27**

3.1 SFL for service-oriented systems 29
3.2 Experimental Setup 32
3.3 Results and Discussion 33
3.4 Related work 37
3.5 Summary 38

4 Effects of Monitoring Topology on Spectrum Based Diagnosis **41**
4.1 Topology Effects 42
4.2 GA for Topology Optimization 43
4.3 Experiments 44
4.4 Discussion 52
4.5 Related Work 54
4.6 Summary 55

5 Diagnosis Improvement Through Increased Monitoring Granularity **57**
5.1 Background 59
5.2 Problem Statement and Approach 62
5.3 System Simulations 65
5.4 Case Study 67
5.5 Runtime Overhead 70
5.6 Discussion and Lessons Learned 77
5.7 Related work 80
5.8 Summary 82

6 Diagnosis Improvement Through Invocation Monitoring **85**
6.1 Problem Statement and Approach 87
6.2 System Simulations 89
6.3 Case Study 94
6.4 Discussion 96
6.5 Related work 97
6.6 Summary 98

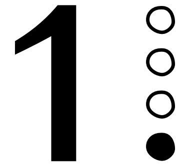
7 Conclusion **99**
7.1 Summary of Contributions 99
7.2 The Research Questions Revisited 100
7.3 Recommendations for Future Work 103

Bibliography **105**

Summary **115**

Samenvatting **117**

Curriculum Vitae **119**



1 Introduction

Service-oriented computing (Papazoglou et al., 2007) has lately attracted huge attention by the software industry. A service-oriented software system uses loosely-coupled and self-contained services as system components and it can adapt to fast-changing business requirements. For example, with the support of dynamic discovery and Service Level Agreement (SLA) management, a service can be deployed into the system at operation time. This enables a service-oriented system to reconfigure to assemble new functionality, in order to meet the changed requirements. Services can also be updated or removed from the system, which promotes the dynamic nature of a service-oriented system. In addition, a service is usually running at its provider's infrastructure. Thus, the service-oriented system that integrates the service may only use it, but does not own it (Turner et al., 2003).

The two main features of service-oriented systems, i.e., loosely-coupling and highly-dynamic, can bring challenges to their quality assurance (Allauddin et al., 2011). Specifically, checking the interaction between services and the integration of services is difficult. This is mainly attributable to their loose coupling, late (runtime) binding, and deployment in many application contexts. Runtime testing (Brenner et al., 2006), which aims to test systems at operational time, can alleviate the detection of failures in highly dynamic systems. However, it requires specific architectures that support it (Gonzalez-Sanchez et al., 2010b), and side-effects need to be eliminated, for example, interference with normal system operation (Greiler et al., 2009).

In order to meet quality of service requirements, service-oriented systems should be able to recover from failures when they are detected. Its inherent features enable it to reconfigure itself by replacing the faulty service with a healthy one at runtime. In addition, to further prevent future problems, the developers of faulty services can also be informed to fix the fault. Therefore, localizing the real faulty service becomes a critical task for the quality assurance of a service-oriented system. Particularly, when a failure is detected in the system, the service where the error is observed may not be causing it, since a fault can be propagated from the previously activated services (Novotny et al., 2012). Thus, besides the need for monitoring facilities integrated into the service-oriented system and its underlying platform (Baresi et al., 2004a), we also need a fault localization technique to

identify the actual faulty service in the system.

Due to the dynamic nature of a service-oriented system, the required fault localization should be applied to a running service-oriented system at operation time. Although the additional fault localization always comes with a performance cost, the disturbance to regular performance of the diagnosed system should be reduced as much as possible. Among existing fault localization approaches, two categories can be distinguished: model-based approaches and spectrum-based approaches (Abreu et al., 2009a).

Model-based techniques calculate the diagnosis by comparing the model, which describes expected behavior deduced from the prior knowledge of a system, with actually observed behavior of the system. Spectrum-based fault localization (SFL) techniques only use the dynamic component coverage information and the pass/-fail results to generate a rank of likely faulty components.

Since the model-based approaches build models with prior system information, such as component dependencies, their diagnosis performance is generally more precise when compared to SFL. However, the system modeling is usually based on the static information of the system, and the model-based approaches would take much more calculation effort for diagnosis (Mayer and Stumptner, 2008).

In contrast to the model-based approaches, the SFL approaches are more light-weight (Zoeteweij et al., 2007b). They do not require prior static system information or modeling of the system, and only take the dynamic trace information as source for diagnosis. Therefore, the spectrum-based fault localization techniques can better meet the requirements of online diagnosis for service-oriented systems. The main contribution of this thesis is to apply SFL to service-oriented systems to pinpoint the actual faulty service automatically.

1.1 Background

Service-Oriented Software Systems

A service-oriented software system is a software system developed following the principles of Service-Oriented Architecture (SOA), which is a style of system architecture designed to support fast-changing business requirements (Josuttis, 2007a).

The basic component of SOA is a service, which is a self-contained unit representing business functionality. Generally, a service is designed to be stateless, thus, users of services do not depend on the state of service providers. A service also has explicitly defined interfaces, which are described in the Web Service Description Language (WSDL). A service can be invoked through its published interfaces over a network connection. The service is governed by a set of service contracts, i.e., Service-Level Agreements (SLAs). The messages exchanged between services are typically defined by XML Schema. Furthermore, a service is loosely-coupled and users of a service can only see the public interfaces of the service, while the

location, the implementation technology or the current status of the service are invisible to users.

Based on the above design concepts, the communication between services is guaranteed through open standards, such as WSDL, XML Schema and SLAs, which are independent from technologies required by services. Therefore, services implemented with different languages or running on various hardware platforms or operating systems, can interact with each other.

Besides the open standards for communication, SOA also provides some techniques, such as service registry and enterprise service bus (ESB), to enable dynamic search, dynamic localization, or dynamic routing for services. Hence, a service can be deployed into a service-oriented system any time during operation. It can also be updated or removed at runtime. This means that interactions between services can become highly dynamic.

Because of the loosely-coupled and highly-dynamic features, a service-oriented system is able to adapt to fast-changing business requirements quickly by combining new services or updating existing services in the system dynamically.

Spectrum-based Fault Localization

SFL is a statistics-based technique that automatically infers a diagnosis from symptoms. The *diagnosis* is a ranking of potentially faulty components (block, source code line, etc.) in a system, with the most likely faulty one ranked top. The *symptoms* are observations about component involvement in a system execution, plus pass/fail information about that execution (Gonzalez-Sanchez et al., 2011). Component involvement is expressed in terms of *block-hit-spectra* (hence its name), producing for each execution a binary coverage value per component (Reps et al., 1997)(Zoeteweij et al., 2007c). Further, each system execution (test), is associated with a binary verdict (pass=0, fail=1) from an oracle. Several tests lead to an activity matrix, representing coverage of each component over time. The binary verdicts lead to an *output vector*. The diagnosis is calculated through applying a similarity coefficient (SC) to each component activation vector and the outcome vector. The similarity denotes the likelihood of a component being the faulty one, and, therefore, determines its position in the ranking. Any SC may be used; however, the Ochiai SC has been found to work best (Abreu et al., 2006). This technique mimics how a human diagnostician would infer a diagnosis from observing which parts of a system were involved in producing a failure.

SFL is illustrated in Table 1.1 by means of a Java program. This example is comprised of components $C_0 - C_{10}$ with a source code line as component granularity. It is exercised with six tests/transactions, leading to the component activation for each transaction $t_1 - t_6$ noted down in the activity matrix. Four transactions have failing test outcomes (1); two have passing test outcomes (0), noted in the output vector. The Ochiai SC is calculated for the output vector and each component activation vector. Then, the similarity values are brought in a descending order. This

Table 1.1: Illustration of SFL

Component	Character counter	t_1	t_2	t_3	t_4	t_5	t_6	SC_o
	public int count(String s){	[Activity Matrix]						
C_0	int upper = 0 ; int lower = 0; int digit = 0; int other = 0;	1	1	1	1	1	1	0.82
C_1	for(int i = 0; i<s.length(); i++){	1	1	1	1	1	1	0.82
C_2	char c = s.charAt(i);	1	1	1	1	1	1	0.82
C_3	if(c >= 'A' && c <= 'Z')	1	1	1	1	0	1	0.89
C_4	**upper += 2;**	1	1	1	1	0	0	1.00
C_5	else if(c >= 'a' && c <= 'z')	1	1	1	1	0	1	0.89
C_6	lower++;	1	1	0	0	0	0	0.71
C_7	else if(Character.isDigit(c))	1	0	1	0	0	1	0.58
C_8	digit++; }	1	0	1	0	0	1	0.58
C_9	other = s.length()-upper-lower-digit;	1	1	1	1	1	1	0.82
C_{10}	return other;	1	1	1	1	1	1	0.82
	}							
	Output vector (verdicts)	1	1	1	1	0	0	

results in component 4 being ranked top with 100% likelihood, which represents the location of the fault in this example system.

Online Monitoring vs. Online Testing

Instead of performing proactive online testing, we favor passive online monitoring plus online diagnosis for identifying residual defects. Monitoring is less intrusive, requires fewer assumptions about the system, and is easier to incorporate into a service-oriented system. Many modern service platforms such as Apache's Axis2[1], Redhat's JBoss[2], or eBay's Turmeric[3] come equipped with extensive monitoring/profiling frameworks that can be adapted to diagnosis. Our case system makes use of Turmeric's monitoring functionality.

The disadvantage of monitoring is that errors hiding in seldom-used parts of a service-oriented system are difficult to trigger by normal execution. They would have to be triggered on purpose through specific tests. Furthermore, such errors are unlikely to be identified. However, this is not an urgent issue, since only those services or parts thereof which are actually used in a particular application, will be exercised and monitored.

1.2 Problem Statement

The main research problem addressed in this dissertation is to diagnose service-oriented systems automatically at runtime. Due to the dynamic nature of service-oriented systems, the traditional offline techniques for fault tolerance become inefficient. For example, since services can be deployed at runtime, some faults, such as execution faults, are more likely to cause failures during runtime (Bruning et al.,

[1] http://axis.apache.org
[2] http://www.redhat.com/products/jbossenterprisemiddleware/
[3] https://github.com/ebayopensource/turmeric-runtime

2007). When a failure is detected, it is necessary to trace the failure back to its actual cause. In the meanwhile, the fault localization should not bring too much disturbance to the system's runtime performance. Therefore, in this thesis, we propose to apply spectrum-based fault localization (SFL), which is a statistics-based and light-weight diagnosis technique, to service-oriented systems.

For the purpose of disturbing system operations as little as possible, the online diagnosis should only be activated when a failure happens. Hence, as the first step of applying SFL to service-oriented systems, we focus on the following research question:

> ***RQ3.1****: How can a failure be detected in an operational service-oriented system?*

In Chapter 3, we will illustrate an online monitoring technique for service-oriented systems to obtain data on their dynamic behavior and actual usage. Based on the monitoring data, a failure can be detected at runtime for service-oriented systems.

Since SFL requires an activity matrix and an output vector to calculate diagnosis, it is necessary to monitor the service system at runtime and obtain the information of service transactions containing service involvement and the results of all transactions to form an activity matrix and an output vector. In Chapter 3, we present how we adapt the concepts of SFL into the service-context, in order to apply SFL to a service-oriented system:

> ***RQ3.2****: How can spectrum-based fault localization be applied in a service-oriented system in order to trace a failure back to its respective root cause?*

To evaluate how many correct diagnoses SFL is able to perform in a service-oriented system, we will conduct experiments in Chapter 3 to answer the following research question:

> ***RQ3.3****: How well does spectrum-based fault localization perform in a service-oriented system in terms of correctness of the diagnosis?*

SFL is a diagnosis technique based on the information of component coverage during system transactions. In a service-oriented system, a monitoring technique is used to collect the activity information for components. We refer to the placement of monitors as the monitoring topology of the diagnosed system. Since monitors can be put anywhere in a service-oriented system, it is better to put the monitors in those places which can facilitate the diagnosis. This represents an optimization problem to create an optimal monitoring topology for diagnosing a service-oriented system. Among existing meta-optimization heuristics, we prefer genetic algorithms, because they are adequate for our problem domain and easy to apply.

With this understanding, we focus on the following research question in Chapter 4:

> ***RQ4.1***: *How can genetic algorithms be used to optimize monitoring topologies for spectrum-based diagnosis?*

Knowing the characteristics of optimal monitoring topologies for diagnosis can guide us to place monitors in the most suitable positions. This can facilitate the application of SFL on a service-oriented system by improving the accuracy of diagnosis without adding needless overhead. Thus, we will answer the following research question in Chapter 4:

> ***RQ4.2***: *What are characteristics of monitoring topologies that are optimal for spectrum-based diagnosis?*

The phenomenon of tight interaction between services can cause suboptimal diagnoses. Tight interaction means several services are always invoked together in transactions. If two services are always invoked together and one of them is faulty, the diagnosis would be such that both services will be convicted, leading to incorrect or inconclusive diagnoses. Such diagnoses would bring extra unnecessary inspection cost for the system, in order to find the real faulty service. Since changing the architecture of a service-oriented system to eliminate tight interactions is not realistic, an optional solution to diagnose a service-oriented system with tight interaction correctly is to change the monitoring granularity of a service-oriented system. Based on this understanding, we focus on the following research question in Chapter 5:

> ***RQ5.1***: *How and to which extent does the monitoring granularity affect the calculation of a diagnosis with spectrum-based fault localization?*

When applying SFL to a service-oriented system, services or service operations can be taken as the diagnosis components, i.e., the monitoring granularity is on the level of service or service operation. Unfortunately, the granularity of service or service operation is very coarse. This brings the following challenge:

> ***RQ5.2***: *How can we increase the monitoring granularity?*

In Chapter 5, we will increase the monitoring granularity by injecting monitors into the service implementation. This solution has a limitation in that it requires the ownership of a service. Therefore, in Chapter 6, we propose to include other contextual information of service transactions into the diagnosis:

> ***RQ6.1***: *To which extent can the usage of information expressing activation of links between services improve diagnosis?*

We will add the link invocation activation information into the diagnosis in Chapter 6. We will perform simulations and a case study to explore the effect of topology information on diagnosis, and we will aim to find the characteristics of a topology which can improve diagnosis:

> ***RQ6.2***: *How does topology, i.e. the organization of the invocation links between services, affect diagnosis, and are there general characteristics of topology that improve diagnosis?*

While the diagnosis can facilitate the fault tolerance of a service-oriented system, it also brings a performance penalty to the running service-oriented system. In order to analyze the trade-off between the diagnosis and its overhead, it is necessary to measure the runtime overhead caused by the diagnosis for a service-oriented system. Since the diagnosis engine is detached from the service-oriented system in our approach, the overhead of diagnosis is mainly from the monitoring in the system. To assess the overhead of monitoring at different levels of granularity, we focus on the following research question in Chapter 5:

> ***RQ5.3***: *What is the overhead caused by the monitoring of various levels of granularity?*

1.3 Research Methodology

To answer our research questions, we use different research methods. Research questions 3.1, 3.2 and 3.3, which aim at applying SFL to diagnose service-oriented system, are studied following the case study research method (Yin, 2014).

We conduct a case study on a real service-oriented system, i.e., SFL-Stonehenge[4], to answer the research questions. The main reasons for choosing the SFL-Stonehenge system are: (1) the system is a real world example extended from Apache Stonehenge[5] and built on eBay's Turmeric SOA platform[6]; (2) the SFL-Stonehenge system is open-source and can be used as a benchmark, so other researchers can compare our approaches with theirs and verify our findings.

Research questions 4.1 and 4.2 aim to find the effects of the monitoring topology on SFL diagnosis for service-oriented systems. To answer these questions, we set up a large number of experiments with the SFL simulator[7] developed at Delft

[4]https://github.com/SERG-Delft/sfl-stonehenge
[5]https://cwiki.apache.org/STONEHENGE/
[6]https://github.com/ebayopensource/turmeric-runtime
[7]https://github.com/SERG-Delft/sfl-simulator

University of Technology to validate our approach. The main reason for using the simulator is because it allows us to set up various complex monitoring topologies quickly and easily, which is infeasible for a real service-oriented system.

For research questions 5.1, 5.2, 6.1 and 6.2, which aim to improve diagnosis for service-oriented system with tight interactions, we follow a mixed-methods approach (Creswell and Clark, 2010), i.e., the combination of experiment and case study. Firstly, we conduct experiments with the SFL simulator for a trial of our approaches. After receiving results from the simulations, we implement our approaches in the case system and perform case studies to obtain a deeper understanding of our findings. Research question 5.3 is about measuring monitoring overhead when applying SFL to service-oriented systems, and is also answered by performing a case study.

1.4 Contributions

The contributions of this thesis are as follows:

A case study system for research related to service-oriented systems (Chapter 2). We perform a small literature survey about case study systems being used in service-oriented research, and find that the service-oriented community is lacking a standard case study. Therefore, we propose the SFL-Stonehenge, an open-source service-based Java software system, as a possible standard case study for researchers working in the area of service-oriented systems.

A simulator for diagnosing a service-oriented system with spectrum-based fault localization (Chapter 2). We introduce a novel simulator for spectrum-based fault localization, which is used to study the effects of changing the observation granularity on the calculation of the diagnosis in many different system configurations. The simulator can be used for assessing different system topologies quickly and easily.

Application of spectrum-based fault localization to service-oriented software systems (Chapter 3). We discuss the requirements of applying spectrum-based fault localization to service-oriented systems and show how such requirements can be realized in a concrete service platform. We demonstrate the application of online spectrum-based fault localization in a real service-oriented system and evaluate to which extent online spectrum-based fault localization can pinpoint faulty service operations automatically.

Formulation of general characteristics of optimal monitoring topologies for spectrum-based fault localization (Chapter 4). We apply genetic algorithms to study the optimality of monitoring topologies through spectrum-based fault localization. We define a set of fitness functions for the application, and derive a set of general characteristics of topologies that improve spectrum-based diagnoses.

Analysis of service diagnosis improvement through increased monitoring granularity (Chapter 5). We describe an approach and implementation for in-

creasing the monitoring granularity in services, and show how this can improve the accuracy of diagnosing faulty services. We use a simulator to study the effects of changing the monitoring granularity on the calculation of the diagnosis in many different system configurations. We also evaluate our approach and implementation in a case study and discuss its implications.

Assessment of monitoring overhead for service-oriented system (Chapter 5). We implement various levels of monitoring required for spectrum-based fault localization in a service-oriented system. We measure their runtime overhead on a running system at different levels, and compare the monitoring overhead at the service communication level and at the service implementation level.

Improving Service Diagnosis Through Invocation Monitoring (Chapter 6). We propose to extend monitoring to include the invocation links between the services, and derive an algorithm to incorporate link invocation information in spectrum-based fault localization. We show through simulations and a case study with a real system under which circumstances service monitoring alone inhibits the correct detection of a faulty service, and how and to which extent the inclusion of invocation monitoring can lead to improved service diagnosis.

1.5 Related Work

There are some studies relevant to diagnosis for service- oriented software systems. Chen et al. present *Pinpoint* (Chen et al., 2002), a similar diagnosis approach plus a tool using similarity coefficients in order to infer a diagnosis from system activation and component involvement. However, even though their title suggests otherwise, they do not address the specific issues of diagnosing services, i.e. the problems of inter-service diagnosis, and the fact that services are used in different contexts.

Yan, et al. (Yan and Dague, 2007; Yan et al., 2009), propose a model-based approach to diagnose orchestrated Web service processes. Modeling is done through discrete event systems, which imposes a heavy burden on the user of the technique. Zhang et al. (Zhang et al., 2009, 2012a) describe approaches for diagnosing quality-of-service problems in service-oriented architectures. However, their diagnosis approaches cannot adapt well to the dynamic nature of SOA, due to the static information they used. Moreover, their bayesian-based approaches are more heavyweight compared to spectrum-based approaches. Additionally, the authors measure the execution time for diagnosis, but their main purpose is to compare the performance of their two approaches, and they did not assess the overhead caused by diagnosis to the performance of service system. Mayer and colleagues (Mayer et al., 2010a, 2012), describe a similar diagnosis approach that is based on analyzing execution traces of failed transactions. However, the models they used for diagnosis are rather complex, and proper evaluation is still pending.

1.6 Thesis Outline

The outline of this thesis is as follows: Chapter 2 introduces the basic techniques used in this thesis to support our research. Chapter 3 presents the application of SFL for service-oriented systems. Chapter 4 introduces how the monitoring topology can affect the spectrum-based diagnosis for service-oriented systems. Chapter 5 contains an approach to increase monitoring granularity to solve the failed diagnosis when SFL is applied to diagnose a service-oriented system with tight interactions. In addition, this chapter also presents the runtime overhead caused by the required monitoring techniques for service-oriented systems. Chapter 6 describes another approach to solve the tight interaction problem for SFL diagnosis by including the monitoring of invocation links between services.

1.7 Origin of Chapters

Each of the chapters in this thesis has been published in at least one peer-reviewed publication. Most of the publications have been co-authored with Hans-Gerhard Gross and Andy Zaidman. The following list gives an overview of these publications:

Chapter 2 contains materials published at the *1st International Workshop on Quality Assurance for Service-Based Applications (QASBA'11)* (Chen et al., 2011), the *16th European Conference on Software Maintenance and Reengineering (CSMR'12)* (Espinha et al., 2012a), and the *7th International Conference on Software Security and Reliability (SERE'13)* (Chen et al., 2013a).

Chapter 3 has been published in the proceedings of the *5th International Conference on Service-Oriented Computing and Applications (SOCA'12)* (Chen et al., 2012).

Chapter 4 contains our work appeared in the proceedings of the *24th International Workshop on the Principles of Diagnosis (DX'13)* (Chen et al., 2013c).

Chapter 5 comprises our findings submitted to the *Software Quality Journal*. This chapter is an extension of our previous work published at the *7th International Conference on Software Security and Reliability (SERE'13) with the most distinguished paper award* (Chen et al., 2013a). The previous article is focused on the improvement of the diagnosis through increasing the monitoring granularity with a preliminary overhead assessment. The main extension of this chapter is the addition of a detailed analysis of runtime overhead caused by the different levels of monitoring.

Chapter 6 contains material published at the *13th International Conference on Quality Software (QSIC'13)* (Chen et al., 2013b).

Additional Publications

The author has been involved in the following publications which are not directly included in this thesis:

- *Comparing Diagnostic Performance of Ochiai and Relief in Service-oriented Systems*, which appeared in the proceedings of the *24th International Workshop on the Principles of Diagnosis (DX'13)*. This paper is refereced as (Chen et al., 2013d).
- *Spicy Stonehenge: Proposing a SOA Case Study*, which appeared in the *4th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS'12)*. This paper is referenced as (Espinha et al., 2012b).

2 Research Infrastructure

This chapter presents the grounding techniques required by the research in the following chapters. (1) The highly dynamic and loosely coupled nature of a service-oriented system leads to the challenge of understanding and maintaining it. To obtain insight into the runtime topology of a SOA system, we propose a framework-based runtime monitoring approach to trace the service interactions during execution. The approach can be transparently applied to all web services built on the framework and reuses parts of information and functionality already available in the framework to achieve our goals. (2) Maintenance research in the context of Service Oriented Architecture (SOA) is currently lacking a suitable standard case study that can be used by scientists in order to develop and assess their research ideas, and for comparison, and benchmarking purposes. For this reason, we built upon an existing open-source system and make it available for other researchers to use. This system is SFL-Stonehenge. (3) Performing experiments with a fully fledged case study is tedious. Every new experiment requires extensive adaptation to new experimental requirements. This lead us to develop a simulator for applying spectrum-based fault localization to service-oriented systems.[1]

2.1 Monitoring for Service-Oriented Systems **13**

2.2 Assessment Vehicles **17**

2.3 Summary **25**

2.1 Monitoring for Service-Oriented Systems

Today, many organizations deploy services for realizing their landscapes of information technology. They aim at exploiting the ability of service technologies to integrate existing legacy components, and to better cope with changing business requirements. These are two core demands of industry which are addressed adequately through highly dynamic and loosely coupled service-oriented architec-

[1]This chapter is based on our papers published in the*International Workshop on Quality Assurance for Service-Based Applications (QASBA'11)* (Chen et al., 2011), the *16th European Conference on Software Maintenance and Reengineering (CSMR'12)* (Espinha et al., 2012a) and the *7th International Conference on Software Security and Reliability (SERE'13)* (Chen et al., 2013a).

tures (SOA) (Papazoglou et al., 2006). In particular, services can be discovered, bound, and executed during operation time, enabling (online) evolution (Gold et al., 2004).

However, loose coupling and the highly dynamic nature of service-based software systems also present challenges in the maintenance and evolution processes. For example, the actual configuration of a system realized with services, and the usage of its parts, can only be seen at runtime (Canfora and Di Penta, 2009a). Although online maintenance and evolution is technically well supported, system comprehension, a key prerequisite for conducting maintenance and evolution (Zaidman et al., 2010), is not (Gold et al., 2004). Understanding complex SOA in order to plan and implement maintenance and evolution, is still one of the major challenges for software engineers (Moe and Carr, 2001).

Information that can be derived statically is not enough for understanding and visualizing how a SOA is deployed at runtime, and how the services interact in order to realize the business goals of various users. Instead, or in addition, runtime monitoring should be employed as the primary means to obtain data on the dynamic behavior of a SOA and its usage. In this way, software engineers can get a better understanding of the service-based software system and, consequently, they can plan and perform necessary system maintenance and evolution activities more adequately and timely. By also adding the usage information of individual services to the extracted views, the engineers can better plan maintenance, thus reducing the disturbances to the nominal system operation of an entire IT infrastructure. Moreover, online monitoring can facilitate SOA governance through supporting load balancing, identifying performance bottlenecks, or usage profiling.

In this section, it is our goal to support software engineers by creating high-level views of how services (dynamically) interact, i.e., the *runtime topology*. In order to realize this, we first identify the associated monitoring data, will subsequently help engineers in the (online) maintenance and evolution of service oriented architectures. Furthermore, in order to acquire the required information, we propose to extend existing service frameworks to support monitoring, and to be able to exploit information readily available inside these frameworks. For example, the addressing information used to send and receive requests and responses can be extracted to reversely reason about the invocation sequences and activities of users. That way, engineers can update and maintain parts of the SOA with low current usage, or they can defer maintenance to periods with expected low usage, thereby minimizing disturbance. Some of the data required for planning and realizing such maintenance activities are already provided by SOA frameworks through logging mechanisms contained in many platforms. Moreover, additional mechanisms can be integrated into frameworks, in order to provide required data according to various comprehension goals. For example, a framework can be extended to add a *sequence id* to SOAP messages, which provides the order of messages caused by an invocation traversing different machines.

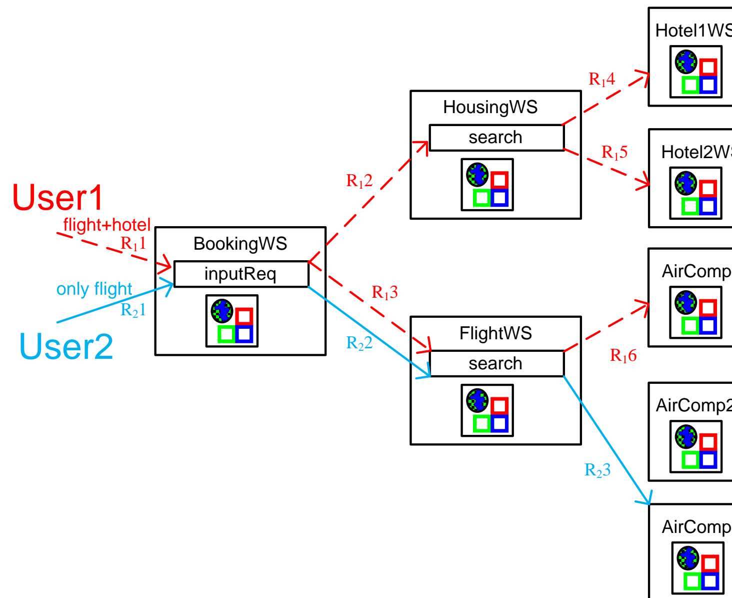


Figure 2.1: SOA system Scenario

Approach

Goal Our basic goal is to support the comprehension process of complex SOA systems. In particular, we are interested in understanding the runtime topology of services, which entails obtaining insight into how services work together to execute a particular functionality. Recovering this type of information calls for a dynamic analysis approach, which means monitoring the SOA system during runtime. An additional benefit of adopting a dynamic analysis approach is that we are able to follow an *on-demand* comprehension strategy, i.e., we only deal with information relevant to the execution scenario and to the part that we want to understand (Cornelissen et al., 2009). In order to accomplish this dynamic analysis, we aim to integrate monitoring techniques into web service frameworks, as to leverage all available information inside the framework for monitoring. This approach (1) can be transparently applied to all web services built on the framework, and (2) parts of information and functionality already existing in the framework can be reused to achieve our goals.

Figure 2.1 presents the runtime scenario of a service-based system: customers with different inputs invoke different sets of services. In particular, depending on whether a customer requires a flight or a flight/hotel combination, a different set of services is invoked. In order to reconstruct the runtime topology of a SOA system, i.e., how services interact at runtime, we at least need to obtain the following data from the web service framework:

service id: before we deduce the interactions between services, we should first be able to identify the services involved. The service name is not enough, as there may be different services sharing the same name. Moreover, those

services can be described in the same service description file with different *target namespaces*. Therefore, in order to uniquely identify a service, we propose a simple scheme using the combination of the URI of the service description file, the target namespace and the service name as the service id.

interface id: to further know which function a user is invoking in a service, it is also necessary to log the name of the invoked service interface. However, a service can contain two operations, i.e., interfaces, with the same name and different parameters. For example, earlier versions of WSDL, one of the most common web service description languages, support operation overloading. Hence, the information of parameters is also required to distinguish an operation.

process id: in order to trace an invocation traversing a set of web services, a specific identifier named *process id* is needed to link all requests and responses involved.

sequence id: as service-based systems are frequently deployed in a distributed context, using only time stamps to deduce the invocation sequence of the services might be problematic, since physical clocks in various machines may deviate from each other. This problem can be mitigated by using either a *logical clock* (Lamport, 1978) or *vector clock* (Fidge, 1988; Mattern, 1989), or by a simple mechanism, which involves adding a *sequence id* to the message being sent to the next service. For each request that comes into the SOA system, a new sequence id counter is created and each time this request causes a new message to be sent to another service, the sequence id is incremented.

SOA frameworks Generally, a web service consists of three major parts: a listener, a proxy, and the service implementation (Snell et al., 2001). When a web service is created based on a service framework, typically the service developers only need to implement the core business logic in the service implementation, and the other two parts are realized in the framework. The listener detects incoming and outgoing messages passing through the server and the proxy deals with messaging and addressing.

Once the listener receives an incoming request, it will forward the message to the proxy, which will parse the request to obtain the information of the invoking service and interface. Then the content of the request is delivered to the target service. After the service sends back the response, the proxy will decode the response and the listener will dispatch the result to the invoker. In order to execute the invocation properly, the service framework stores the addressing information to dispatch requests and responses at runtime. In addition, each service creates a specific instance of itself for each invocation, and an object id is assigned to the instance for the aim of identification.

Hence, some information required to rebuild the runtime topology of the system, such as the information for the service id and the interface id, is already in-

side the framework. However, obtaining the other information elements from the framework requires extra work. Generally, a framework does not offer a process id for each message. Thus, we can either extend the framework to enable the new id generation, or reuse the existing ids inside the framework. For example, it is feasible to reuse the object id of the service firstly invoked in a sequence as a process id (the mechanism to determine the first invoked service will be considered in future work). The framework keeps passing the id to all following messages involved in the same activity. After logging the information, we can identify all messages containing the same process id as belonging to the same invocation. For the sequence id, however, a particular mechanism should be added into the framework to guarantee its delivery and incrementation.

Service frameworks typically have a logging system in place to track abnormal behavior that might arise. It is our aim to reuse these monitoring mechanisms and extend them for our purposes.

2.2 Assessment Vehicles

Stonehenge System

While the actual term Service-Oriented Architecture (or SOA) was first coined in the mid 1990's by Gartner (Natis, 2003; Josuttis, 2007b), the ideas behind it, namely building software systems that are composed out of loosely coupled, interoperable components or services, goes back further. It was, however, the technology of *web services*, launched in 2000 as a set of standards to allow computers to communicate with each other (Josuttis, 2007b), that acted as a catalyst for both industry and academia to really start investigating the possibilities of Service-Oriented Architectures. In particular, SOAs promise to (1) allow businesses to be more flexible as business needs change and (2) ease evolution due to the loosely coupled nature of the system (Gold et al., 2004).

When looking at the past decade of research in service-orientation, we can observe that although a lot of fruitful research has been carried out (e.g., see (Benatallah and Motahari Nezhad, 2008; Benbernou et al., 2008)), many of the research efforts are isolated in nature. While this isolation is not bad per se, it does hinder progress. Symptomatic of the isolated nature of research in this area is the absence of a common case study that can be used as a benchmark. Indeed, Sim et al. report that benchmarking, when embraced by a community, has a strong positive effect on the scientific maturity of a discipline (Sim et al., 2003). In particular, it allows to easily compare solutions and to perform replication studies. In many fields of software engineering, researchers have resorted to benchmarking in order to compare approaches and ultimately advance the field. Prime examples being the aspect-mining community that settled on *JHotDraw* as a standard case study (Ceccato et al., 2006), or the refactoring community that introduced the *LAN simulation* (Demeyer et al., 2002).

In order to unify the SOA community around a single case study, we propose a system that is at the same time realistic, easy to understand and which most researchers should be able to use as a "standard case study system". The system we propose — SFL-Stonehenge[2]— is based on Apache Stonehenge[3] and consists of an application composed out of several web services. The open-source nature of SFL-Stonehenge and its availability should stimulate researchers in the area of SOA, that normally resort to small examples built specifically for the context of their research, to choose for SFL-Stonehenge, thus enabling the benchmarking process that the community needs.

Background Research

Table 2.1: Selected SOA research papers with case studies

Paper	Complexity	Impl. Tech.	Availability
(Barbon et al., 2006)	3 web services	Unknown	No
(Pistore and Traverso, 2007)	1 web service	Unknown	No
(Domenico and Carlo, 2007)	1 web service with 3 interfaces	Unknown	No
(Heward et al., 2010a)	1 web service	Unknown	No
(Marconi and Pistore, 2009)	2 services	Unknown	Industry, API avail.
(Ardissono et al., 2006)	3 web services	Unknown	No
(Baresi et al., 2004b)	Unknown	Unknown	No
(Mahbub and Spanoudakis, 2005)	Unknown	Unknown	No
(Momm et al., 2007)	Unknown, KIM[4] project	Unknown	No
(Nasr et al., 2011)	700+ services	J2EE, IBM WebSphere etc.	Industry case
(Ahmad and Pahl, 2011)	Unknown	Unknown	No
(Schmerl et al., 2011)	120+ services	Apache CXF etc.	No
(Bertolino et al., 2009)	1 service: AECS[5]	Unknown	Industry, API avail.
(Denaro et al., 2009)	del.icio.us[6] and OpenSocial[7]	Unknown	Industry, API avail.

In our reconnaissance of the research area of Service-Oriented Architectures, we noticed that there is no standard case study being used by researchers. Furthermore, during our exploration of the field we also got the impression that a wide variety of small and/or closed source systems were being used as case studies for evaluating the research. In order to get a better feeling of how research in the area of SOA is conducted, we have performed a small literature survey where we specifically focused on the software systems that are being used in case study research.

In order to characterize the case study systems being used in SOA research, we compiled Table 2.1, which represents a small subset of research papers in the area of SOA. The papers that we selected for this overview originate from:

[2]https://github.com/SERG-Delft/sfl-stonehenge
[3]https://cwiki.apache.org/STONEHENGE/
[4]http://kim.cio.kit.edu/
[5]http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
[6]http://delicious.com/help/api
[7]http://code.google.com/apis/opensocial/docs/0.7/reference

- The state-of-the-art report on service monitoring from the European S-Cube[8] project on software services (Benbernou et al., 2008). We selected this survey because our research goals are aligned with many of the papers mentioned in this report.

- A selection of recent papers published at venues like CSMR, ICSE and ESEC/FSE, from which we expect a thorough validation.

The 14 papers listed in Table 2.1 are all representatives of research following the case study methodology (Wohlin et al., 2000). We now list some of our observations:

Self-created case study systems. From this selection of papers we noticed that some authors created their own simple non-industrial examples as case systems, which contain a very small number of services, e.g., (Domenico and Carlo, 2007) and (Ardissono et al., 2006) have one and three services respectively. It is arguable whether these small case study systems are representative of real service-based software systems. Some self-created systems also appear more complex. For example, Baresi et al. (Baresi et al., 2004b) describe an IT certification system which gives enrolled students a chance to try a certification test for free. However, the paper only describes the conceptual details of the system.

An important issue with self-created systems is that their set-up might be favoring the researched technology, which becomes extra hard to verify when these self-created systems are not publicly available. Looking at Table 2.1 we see that unfortunately, almost all systems are not publicly available.

Closed-source systems. Some researchers are cooperating with industry and have the chance to get a real-world system as their research vehicle. For example, Momm et al. (Momm et al., 2007) apply their approach to a practical scenario developed in a project aiming to redesign a university's business process; Nasr et al. (Nasr et al., 2011) provide an industry case study supported by a business service IT company. Also, in the paper by Pistore et al. (Pistore and Traverso, 2007), the authors mention that their approach was applied to some real applications, but no more details are provided.

Industrial case studies are extremely important in software engineering research, however, due to the closed-source nature of these software systems they cannot be obtained by other researchers. This means their results cannot be reproduced or compared, which strengthens our call for a common case study to compare techniques on.

Implementation technology. During this survey, we also focus on investigating the implementation technologies used in those case study systems, such as the

[8]http://www.s-cube-network.eu

programming language, the underlying frameworks, the communication protocols, etc. These pieces of information are necessary in different situations, e.g., (1) when practitioners want to use the experimental results and want to verify whether the results are applicable in specific circumstances or (2) when researchers want to replicate a study or perform a meta-analysis (Kitchenham et al., 2002).

However, as Table 2.1 shows, most papers do not provide implementation details. The notable exceptions are paper (Schmerl et al., 2011) and paper (Nasr et al., 2011), which clearly mention that their systems are built on the Apache CXF framework and the IBM WebSphere respectively. In the case of industrial case studies, sometimes the APIs are open, but the implementation techniques are kept confidential.

Summary and recommendations. The small survey that we present in this section makes it clear that comparative studies or replications are difficult to perform considering the fact that many (implementation) details are not presented in the papers considered. While this is perfectly understandable in the case of closed-source software systems, this is less so in other cases. These observations reinforce our stance that the SOA (maintenance) community would benefit from having a standard case study in order to benchmark solutions.

When reflecting on the case studies that we came across during our small survey, we established a number of details that we would ideally want to know from all case studies:

- The implementation technology (e.g., the programming language or the communication protocol) and the used frameworks.
- The complexity of the service-based system (e.g., the number of services or interfaces).
- The availability of the system.

With these criteria in mind, we will introduce and describe Stonehenge, the standard case study system that we propose in the next section.

Stonehenge

Apache Stonehenge is a simulation of the stock market consisting of a web application and several web services. Stonehenge provides the possibility to buy and sell shares in a single stock market, with a single currency. Apache Stonehenge was built as a joint cooperation between Microsoft and the Apache Software Foundation to showcase service interoperability between different technologies.

Our goal, however, is not to explore the field of interoperability but that of maintenance in SOA, and all that it entails. We chose Stonehenge as it provides a real world example of how services can interact together to compose a software system. However, conscious of its size, we decided to extend it in order to make

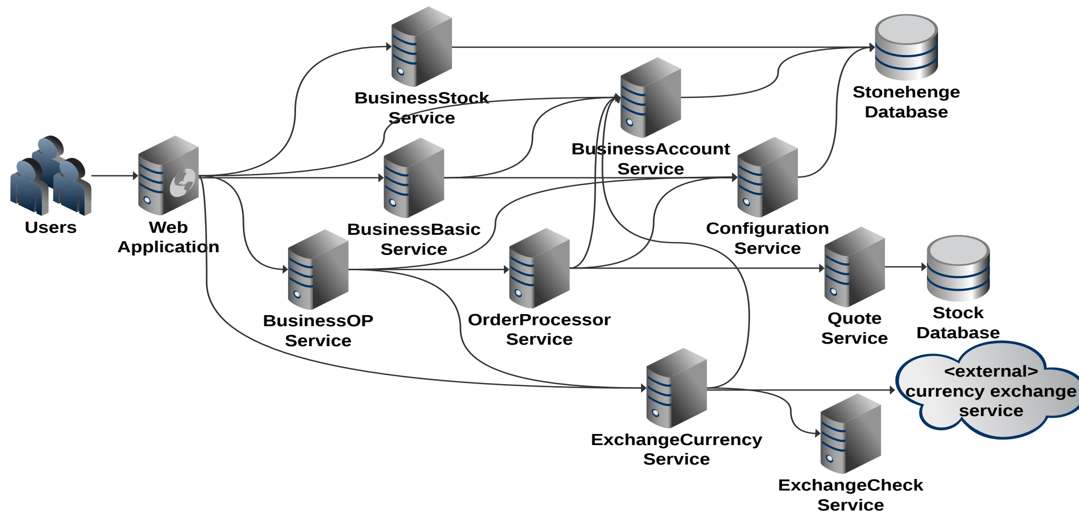


Figure 2.2: Architecture of the extended SFL-Stonehenge

it more realistic and complex. We have extended it with several new features to make the system more complex on what concerns business logic and number of services. That is, we added the possibility to maintain several wallets in different currencies, to exchange money amongst the different currencies, to enable users to buy and sell stocks in different currencies with automatic currency exchange and to use real-world data from the stock market. The result of our changes is called SFL-Stonehenge which relies substantially on the business logic of the original implementation. We have also ported the original JAX-WS-based implementation to the Turmeric SOA platform[9] due to our research goals.

Motivation

In our background research we have established that in service-oriented research there is no case study which researchers can use to compare their approaches and results. For this reason, we decided to bring forth a system that meets the criteria needed for a standard case study. For such a system we feel it is necessary that: a) it reproduces the behavior of a real-world system, b) is large or at least provides many extension possibilities that all researchers can build upon and c) it must be easy to port to different frameworks.

With SFL-Stonehenge we feel we have met these three criteria. SFL-Stonehenge provides similar behavior to that of the stock market, it is already fairly large in number of services and we plan on extending it to make it even more similar to a real system. This way, we believe SFL-Stonehenge can become the standard system which every researcher in this field can use as the “common software system" mentioned in (Sim et al., 2003).

System Description

Figure 2.2 illustrates the architecture for the SFL-Stonehenge which is comprised of 10 web services including one *external currency exchange service*, plus a *web application* for user interaction and two databases. In this section we provide an overview of what each service does and further into the section, what data is stored in each table.

Services:

- The **ConfigurationService** acts as a registry for all the deployed instances of the other services. All the other services need, therefore, to know in advance the endpoint of at least one instance of the *ConfigurationService*.
- The Business-⋆ Services mediate the interaction of the web application with the business logic of the system. For this reason, the Business-⋆ Services contain all the operations the web application is capable of performing. For example, **BusinessBasicService** and **BusinessAccountService** provide the functions for user authentication, login, and the user account. **BusinessOPService** and **BusinessStockService** are used for buying and selling stock, and checking orders and market summaries.
- The **OrderProcessorService** is solely responsible for processing the buying and selling of shares. It is meant to be invoked by the *BusinessOPService* whenever a user performs a purchase or sale of shares in the web application.
- The **ExchangeCurrencyService** makes use of Google's API for currency exchange. This service can be invoked whenever the user explicitly requests for currency to be exchanged from a wallet in a certain currency into another wallet, with a different currency. It can be also invoked by *OrderProcessorService* when the user buy a stock in a currency different to the stock's currency, the system will automatically calculate and apply the correct exchange rate.
- The **ExchangeCheckService** are responsible for checking the user's input for *ExchangeCurrencyService*. It can check if the input currencies are valid or the user has enough amount of money to exchange.
- The **QuoteService** is in fact composed of two services. Referring to Figure 2.2, the service described as *Quote Service* is a normal *pull-based* service with a SOAP interface that the *OrderProcessorService* can invoke to obtain data about specific stocks on-demand. On the other hand we also have the *Quote Data* service which performs two tasks: 1) it fills the *Stock Database* table with data and continuously updates it with data from Yahoo Finance, and 2) it provides a publish/subscribe interface (implemented using the ZeroMQ

[9]https://github.com/ebayopensource/turmeric-runtime

library) which other services, such as the *OrderProcessorService* can bind to in order to be notified for price changes in specific stock symbols.

Databases:

- The **Stonehenge Database** contains the information necessary for the basic operation of the system. Namely it contains user information, including how much money and which stocks each user owns. It also contains information about the services' endpoints and the mapping between service instances (which instance should each service use).

- The **Stock Database** contains solely information about stock prices. This table is kept separately as it is meant to be filled by an external service which continuously checks whether there are new data from Yahoo Finance and pushes them to the database.

With these services we can then have different usage scenarios. These are summarized in Table 2.2.

Table 2.2: Features available for SFL-Stonehenge

Currently available features
Purchase and sale of stocks
Price information about stock symbols
Wallets in different currencies
Automatic conversion of currencies
Management of service endpoints
User registration

In addition, we show five typical system transactions that can be performed with SFL-Stonehenge. For example, the service operation "*BusinessOPService.sell*" invokes the *ConfigurationService* service to get the url locations of the *OrderProcessorService* service, and it continues to invoke the *OrderProcessorService* service through the returned urls to submit an order of selling some stocks, which requires access to the *QuoteService* service to get the realtime price of the stocks and then update the wallet in the user's account with the returned money.

```
1. BusinessBasicService.login -->
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.getAccountProfile
    BusinessAccountService.updateAccountForLogin
2. BusinessBasicService.logout -->
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.updateAccount
3. BusinessBasicService.register -->
```

```
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.getAccountProfile
4. BusinessOPService.sell  -->
    ConfigurationService.getOPSLocations
    OrderProcessorService.submitOrder -->
        ConfigurationService.getQSLocations
        QuoteService.getQuotes
        ConfigurationService.getBSAccountLocations
        BusinessAccountService.updateWallet
5. ExchangeCurrencyService.exchCurrency -->
    ConfigurationService.getECheckLocations
    ExchangeCheckService.checkCurrency
    ExchangeCheckService.checkAmount
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.updateWallet
```

SFL Simulator

Performing experiments with a fully fledged case study is tedious. Every new experiment requires extensive adaptation to new experimental requirements. This led us to the development of a simulator, its source code is available for download[10]. It has been developed in Ruby, and used for assessing different system topologies quickly and easily. It provides functions for setting up component topologies, executing the topologies thereby gathering coverage information, and calculating diagnoses. In particular, setting up a system topology in the simulator is easy and flexible, and the simulator can run a large number of experiments for each system topology in a very short time.

A topology is created by defining a number of components. Each component is defined by the component name, component health, and failure probability. Health denotes the probability that a component will not produce an error when it is executed. 1.0 represents a healthy component, while a value in the range (0.0, 1.0) represents a faulty component with intermittent fault behavior. 0.0 denotes no fault intermittency, i.e., the component will always produce an error if activated. Failure probability denotes the likelihood of a component to propagate an error into a failure, i.e. the fault observation. 1.0 means that if a component encounters an error, this component will issue a failure, and the simulated execution will be stopped. This can also be used to discriminate fatal failures (i.e. component health < 1.0 and failure probability $= 1.0$) from warnings (i.e. failure probability $= 0.0$). In the case of a warning, the system execution will continue normally and issue a failed transaction at the end.

Components in a topology can be connected through defining a link between them with an associated invocation probability. This denotes the likelihood that a linked component will be invoked during execution. 1.0 denotes that two components will always be invoked together (i.e., representing tight coupling), and 0.0 determines that a link is never exercised.

[10]https://github.com/SERG-Delft/sfl-simulator

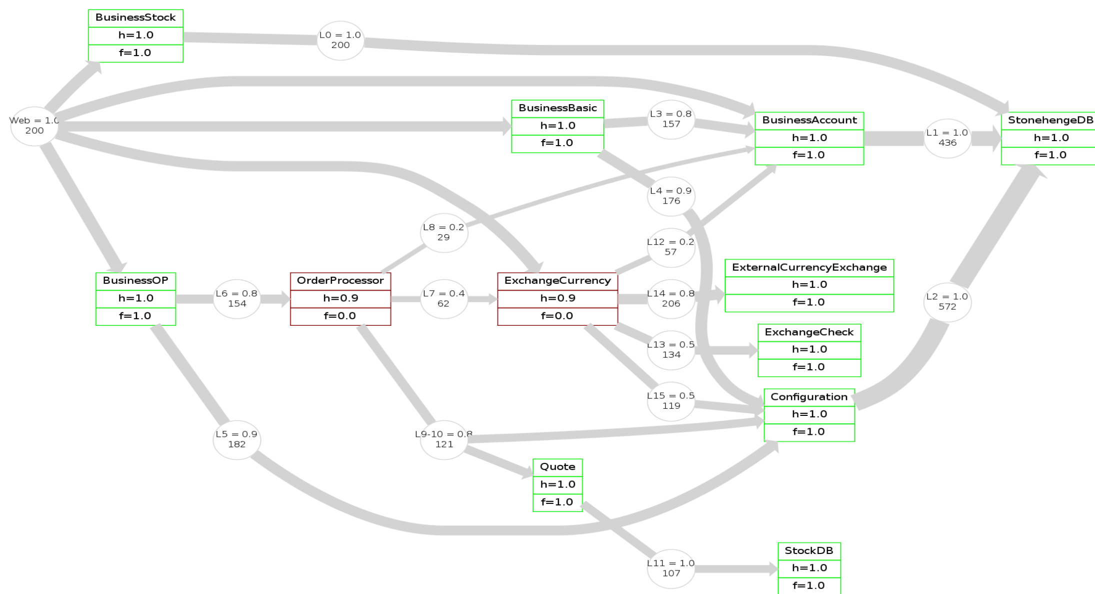


Figure 2.3: Topology of the case system produced by the SFL Simulator

Based on the topology with components and invocation links, the simulator can be controlled to perform executions. This requires that one or several entry points (components or links) are activated. Every activation of the topology leads to a particular control flow according to the initially defined probabilities, thereby generating coverage and pass/fail information. These observations are collected and used in order to calculate a diagnosis.

For illustration purposes, Figure 2.3 displays an example topology of our case study system (as shown in Figure 2.2) produced by the SFL simulator. It shows components (i.e. the services as boxes) with health and failure probabilities, h and f, respectively, and link nodes (as ovals) with their respective transaction probabilities. Figure 2.3 also shows a particular instance after 200 transactions from the Web Application (denoted as “Web" at the left hand side of the figure). The whole numbers in the link nodes denote the frequencies of invocations, and the thickness of each line also indicates this.

The usage of the SFL simulator for the research described in this thesis was twofold. First, we used it to develop our approaches described in the following chapters. Second, we applied it to simulate our original case system described in Section 2.2, for an initial assessment of our ideas in a more realistic setup.

2.3 Summary

In this chapter, we have presented a framework-based monitoring approach for service-oriented software systems, to facilitate the conprehension and maintainence for service-oriented systems. We have also proposed an open source SOA system

named SFL-Stonehenge, which we developed out of an existing application, and now put forward as standard case study system. Furthermore, we have introduced a SFL simulator, which can be used to easily and quickly assess the application of spectrum-based fault localization technique to various types of service topologies.

3 Spectrum-based Fault Diagnosis for Service-Oriented Systems

Due to the loosely coupled and highly dynamic nature of service-oriented systems, the actual configuration of such systems only fully materializes at runtime, rendering many of the traditional quality assurance approaches useless. In order to enable service-oriented systems to recover from and adapt to runtime failures, an important step is to detect failures and diagnose problematic services automatically.

This chapter presents a lightweight, fully automated, spectrum-based diagnosis technique for service-oriented software systems that is combined with a framework-based online monitor. An experiment with a case system is set up to validate the feasibility of pinpointing problematic service operations. The results indicate that this approach is able to identify problematic service operations correctly in 73% of the cases.[1]

3.1 SFL for service-oriented systems 29
3.2 Experimental Setup 32
3.3 Results and Discussion 33
3.4 Related work 37
3.5 Summary 38

Service-oriented software systems offer many benefits in realizing flexible, interoperable, and adaptable distributed infrastructures for information technology. These benefits are mainly attributable to the loose coupling of services, facilitated through underlying modern communication platforms, and their natural disposition to dynamic deployment, reconfiguration, and evolution. However, this nature of service-oriented system also presents many challenges (Greiler et al., 2009), particularly concerning quality assurance. The fact that service-based applications only fully materialize when deployed in production, i.e., ultra-late binding (Bennett et al., 2000), renders many of the traditional (offline) quality assurance methods inefficient (Canfora and Di Penta, 2006). In particular, many failures only

[1]This chapter is originally published in the proceedings of the 5th *International Conference on Service-Oriented Computing and Applications (SOCA'12)* (Chen et al., 2012).

emerge during operation time, triggered through runtime re-configuration or re-deployment of services (Canfora and Di Penta, 2006), or resulting from incompatibilities in service versioning (Papazoglou, 2008).

Although, by their very nature, service-oriented systems provide all the ingredients necessary to recover from and adapt to operation time failures (Di Nitto et al., 2008), there is no standard means in those systems to detect and diagnose emerging problems automatically, and make propositions as to *what to recover* and *where to adapt?* The fact that a problem is detected in a particular service does not necessarily mean that this service is corrupt and should be exchanged. Faults located in other services may propagate through the system and cause an otherwise healthy service to break (Mohamed and Zulkernine, 2008).

Automated software fault diagnosis can be applied to service-oriented systems in order to trace a detected problem back to the service where it originated. Fault diagnosis refers to the detection of a failure, i.e., a discrepancy between expected and observed behavior, plus the localization of its root cause, i.e., the fault that caused an erroneous system state (Zoeteweij et al., 2007a). Being able to perform automated fault diagnosis in an operational service-oriented system with minimal performance impact demands a fault localization technique with ultra-low computational overhead such as spectrum-based fault localization (SFL) (Piel et al., 2011), and inbuilt monitoring approaches for detecting failures.

In this chapter, we identify, discuss and address the issues concerning the application of SFL as fully automated diagnosis technique in service-oriented systems. Our research focuses on diagnosing problems emerging from combinations of services and their interactions, rather than identifying faulty code blocks in the services themselves. A specific issue arises through the fact that a single service is typically part of many application contexts, participating in many business goals, and, therefore, interacting with potentially many other services. This diversity in service interactions cannot typically be assessed a priori, and it cannot be guaranteed that all permutations of service connections will not eventually lead to residual defects in the overall system. We concentrate on the following research questions:

RQ3.1: How can a failure be detected in an operational service-oriented system? This is concerned with extracting relevant information from a running service-oriented system for initiating diagnosis.

RQ3.2: How can spectrum-based fault localization be applied in a service-oriented system in order to trace a failure back to its respective root cause? This focuses on identifying and providing the right input for SFL in a service-oriented system.

RQ3.3: How well does spectrum-based fault localization perform in a service-oriented system in terms of correctness of the diagnosis?

The main contributions of this chapter can be summarized as follows. We demonstrate the application of online SFL in service oriented systems, discuss the requirements of such an application and show how it can be realized in a concrete service platform. We evaluate to which extent online SFL can pinpoint faulty service operations automatically in a case system.

The chapter is organized as follows. Section 3.1 focuses on the concepts and implementation of SFL for service-oriented systems. Section 3.2 describes the case system and the setup of the experiment. Section 3.3 discusses the experimental results and the limitations of the approach. Related work is presented in Section 3.4. Finally, Section 3.5 concludes this chapter.

3.1 SFL for service-oriented systems

Concepts of SFL for service-oriented system

Applying SFL in service-oriented systems requires the SFL concepts to be adapted to the service context.

Component granularity

A service is the basic unit that can be restarted, exchanged, or otherwise treated in a service-oriented system, in case it is convicted in a diagnosis. It is a natural choice for determining the component granularity. Alternatively, a service operation, which represents a business functionality of a service, may denote the finer level of the component granularity. The component granularity affects the monitor required for measuring the component involvement (see below).

System activation

In traditional, monolithic systems a component instance will always be activated or exercised from within its own application context. Subordinate components deeper in the call graph will be activated from superordinate components, and those will be activated from users in the system context. Here, the notion of a system execution is obvious.

In service-oriented systems, this is not the case. Because a service instance serves many applications, it will not be activated exclusively from within one application context, but from a potentially arbitrary number of other applications in other contexts. To apply SFL in a service-oriented system, a system execution needs to be made explicit through introduction of a unique transaction ID. This allows a clear separation of system executions in the activity matrix of SFL.

Component involvement

In the basic SFL approach, component involvement is measured through coverage tools. However, in service-oriented systems, coverage is a delicate issue. Because of its inherent distributed nature, there is no single controlling authority that is able to produce service coverage information, by overseeing all service invocations

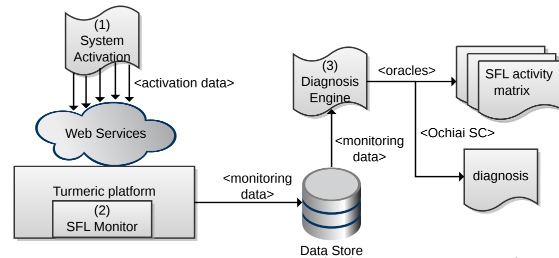


Figure 3.1: Monitoring and diagnosis architecture

and associating them with the different application contexts in which a service is used. This can only be done by the services themselves, or an underlying service framework. Applying SFL in service-oriented system requires dedicated monitors that observe the service communication and associate the services/operations with their corresponding transaction IDs.

Oracle

The oracle turns a system activation into a pass/fail-verdict. Runtime errors, exceptions, warnings and logs are natural choices for realizing oracles. These observations of the system state are readily available through the platforms managing the communication between individual services, or they are initiated through the business logic, i.e., the services themselves.

In summary, applying SFL in a service-oriented system requires that services participating in the processing of a transaction can be associated with a pass/fail observation from an oracle, thereby forming an activity matrix and an error vector. The computation of their similarity yields a diagnosis.

Implementation of SFL for service-oriented systems

The first step in applying SFL to a service-oriented system requires online monitoring to obtain information about each user transaction with the system. The second step involves the construction of a diagnosis engine that maintains the SFL activity matrix, and calculates the diagnosis. Third, component granularity is set to the service operation, because it permits a more fine-grained diagnosis. The SFL implementation for our case study is summarized in Fig. 5.1 and explained in the following sub-sections.

System Activation

Typically, a system is invoked through its user interface. However, in our case, user interaction is automated in order to evaluate our approach. We use SoapUI[2] to create XML templates of SOAP messages which are required for calling the services. Then, the templates are passed to JMeter[3] in order to generate multiple user requests that are exercised automatically.

[2] http://www.soapui.org
[3] http://jmeter.apache.org

Online Monitoring

Online monitoring follows a framework-based approach detailed in Section 2.1, realized in Turmeric[4], eBay's open source service framework. Turmeric offers many inbuilt features supporting the implementation of online monitoring required in our approach, and it confines the necessary amendments for online SFL to the absolute minimum, yielding a slender implementation.

Turmeric's internal communication is based on a pipelined architecture and controlled by two components. The *Service Provider Framework* (SPF) carries all messages sent to and received from a service at the service's provided interface, and the *Service Invocation Framework* (SIF) carries all messages sent to and received by a service at its required interface. These components handle all incoming and outgoing communication of a service. All messages sent to and received from a service are funneled through these four pipelines, where each can be accessed through a custom built handler, i.e, our online monitor. That way, we can retrieve the (unique) transaction ID, the message content, and the service plus the operation name that created the message. The transaction ID denotes all messages that belong to one transaction. This is very specific to Turmeric and essential in our approach for deducting service involvement, and, consequently, creating an activity matrix. In addition to the information encoded in the message, we retrieve information about which pipeline handled the message. With this setup, we are able to determine service operation involvement in a transaction.

Another monitoring requirement is the observation of exceptional behavior in the service-oriented system. This is used as oracle by the diagnosis engine (explained later in Section 3.1), but it is also realized in the four handlers already introduced. All services in our case system are designed to log their occurring exceptions in a data store. The handlers constantly monitor the data store for new exceptions. Once an exception is detected, it will be associated with the correct transaction through the transaction ID in the data store.

In summary, we use the following monitoring data:

- *Transaction ID*: Turmeric generates a unique ID to associate messages involved in the same transaction.
- *Service and operation name:* the name of a component in the diagnosis is made up of the service name plus the operation name.
- *Message body:* the content of the message can be checked for failures.
- *Exception:* indicates that a transaction threw an exception.
- *Pipeline:* information about which pipeline handled the message; this distinguishes between requests and responses in provided and required interfaces.

[4]https://github.com/ebayopensource/turmeric-runtime

Diagnosis Engine

This denotes the core component of our SFL implementation. It automatically reads the monitoring data from the data store, generates service involvement from each transaction, creates the output vector with the verdicts, and calculates a diagnosis by applying the Ochiai similarity coefficient.

A transaction is associated with a transaction ID, and it refers to a test case in the basic SFL approach (shown in Table 1.1). It translates to a column in the activity matrix by associating a '1' with a service operation that took part in the transaction, and a '0' with one that did not.

The output vector with the pass/fail verdicts comes from applying a built-in oracle. For demonstration purposes, we decided to focus on serious faults that either cause services to crash, or represent unexpected behavior of a service, or a faulty internal state. In general, any arbitrary oracle can be used as long it distinguishes a passing transaction from a failing one. Our oracle operates in three phases (for simplicity):

1. Serious problems that cause a complete service to crash result in missing responses from the service. The first phase of the oracle checks whether a service request generates a response, or not. If no response is returned, the oracle issues a fail.
2. If there is a response, the next phase assesses potential exception entries in the data store (generated by the monitor). If the transaction is associated with an exception, this second oracle phase will issue a fail.
3. Otherwise, the third phase will check the correctness of the message content, and the internal data states of the services involved. In case of deviations from the expectations encoded in this last phase, the oracle will issue a fail.

In all other cases, the transaction is assigned a pass.

Once the activity matrix and the output vector are complete, the similarity coefficient can be applied to calculate the likelihood of each service operation to be the faulty one in the range [0..1]. Sorting the service operations according to decreasing similarity coefficients results in the diagnosis.

3.2 Experimental Setup

We devised a case study based on Stonehenge case system detailed in Section 2.2 to demonstrate how online SFL can be applied in service oriented architectures, and validate to which extent SFL helps to pinpoint problematic service operations.

We created 160 faulty versions of our case system outlined in Fig.2.2, by applying the PIT mutation tool[5]. For each faulty version, we applied JMeter to execute 48 web service requests consecutively to cover all service operations. Upon completion of all transactions for one faulty system version, the diagnosis engine was

[5] http://pitest.org/

invoked to parse the monitoring data, identify the failures in the system, and create an activity matrix with an output vector. Then, it was assessed whether the resulting diagnosis correctly pinpoints the faulty service operation. The whole experiment was designed for the single fault case, i.e., we ensured that each version of the system contains only one fault.

Fault Injection

In our experiment we focused only on the correct functioning of the service-oriented architecture. Non-functional aspects were not considered. We were interested only in detecting a failure in the system, and tracing it back to its root cause in a service, or service operation. The scope of faults seeded into the system was, therefore, limited to this aspect. However, in general, SFL is able to identify and trace back all types of faults.

There are many mutation tools available such as μJava, Jumble, or Javalanche. However, we chose PIT, because it mutates the byte-code, rather than the source code. This represents a quick way to mutate code, and it is also safe, i.e., generation of invalid programs is avoided. Moreover, PIT provides extensive documentation, a wide range of useful mutators (i.e., mutation operations)[6], and a comprehensive reporting function. Its only drawback comes from the fact that it maintains its mutated classes in memory, so that we have to extend it with the ability to save the mutants as files.

Only the service implementation classes are mutated, and not the platform or library code. The mutation operations applied to a subject depend on what PIT finds in the service's implementation logic. Several mutators may be applied per implementation class, of which we choose one for generating one fault in the system. This is due to the single fault scenario, and it explains the high number of faulty system versions. For every version of the system, we replace the original class with its respective mutated one in the service's *.war-file*, and execute the system. All nine internal services shown in Fig. 2.2 are mutated that way.

Table 3.1 shows six mutators that PIT applies to the services of our system. In addition, the total number of each type of mutation applied in the system is shown, the kind of failure produced by this mutation, and the phases of the oracle used.

3.3 Results and Discussion

Using the experimental setup described in Section 3.2, we conducted an experiment in order to assess to which extent our approach can diagnose faulty service-oriented systems.

[6]http://pitest.org/quickstart/mutators/

Table 3.1: Active Mutators in the experiment

ID	Mutator	#	Error in the system	Oracle
1	Negate Conditionals	44	wrong internal state or response, null or runtime exception	1-3
2	Return Values	50	wrong response, null or runtime exception	1-3
3	Conditionals Boundary	3	wrong internal state or response	3
4	Void Method Call	60	wrong internal state	3
5	Math Mutator	1	wrong internal state	3
6	Increments Mutator	2	wrong response	3

Experimental Results

A diagnosis refers to a component ranking according to the similarity coefficients. If the diagnosis is not accurate, it might well rank healthy services before the faulty one. Accuracy of a diagnosis can be measured through residual diagnosis cost (Gonzalez-Sanchez et al., 2010a), i.e., the cost of unnecessarily treating healthy services before arriving at the real faulty one.

We are not so much interested in the accuracy of an individual diagnosis, but rather look at the overall diagnosis capability of our proposed approach in a service-oriented architecture. We refer to this as the correctness of the diagnosis. This is a stronger criterion than residual diagnosis cost, but it simplifies the analysis. Here, a correct diagnosis is achieved, if the real faulty service is always ranked at the top. If the faulty service is ranked lower (e.g., 2nd, 3rd, ...), we consider the diagnosis to be incorrect.

Table 3.2: Experimental Results

Services	Applied Mutators	# of Mut.	Diagnosis Correct	Diagnosis Incor.	Correct Diagn.
BusinessAccountService	2,4	7	7	0	100%
BusinessBasicService	1,2,4	27	23	4	85%
BusinessOPService	1-4	19	14	5	75%
BusinessStockService	2	8	8	0	100%
ConfigurationService	2	9	9	0	100%
ExchangeCheckService	1-3	8	8	0	100%
ExchangeCurrencyService	1,2,4	24	3	21	13%
OrderProcessorService	1-5	41	28	13	68%
QuoteService	1-4,6	17	17	0	100%

Table 3.3: Reasons for Incorrect Diagnoses

Services	Incorrect Diagnoses	No Activation	Tight Interaction on Failure
BusinessBasicService	4	2	2
BusinessOPService	5	1	4
ExchangeCurrencyService	21	2	19
OrderProcessorService	13	4	9

Table 3.2 summarizes the diagnosis results for our extended version of SFL-Stonehenge[7]. For each service, the table indicates the type of the mutation operations used (IDs displayed in Table 3.1), the total number of mutations performed, the correctly and incorrectly performed diagnoses, and the percentage of correct diagnoses.

In total, 117 out of the 160 faulty system versions are diagnosed correctly, yielding a 73% success rate for our experiment. Faulty versions of five out of the nine services used in our system can always correctly be diagnosed by SFL. However, the mutants of four services cannot be diagnosed so successfully. A careful analysis of these cases presents a number of issues to be discussed in the following.

Reasons for Incorrect Diagnoses

We can identify two significant reasons for why diagnoses are incorrect (summarized in Table 3.3):

(1) No activation of the fault: if the fault in a service implementation is not triggered, e.g. through user interaction with the system, there will be no failure, and, consequently, the diagnosis will be incorrect. This is the case in five system versions, and it must be regarded as a general problem in all passive monitoring-based approaches. Residual defects in a service-oriented system can only be diagnosed when they are actually triggered and detected.

(2) Tight service interaction: this presents a particular challenge in SFL. When services are always invoked together, the similarity coefficient will assign the same value to all tightly linked services. They are treated as if they were one combined service. However, in our case system, a peculiar situation can be observed. Some services work together in combination in one transaction and make it fail, while they participate as individuals in other transactions that pass. Here, these services are not treated as if they were one combined faulty service, and it is attributable to the calculation of the similarity between the outcome vector and the activity vector. Involvement in a passing transaction weighs more than non-involvement in a failing transaction. Services that participate in a failing transaction may be convicted by the similarity coefficient, but if one service participates in a passing transaction, its conviction will be exonerated, which leads to an incorrect diagnosis in such cases. Table 3.3 indicates that this happens quite often in our example system.

Multiple Faults

Initially, we stated that we are only interested in the single fault case, and the Ochiai similarity coefficient represents a single-fault approach. However, in our example system, we observe that one mutation in a service implementation may affect more than one service operations. Since the granularity is at the service operation-level, rather than at the service-level, we actually introduce multiple

[7] https://github.com/SERG-Delft/sfl-stonehenge

faults into our system. This problem is attributable to a mismatch between the granularity of the fault injection and the granularity of the diagnosis. SFL always ranks one of the faulty service operations at the top (but not all of them), meaning it finds the fault. Which of the several faulty service operations will be ranked top, depends on its number of activation. According to our definition of correctness, we treat this result as a correct diagnosis.

Discussion and Lessons Learned

The experiment demonstrates the feasibility of applying online SFL to diagnose service-oriented systems. The results indicate that our approach is able to pinpoint problematic service operations with high correctness in many cases.

Methodological Limitations

The experimental results also demonstrate that no activation of a fault causes incorrect diagnoses, i.e., in our evaluation. In a real setting, a fault that is not activated does not exist, and it highlights a fundamental problem in all coverage-based quality assurance approaches. The online monitor can only passively wait for the system invocations to appear. Monitoring cannot actively initiate relevant transactions to cover a fault. In order to trigger such residual defects, it is possible to conduct online testing and compensate the deficiency of monitoring by actively running test cases to add the required coverage. However, this is out of the scope of our current research, and it will be considered in the future.

We also observe that tight service interaction can influence the diagnosis. Service operations that are always invoked together in passing as well as in failing transactions, actually behave as if they were one single component, and the diagnosis treats them as such. If one component contains the fault, every one of its tightly coupled peers will also receive the blame for this fault according to the diagnosis. This is an interesting observation, and it raises the question of what an adequate architecture is. Could services be designed in order to become better diagnosable, e.g. increase their cohesion? Or, can their interactions be designed in different ways, as to permit more variety in their invocations, e.g. increase their coupling, so that alternative invocation paths may yield more or better diagnostic information? These are also interesting questions for future work.

A special case of tight coupling comes from service operations that always cooperate in failed transactions, but pass when invoked individually. This is attributable to how the similarity coefficient is biased towards convicting services that participate in faulty transactions, and exonerating services that participate in passing transactions. Future research should carefully assess the relation between the architecture and the similarity coefficient applied, and evaluate to which extent additional information can improve the diagnosis. This is also related to the previous discussion.

Although, in this work, we specifically target the single fault case for demon-

stration purposes, we acknowledge the fact that this is not realistic. The Ochiai coefficient is limited to the single fault. Even though Ochiai identifies the root cause of the failure, it cannot pinpoint all operations involved in exhibiting it. In this case Ochiai fails to convict all faulty operations, which is to be expected. Heavyweight, bayesian- and model-based diagnosis approaches (Abreu et al., 2009b) do work in the multiple fault case. However, it remains to be evaluated in future work how such techniques can be applied online and in the context of service-oriented architectures.

Implementation Limitations

A fundamental concern that we have not considered in our experiment is the performance overhead incurred, through incorporating online diagnosis in a service-oriented system. In our current setting, only the monitoring is performed online. The other steps are done by the diagnosis engine which is completely detached from the service-oriented system, and are, therefore, not creating any overhead in the services. Monitoring is heavily based on Turmeric's internal profiling mechanisms. These are permanently activated in the framework. The handler code we added is marginal, but obviously not negligible. In future work, we intend to measure not only our own overhead incurred by the handlers, but, more importantly, also assess the performance overhead of Turmeric's internal profiling mechanisms. This is an important research question for the future, since many modern service frameworks come well equipped with similar monitoring and profiling tools.

Other implementation limitations also concern the service platform we chose, i.e. Turmeric. For example, our first oracle phase checks for missing responses. This is very specific to Turmeric. A Turmeric service is supposed to always return a message. Otherwise, it indicates a serious problem.

Another issue which is not documented in the experimental results is the fact that our online monitoring implementation cannot fully support asynchronous communication in our case system. During the implementation of our example system, we realized that the SFL monitor sometimes misses a service response coming from an asynchronous invocation. This might be attributable either to faulty behavior of Turmeric, or due to an undocumented feature of Turmeric. In any case, this makes the diagnosis fail, and eventually, we resigned from including asynchronous service invocations. In future work, we will definitely aim at resolving these issues and include asynchronous service invocation.

3.4 Related work

Chen et al. present *Pinpoint* (Chen et al., 2002), a tool based on similarity coefficients. However, they do not address the problems of inter-service diagnosis (that services are used in different contexts), and use a weaker similarity coefficient. Zhang et al. (Zhang et al., 2012b) propose a hybrid approach, combining a matrix- (Zhang et al., 2009) and a Bayesian-based probabilistic diagnosis method

for SOA systems. Since the dependency matrix is generated before operation, the diagnosis cannot adapt well to the dynamic nature of SOA. Even though, the authors considered various ways to reduce the computational complexity, bayesian approaches are still heavyweight compared to spectrum-based approaches. Mayer et al. (Mayer et al., 2010b) diagnose faults in business processes for SOA systems. Their approach requires partial information of process executions by reasoning about possible activities in system behavior. However, the models for diagnosis are rather complex, and proper evaluation is still pending.

Grosclaude describes a model-based monitoring approach for component-based systems, and suggests to use transactions IDs in order to associate messages sent between components (Grosclaude, 2004). This is also proposed by (Chen et al., 2002), and we see it as a standard approach to determine which service takes part in which system transaction. Although slightly less related, Zhang et al. (Zhang et al., 2009) present a framework for diagnosing QoS problems in SOA through monitoring service states. Another interesting approach is introduced by Heward et al. (Heward et al., 2011), in which they propose an algorithm for optimization of monitoring configurations for web services. They use an optimization algorithm in order to reduce the monitoring overhead in a service-based system, something that would also benefit our proposed techniques.

3.5 Summary

The goal of our work presented in this chapter is to demonstrate a first realization of automated online fault diagnosis for service-oriented architectures. Referring to our original research questions, we looked at:

RQ3.1: How a failure can be detected in an operational service-oriented system

We enabled framework-based monitoring, reusing the tools of an existing service platform, i.e. Turmeric, for transaction tracing. In addition, we devised a three-phased oracle using the monitoring in order to associate failure information with the transaction traces. Both monitor and oracle generate component involvement and pass/fail information required in fault diagnosis.

RQ3.2: How spectrum-based fault localization can be applied in a service-oriented system

The fault localization technique is implemented in a dedicated (external) diagnosis engine for efficiency. This accesses the information generated by the monitor and the oracle and turns that into an activity matrix and an output vector, and then, calculates the diagnosis.

RQ3.3: How well spectrum-based fault localization can identify faults seeded into service implementations

The results confirm the feasibility of the approach, and indicate a high success rate of the diagnoses, i.e., 73% correctness. The fraction of incorrect diagnoses can be explained after careful analysis, which results in a number of feasible directions for future work.

The limitations of our current approach are readily recognized: diagnosis based only on passive monitoring and implementation-specific monitoring, influence on the diagnosis through the service topology, and the single fault case. Our next steps in future work will address multiple faults in a service-oriented system, which is more realistic. Later, we will assess the performance overhead, in order to optimize the monitoring and the oracle (Chapter 5). Finally, it would be interesting to see how different topologies of service-oriented system affect the accuracy of diagnosis (Chapter 4).

4

Effects of Monitoring Topology on Spectrum Based Diagnosis

Spectrum-based fault localization (SFL) is a statistical fault diagnosis technique that infers diagnoses from runtime observations. It works by monitoring system transactions, and comparing activity information with pass/fail observations. SFL requires the monitors, which recover the activity data, to be organized to produce optimal information for the diagnosis. This organization is termed topology.

Optimality of monitoring topology for diagnosability represents a search or optimization problem amenable to be addressed by meta-heuristic algorithms. In order to study the effects of topology on the production of diagnoses through SFL, we use genetic algorithms (GA) to generate topologies that lead to improved diagnosability. We illustrate how monitoring topologies affect the diagnosability of systems, and how GA can help to study these effects. We derive general characteristics of topologies to facilitate SFL-based diagnoses.[1]

4.1 Topology Effects **42**
4.2 GA for Topology Optimization **43**
4.3 Experiments **44**
4.4 Discussion **52**
4.5 Related Work **54**
4.6 Summary **55**

Spectrum-based fault localization (SFL) is a lightweight statistics-based automatic diagnosis approach that can be applied to identify misbehaving system parts (Chapter 3). It works by automatically inferring a diagnosis from symptoms (Abreu et al., 2009a). The diagnosis is a ranking of potentially faulty system components and the symptoms are observations about component involvement in system activation, plus pass/fail information for each activation (Gonzalez-Sanchez et al., 2011). The activation of the system is expressed in terms of a binary activity matrix representing for each component whether it has been involved in a transaction.

[1]This chapter contains our work published at the *24th International Workshop on the Principles of Diagnosis (DX'13)* (Chen et al., 2013c).

The pass/fail information is expressed in terms of a binary output vector. A diagnosis is determined by calculating the similarity between each component's activation vector and the output vector. A component whose activity vector is more similar to the output vector is more likely faulty than other components, and ranked higher as suspect.

The application of SFL creates a particular challenge, i.e. the placement of the monitors for gathering component involvement information. We refer to this placement as the *monitoring topology* of the diagnosis system. In principle monitors may be placed anywhere in the monitored system. However, the places should be selected carefully to yield the best results in terms of calculating correct diagnoses. Typical places are in or around the system components, or collections of system components, or between them. Finding monitoring topologies that lead to high diagnosability represents a difficult optimization problem amenable to be solved by meta-heuristic algorithms, such as genetic algorithms. This brings us to the formulation of the following research questions:

RQ4.1: How can genetic algorithms be used to optimize monitoring topologies for spectrum-based diagnosis?

RQ4.2: What are characteristics of monitoring topologies that are optimal for spectrum-based diagnosis?

One contribution of this chapter is the application of GA, including the definition of adequate fitness functions, in order to study the optimality of topologies for better diagnosability. Another contribution is the formulation of general characteristics of topologies that improve SFL-based diagnoses. Optimization of topology is a well-known problem domain to be addressed by genetic algorithms, e.g. (Chapman et al., 1994), however, the use of GA in spectrum-based software fault localization is novel, in particular the formulation of the fitness introduced.

The remainder of this article is organized as follows. Section 4.1 introduces SFL and how it is affected by topology. Section 4.2 illustrates how GA can be applied for SFL topology optimization. Section 4.3 outlines our experiments performed, and Section 4.4 presents the discussion of their results, and lessons learned. Finally, Section 4.5 lists the related work, and Section 4.6 summarizes and concludes the Chapter and gives an outlook on future work.

4.1 Topology Effects

Table 4.1 illustrates how the monitoring topology affects SFL. The three topologies shown are comprised of six components, C_1 – C_6. In Topology A and B, every component represents a monitor collecting component activation information. In Topology C, component C_3 is split to represent two monitors, i.e. $C_{3.1}$ and $C_{3.2}$. Component C_4 is faulty with health h=0.0, all other components are healthy

Table 4.1: Activity and Similarity Coefficient (SC) for 3 example topologies

Topology A		
Comp.	Activity Matrix for Topology A	SC_o
C6	0000000000000000000000000000000000000000	0.000
C2	10001111010000001100001010010110000000100	0.592
C5	10011111011111001101100010101110001001 00	0.742
C4	1111111111111111111111111111111111111111	1.000
C3	1111111111111111111111111111111111111111	1.000
C1	1111111111111111111111111111111111111111	1.000
Output	1111111111111111111111111111111111111111	

Topology B		
Comp.	Activity Matrix for Topology B	SC_o
C6	0000000000000000000000000000000000000000	0.000
C2	0000000000000100001000000001100001000010 0	0.340
C5	0011101001100000000111010101110010010100 1	0.668
C1	1111111111111111111111111111111111111111	0.949
C3	1111101111111011111111111111111111111111	0.973
C4	1111101111111010111111111111110111111111 11	1.000
Output	1111101111111010111111111111110111111111 11	

Topology C		
Comp.	Activity Matrix for Topology C	SC_o
C6	0000000000000000000000000000000000000000	0.000
C2	1010000000000000000010000000100000000000 00	0.344
C5	0001000000011000110111001010011100000000 00	0.445
C3.2	1001001000000111001011110110111001110101	0.601
C1	1111111111111111111111111111111111111111	0.689
C3.1	1001100100111001110110111010110000100110	0.851
C4	1001000100001011110110111010010001100110	1.000
Output	1001000100001011110110111010010001100110	

(h=1.0). The invocation probabilities between the components are represented by the numbers noted down at the arrows. Topology A shows a particular system characteristic, with components C_1, C_3, C_4, and C_6 being tightly coupled, indicated by the 1.0 invocation probabilities. This represents a specific inhibiting factor for the calculation of the diagnosis as shown by the poor component ranking in the corresponding activity matrix. Components C_1, C_3, C_4 get attributed the same ranking, rendering this diagnosis ambiguous. Topology B represents a relaxation of the tight couplings between components C_1, C_3, C_4, and C_6 to a lower value of 0.9. This leads to a better Similarity Coefficient SC_o calculation and a correct diagnosis of the faulty service for Topology B. The same effect is achieved, if a component is split up into two observation points (C3.1 and C3.2 in Topology C). The calculation of the SC_o for Topology C is also correct and unambiguous. This simple example suggests that topology has a major influence on the calculation of a diagnosis with SFL. The goal of this chapter is to study these effects with the help of GA, and derive general characteristics of topologies resulting in high diagnosability.

4.2 GA for Topology Optimization

We favor genetic algorithms over other optimization heuristics, since they are adequate for our problem domain (Chapman et al., 1994) and easy to apply. GA

represent a group of optimization techniques, loosely related to the mechanisms of natural evolution with reproduction and selection (Goldberg, 1989). The parameters of the optimization problem are encoded as binary string (chromosome). Each chromosome represents an individual in a pool of solutions (population). During reproduction, pairs of individuals are selected for recombination and some parts of their chromosomes form a new individual. This is termed crossover and controlled by the crossover operator according to probability P_c. After recombination, individual bits of the new chromosome are mutated by a mutation operator according to a low mutation probability P_m. The resulting new individuals are assessed with the fitness function. This measures how well an individual solves the original problem. Fitter individuals have a higher chance to reproduce. This is controlled by the so-called selection operator. The fittest individuals remain in the population and build the basis for the next generation.

In SFL, the topology is represented in the activity matrix. It expresses for every observation point (monitor), whether it has been activated in a transaction or not. The coverage of the topology can be expressed as one binary string, making a mapping to a GA-chromosome straightforward. Every line in the activity matrix becomes a substring of the chromosome. The fitness distinguishes good from poor solutions, and it represents the adequacy of a topology to support the calculation of a diagnosis. Diagnosability can be expressed in terms of the extent to which all diagnoses carried out on an activity matrix coming from that topology, are correct diagnoses. In other words, if a topology is organized such that every faulty component can be identified correctly, the topology may be referred to as highly diagnosable. This can be achieved by consecutively setting all components used in the activity matrix to be faulty, and then calculating the similarity coefficient for each fault scenario. This yields a value representing how well a topology facilitates the discovery of faults in components. Topologies leading to higher fitness values will lead to better pinpointing of all faulty components.

The ruby-method `f_high` (Fitness A in Table 4.2) represents the basic fitness function yielding high overall SC. First, in the so-called *genotype-phenotype transfer*, the GA chromosome is translated into the problem domain, i.e. the binary gene-string is transformed into a binary activity matrix. Second, each component activation vector is set to be the output vector, and the SC is calculated. Third, the SC values are summed up.

4.3 Experiments

We performed a number of experiments in order to have GA generate highly diagnosable topologies, and then to derive general characteristics for diagnosable topologies. The genetic algorithm used for these experiments can be downloaded.[2] It uses the following rudimentary operators.

[2]https://github.com/SERG-Delft/rusiga

Table 4.2: Fitness A: high overall SC

```
# Fitness A: high overall SC
def f_high(chrom, act)
  # genotype -> phenotype transfer
  activity = Array.new
  while (a=chrom.take(act)) != [] do
    activity << a
    chrom = chrom.drop(act)
  end
  # SC calculation
  sc = Array.new
  activity.each do |output_vec|
    activity.each do |activity_vec|
      sc<<ochiai(activity_vec, output_vec)
    end
  end
  # fitness: sum up sc values
  fitness = sc.inject{|sum,x| sum + x}
  return fitness
end
```

Two individuals are selected for recombination based on tournament selection (Miller and Goldberg, 1995). This chooses N_t individuals from the population randomly, and returns the fittest in this tournament. The actual recombination is done according to the uniform crossover operator (Syswerda, 1989). It determines for every bit in the chromosome, according to a probability P_c, whether the value for the new individual (offspring) is taken from the first or from the second parent.

The other GA-parameters depend on the complexity of the particular problem size to be solved. The population size N_p, and the tournament size N_t are set to different values in the different experiments, reflecting the chromosome size of the respective problem, i.e. according to the size of the activity matrix (or based on experience). Bigger activity matrices represent larger search spaces and require bigger populations for better sampling of the search space. Experiments with large topologies are possible but would require more space for presentation. Therefore, the topologies shown are limited to five components. Experiments with larger numbers of components yield similar results. The GA maintains and evolves the N_p fittest individuals. Crossover probability P_c is set to 0.5 in all experiments, and mutation probability P_m is set to a low value of 0.001. These were determined through initial experiments and found to provide acceptable results. Every experiment was repeated 20 times. There may be better GA implementations or operators to chose from, however, the ones introduced here are sufficient to produce usable results.

Assessing the Setup.

The first experiments performed serve as assessment in terms of whether or to which extent the GA is able to generate highly diagnosable activity matrices. We assume a diagnosable topology is represented by high overall SC values. This can be tested by iteratively setting the output vector in the fitness function equal to each component's activation vector (Fitness A in Table 4.2). Each component is set to be faulty in the calculation of the SC (single fault case), resulting in $SC_o = 1$

Table 4.3: Assessment of the experimental setup

100 Generations, 40 Activations	
N_p=120, N_t=6, P_c=0.5, P_m=0.001	
best random individual (fitness=16.75)	
C_1	1010101001111001011111000110101110100100
C_2	0011101100110000101101101010110011110101
C_3	1001010011011111111000011110110001011101 1
C_4	0101101110011001100101101011100010110001
C_5	0101111111001001010101001110101010101001
best final individual (fitness=24.88)	
C_1	0111101110111111101111101111111110110101
C_2	0111101110111111101111101111111110110111
C_3	0111101110111111101111101111111110110111
C_4	0111101110111111101111101111111110110111
C_5	0111101110111111101111101111111110110111

for this comparison, and we expect the GA to produce activity matrices in which all component activations are alike. An example is shown in Table 4.3, above. The first activity matrix (fitness=16.75) represents the best random individual from the first generation. The second activity matrix (fitness=24.88) represents the fittest individual after 200 generations. The success of this optimization example is quite obvious. All component activity vectors are highly similar, representing a highly diagnosable activity matrix expressed by the calculation of high overall SC. In fact, the most optimal solution in this example is fitness=25, when all combinations of component activity vector and output vector yield a 1.0 as SC value, i.e., when they are identical. In this example, the fittest individual is only 1 bit flip away from the optimal solution, i.e. in the penultimate spectrum of C_1.

Even though, this experiment is successful in terms of assessing our experimental setup, it is useless in diagnosis, because the activity matrix represents a topology in which all components are tightly coupled. If C_1 is invoked, all other components will also always be invoked, leading to components C_1 to C_5 being assigned the same ranking (SC = 1.0; compare with Topology A in Table 4.1), and resulting in an ambiguous diagnosis. As a consequence, we have to extend the adequacy criterion for topologies: "A topology is diagnosable, if it facilitates the detection of all faults in a system, *and their unambiguous identification*," i.e. it must not generate duplicate top SC_o.

Topologies for Discriminable Diagnoses.

In this experiment, the fitness function from the previous setup is adjusted to award topologies higher fitness, which result in high overall SC, but also lead to *discriminable* diagnoses, thereby addressing ambiguity. The fitness function `f_discrim` (Fitness B in Table 4.4) illustrates this extension. It awards individuals that lead to one top ranked component, and a number of lower-ranked components. Moreover, it can be configured to minimize (`diff=:low`) or maximize (`diff=:high`) the difference between the top ranked and all lower-ranked components. Table 4.5 shows examples for both optimization goals.

Adjusting `diff` to `:high` leads to a large number of '0's in the final activity ma-

trix compared to a random activity matrix from the early generations, representing a lot of unique component activation. This means that discriminable diagnoses, indeed, can be supported by the topology of the system, and that inactivity of the components, indicated through the many zeroes, supports this. In other words, high diagnosability can be achieved through inactivity observations, or through activation of components in isolation, which is the opposite of tight component coupling. This is an interesting result, because for the topology it means, that having components which may be activated individually rather than in combination with other components, helps separating system executions, and thus, improves the diagnosability of the system. This comes from how the SC_o calculates similarity. Completely inactive spectra are ignored by the SC_o, but spectra with fewer activations provide more useful information for SFL than spectra with more activations. For example, a spectrum with $a_i = [0, 0, 0, 0, 1]$ is more useful than another one with $a_j = [1, 1, 1, 1, 0]$, because if the transaction a_i fails, this will result in the only one activated component in a_i being blamed more. This outcome may seem like "the bleeding obvious," but, because complete decoupling of all components is not realistic in real systems, in the future, we will have to assess whether or to which extent a GA may be able to generate optimal monitoring locations that help to exploit this property, at least to a certain extent.

Table 4.4: Fitness B: discriminable SC

```
# Fitness B: discriminable SC
def f_discrim(chrom, act, diff=:high)
  # genotype -> phenotype transfer
  # same as f_high()
  ...
  # SC calculation
  # same as f_high()
  ...
  # fitness: discriminiable SC
  highest_sc = (sc.sort!)[-1]
  pivot = sc.find_index(highest_sc)
  low_sc = sc[0..pivot-1]
  top_sc = sc[pivot..-1]
  sum_top = top_sc.inject {|sum,x| sum+x}
  sum_low = low_sc.inject {|sum,x| sum+x}
  return sum_top - sum_low if diff==:high
  return sum_low - sum_top if diff==:low
end
```

Setting `diff` to `:low` shows different results. Even though the activity matrix contains many '1's, indicating tight coupling between the components, conclusive diagnoses can be calculated, if the topology can provide just enough discriminative information, e.g. some '0's in some spectra. Looking only at the failing spectra in which each component was activated, would lead to ambiguous diagnoses (comparable with Topology A in Table 4.1). Because there is slight variation in other spectra to compensate for the tight coupling, the information contained in the activity matrix is just diverse enough in order for the diagnosis algorithm to

Table 4.5: Examples for discriminable diagnoses

30 Generations, 40 Activations N_p=50, N_t=3, P_c=0.5, P_m=0.001; **diff=:high**		**30 Generations, 40 Activations** N_p=50, N_t=3, P_c=0.5, P_m=0.001, **diff=:low**	
fitness=-2.98 (best random individual)		fitness=7.092 (best random individual)	
C_1	1110100010001000101100111001110100100011	C_1	1001111101010000100011101111100100011101
C_2	1000101000100100100110100000000011001110	C_2	1011011000010110111100111101110110011010
C_3	1111010000100101100001000100100101110000	C_3	1100111001010110101100101111111010101110
C_4	1001001111110111100111110111011000010100	C_4	0101101110001000101010101101101001110011
C_5	0100110111000010010110110011010000101001	C_5	1010001001000110101100011001111101101011
fitness=4.606 (best individual after 30 gen.)		fitness=12.978 (best individual after 30 gen.)	
C_1	0000100000000100000000000011000000011100010	C_1	1011111001011111100010111111100111111111
C_2	0110000000000010000000101000010001000011100	C_2	1111111011011111101110111111100111111011
C_3	0000000001000010001000000010000000000001	C_3	1111111011011111101000111111111111111011
C_4	0001001100100001000100000000000000010000	C_4	0111101011011100101010111111100101111111
C_5	1100010110010000110001010000111000000000	C_5	1111100011011111101000111111100111111011

come up with an unambiguous ranking. An increase in observation diversity can be achieved by adding observation points. One approach could be the inclusion of observations representing the invocation links between the components. Another approach is the instrumentation of the components themselves in order to acquire more diverse observations. This second approach has been demonstrated to improve diagnosis considerably for service-based systems (Chen et al., 2013a). In any case, both approaches also raise the question of the optimal number of observation points for high diagnosability w.r.t. low monitoring overhead, to be addressed in future work.

Topologies for Intermittent Fault Behavior.

In the previous experiments, activation of a faulty component always lead to a failure. Here, we would like to assess to which extent topology influences the quality of the diagnosis when components exhibit intermittent fault behavior. Intermittency, i.e. a component fails occasionally, is quite common in software, and it is not attributable to random faults (as in hardware). Even though, software exhibits deterministic fault behavior, intermittency comes from the mismatch between the monitoring granularity and the activation granularity (basic block level). Hence, intermittency presents a monitoring topology issue.

Fitness function `f_randinterm` (Fitness C in Table 4.6) realizes intermittency through removing all '1's from each output vector except for a number of randomly chosen ones (e.g. 3 random failure observations). This yields similar results as presented in Table 4.5, with `diff` set to `:high` and `:low`, respectively, so we omitted an example. Consecutively using each activation vector as output vector, leads the optimization to be focused only on the generation of high/low differences between top ranked activations and the lower ranked activations, thereby ignoring the intermittency target.

Amending the fitness function by focusing on only one faulty component, leads to a more differentiated outcome (through Fitness D, in Table 4.7). Table 4.8 shows two examples with five constant failures seeded into the output vector, and with `diff` set to `:high` and `:low`, respectively. Looking at the two examples, the so-

Table 4.6: Fitness C: random intermittency

```
# Fitness C: random intermittency
def f_randinterm(chrom, activ, diff=:high)
  # genotype -> phenotype transfer
  # same as f_high()
  ...
  # SC calculation
  sc = Array.new
  activity.each do |output_vec|
    output_vec.remove_all_ones_except_rand(3)
    activity.each do |activity_vec|
      sc<<ochiai(activity_vec,output_vec)
    end
  end
  # fitness: discriminable
  # same as f_discrim()
  ...
end
```

Table 4.7: Fitness D: constant intermittency

```
# Fitness D: constant intermittency
def f_constinterm(chrom, activ, diff=:high)
  # genotype -> phenotype transfer
  # same as f_high()
  ...
  # SC calculation with const. output vector
  output_vec = [0,0,0,1,0,0,0,0,0,1,0,0,0,...]
  activity.each do |activity_vec|
    sc<<ochiai(activity_vec,output_vec)
  end
  # fitness: discriminable
  # same as f_discrim()
  ...
end
```

lution of the GA to the intermittency problem is both cunning and ironic: *"in an optimal topology, faulty components should only be executed when they are guaranteed to fail,"* which avoids intermittency altogether and is not very useful. Further, when `diff` is set to `:high`, it becomes apparent that when the failing component, C_5 in this example, is activated, none of the other components is activated, suggesting again, that the ability to activate components in isolation is advantageous. And when `diff` is set to `:low`, ambiguous diagnoses can be resolved through additional observations, i.e. through the very few additional '1's in the bottom activity matrix. This confirms our previous observations. Intermittency cannot be addressed with this kind of experiment.

Freely Evolved Topologies.

Up to this point, we have had the GA evolve topologies based on a predefined output vector with seeded faults. That way, we could define the interesting error scenarios, and have the GA generate optimal activity matrices. In this experiment, we let the GA not only evolve the activity matrices, but also their corresponding output vectors. It means, we have no control over the number of failure obser-

Table 4.8: Examples for fault intermittency

	200 Generations, 40 Activations	
	N_p=200, N_t=3, P_c=0.5, P_m=0.001; **diff=:high**	
	fitness=0.377 (best random individual)	
C_1	0000110011100110100000000100010110011000 1	
C_2	0001011111000100001001010111011110101111	
C_3	1100101010101110110101001101000010110001	
C_4	1001100110111100101100010000011011111110	
C_5	1010010010110111111011100001001100111100 1	
O	0111000000000000000000000000000000000110	
	fitness=1.0 (best individual after 200 gen.)	SC_o
C_1	0000101111000000110100111011111100011000 0	0.00
C_2	1000010101010001101001000111110100010000	0.00
C_3	0000010011111011011110100011000110110001	0.00
C_4	0000111111110010011000101110000001111001	0.00
C_5	0111000000000000000000000000000000000110	1.00
O	0111000000000000000000000000000000000110	
	200 Generations, 40 Activations	
	N_p=200, N_t=3, P_c=0.5, P_m=0.001; **diff=:low**	
	fitness=1.105 (best random individual)	
C_1	1111110000110000100011111100010011110010	
C_2	1111100100010001000111100001100000010011	
C_3	0111100111101010101101000110000011110101	
C_4	0111110100100011101000111000111100000101	
C_5	0111010010010100100100101010001111110110	
O	0111000000000000000000000000000000000110	
	fitness=2.652 (best individual after 200 gen.)	SC_o
C_1	0111000000001000000000000000000000000110	0.91
C_2	0111100000000000000000000000000000000110	0.91
C_3	0111000001000000000000000000000000000110	0.91
C_4	0111000000000000000000000000000000000110	1.00
C_5	0111000010000000000000000000000000000110	0.91
O	0111000000000000000000000000000000000110	

vations generated in the output vector, and we cannot tell whether the diagnosis is correct, because we cannot seed any particular faults. For these experiments, a 6th component is added to the GA chromosome representing the output vector, and Fitness E in Table 4.9 is used for evaluation of the individuals. The fitness function is slightly different compared to the earlier ones, because of the output vector taking part in the evolution. Setting `diff` to `:low` results in a selective pressure favoring many failure observations to be produced as shown in the example activity matrix on the top right hand side of Table 4.10. Because the number of failing transactions is unrealistically high for real software systems, we set `diff` to `:high`, resulting in much lower number of failure observations. This is shown in the example activity matrix on the bottom right hand side of Table 4.10.

Two noteworthy results can be observed in this second case. First, as noted earlier, being able to activate components individually supports the diagnosis. Second, intermittency can be dealt with. The faulty component, C_3 in this example, is sometimes invoked without resulting in a failed transaction. This leads to the $SC_o < 1.0$. In fact, when C_3 is invoked in isolation, it fails, if it is invoked in combination with any of the other components it passes. This is a clear indicator of a missing observation point, either in the component C_3 itself, or in one of its peers

Table 4.9: Fitness E: Freely evolved topologies

```
# Fitness E: freely evolved
# with output vector
def f_discrout(chrom, act, diff=:low)
  # genotype -> phenotype transfer
  # same as f_discrim()
  ...
  # SC calculation
  # ouput -> last comp act vector
  output = activity_matrix[-1]
  activity_matrix.delete_at(-1)
  sc = Array.new
  activity_matrix.each do |activ|
    sc << ochiai(activ, output)
  end
  # fitness: discriminable SC
  top_sc  = (sc.sort!)[-1]
  top_cnt = sc.count(top_sc)
  low_sc  = sc[0..-2]
  sum_low = low_sc.inject {|sum,x| sum+x}
  return (top_sc - sum_low) / top_cnt if diff==:low
  return (sum_low - top_sc) / top_cnt if diff==:high
  #return (sum_low - top_sc) / (top_cnt + output.count(1))
  # favor. low num. of failures
end
```

Table 4.10: Examples for Freely evolved topologies

	200 Generations, 40 Activations N_p=200, N_t=3, P_c=0.5, P_m=0.001; **diff=:low**	
C_1	1111111111111111111111111111111101111111	0.97
C_2	1011111111111111110111111111111111111111	0.98
C_3	1011111111111111111111111111011111111111	0.98
C_4	1011111111111111111111111111111111111111	1.00
C_5	1011111111111111111111111111111101111111	0.98
O	1011111111111111111111111111111111111111	
	200 Generations, 40 Activations N_p=200, N_t=3, P_c=0.5, P_m=0.001; **diff=:high**	
C_1	11000111000000001100101011000101010010000	0.00
C_2	01000000011010000000010000010101110000001	0.00
C_3	0101100001000100001000100010010001100000	0.80
C_4	0111000000100001001000000100010000010010111	0.00
C_5	100000100110000011101000001110001010011000	0.00
O	000010000000001000010001000000010001100000	

which is invoking it individually. Translated to a real system it means, that if $C3$ is invoked from an external (unmonitored) system, it is exercising a different internal route, than if it was called from one of its monitored peers. In future work, we will assess to which extent missing observation points can inhibit proper diagnosis.

Freely Evolved Topologies with Fewer Failures.

Even when `diff` is set to `:low`, we can get useful results, when the fitness function favors individuals with low number of failures. We can amend the Fitness E shown in Table 4.9 by adding a scaling factor, i.e. `output.count(1)`, shown in the last line of the fitness function's code, penalizing individuals with high numbers of failures. Table 4.11 shows an example with a an activity matrix containing only

Table 4.11: Freely evolved topologies, fewer faults

200 Generations, 40 Activations N_p=100, N_t=3, P_c=0.5, P_m=0.001; **diff=:low**		
	fitness=0.081 (best random individual)	
C_1	0011010100111001111111111001010001000111	
C_2	0011110101001011000100111010010101110111	
C_3	0011111100001100111100001110000100110111	
C_4	0110010010001111010000011001100000100111	
C_5	0110101101000001000100110110110001111001	
O	0010111000110111000000100101010001101111	
	fitness=0.44 (best indiv. after 200 gen.)	SC_o
C_1	00101001001001000000001100000100000110000	0.316
C_2	00001110101011000011001000000010000010000	0.289
C_3	0000101000010011011100000000000000010000110	0.302
C_4	0010100001100000000000001110000000001100011	0.302
C_5	0110100010000001000100100011000000000100010	0.302
O	0000100000000000000000000000000000000000	

one failure.

Remarkable, again, is the high number of zeroes indicating that monitoring of inactivity is advantageous for high diagnosability. This is in line with our earlier results. However, the result shown in Table 4.11 also hints to another interesting topological issue. Even though, all components are activated together in case of the failing transaction, representing tight interaction of the components in the fault case, the activity matrix contains enough discriminative information in order to reach an unambiguous diagnosis. This is, again, a strong indicator that the ability of a monitoring topology to observe various combinations or patterns of component invocations will help in reaching unambiguous diagnoses. The other occasional activations lead to sufficiently diverse information in order to being able to separate the tight coupling of the components on failure.

From this observation, we can deduce that not only diverse coverage benefits diagnosability, but moreover, also distinct coverage. In other words, topologies with more diverse execution routes, covering distinct components, facilitate diagnosability. In the topology, this can be achieved through monitoring not only activation or non-activation of a particular entity, i.e. the fact that something has been used, but also through monitoring the context in which something has been used, i.e. incoming and outgoing combinations of activations. In other words, the fact that something has been covered through various routes, or in particular sequences, which can be monitored, has an influence on the diagnosability of a topology. In the future, we will take a closer look at the influence of the traces leading to an activity matrix, rather than merely the activations themselves.

4.4 Discussion

Revisiting the Research Questions

In the introduction, we asked ourselves *how genetic algorithms can be used to study the effects of the monitoring topology on the diagnosability of systems*. We will now

address this problem by providing answers to the two research questions formulated:

RQ4.1: How can genetic algorithms be used to optimize monitoring topologies for spectrum-based diagnosis? The topology of a system is represented by an activity matrix, whereby, for each observation point, it expresses whether that point has been activated. The coverage of all observation points can be expressed as a binary string, making a mapping to a GA-chromosome straightforward. For the fitness, we propose several approaches. First, a function that expresses the diagnosability of a monitoring topology, i.e., the extent to which all diagnoses carried out on an activity matrix coming from that topology, are correct diagnoses. In the fitness function, each component is set to be faulty per diagnosis. Then, the fitness function calculates to which extent all similarity coefficients combined from all runs represent correct and distinguishable diagnoses. This yields a value representing how well a topology facilitates the discovery of each potential fault in every component. This basic fitness function can be amend in order to address the different optimization criteria required in the different experiments, e.g. favor high or low differences in the SC_o, or favor output vectors with low number of failures.

RQ4.2: What are characteristics of monitoring topologies that are optimal for spectrum-based diagnosis? According to the fitness function, a topology is a “good" or a diagnosable topology, if it facilitates the detection of all faults in a system in an unambiguous manner. The application of GA hints at a number of routes to satisfying this fitness goal. Our results show that

- being able to invoke components in isolation is beneficial for diagnosability, because it helps separate component involvement in system executions better.
- adding observation points (monitors) in the system, and including the monitoring of inactivity, helps separating system executions, which also facilitates the diagnosability of the system.
- including monitoring of the system context (for example, external components from other systems, incoming and outgoing activations, etc.) can support diagnosability through incorporating different invocation routes.
- including tracing information which represents combinations or distinct patterns of component coverage, may suport SFL-based diagnosis.

All these items also raise the question of the optimal number of observation points for high diagnosability w.r.t. low monitoring overhead.

Lessons Learned

Besides the more general characteristics of diagnosable topologies stated above, the application of GA taught us a lot about the behavior of the SFL approach. It is interesting to see how a search heuristic cannot only help to provide solutions, but also point to issues, both known, and unknown.

In the initial assessment of our setup, the GA generated topologies with tightly coupled components. This was due to our poor fitness definition. We knew already that tight component interaction is bad for diagnosability of a topology, and the GA was, in fact, pointing to this issue, so that in subsequent experiments, the fitness function could be adjusted.

The fact that having fewer activations within a spectrum provides better information for SFL than more activations was not obvious initially. Creating monitoring topologies that lead to such observations is, therefore, an essential goal for future work.

Finally, from the last experiments we can deduce that the context of activity is an important factor in the calculation of a diagnosis. In other words, if a component is activated, which route did this activation take? We knew already that introducing more information into the calculation of the SC yields better diagnoses. But this points to very particular information to be included, i.e., the activation paths through the system. In future work, we will derive the activation sequences from the traces generated by the monitors and encode this in the activity matrix.

Threats to Validity

In this initial application of GA to studying the effects of topologies on diagnosability, we have used activity matrices instead of real topologies. An activity matrix represents component involvement in system transactions and must be regarded as a simplification of a topology. It does not explicitly express the links between components. We can, therefore, only infer very general characteristics of potentially diagnosable topologies.

In the experiments, we have only looked at a low number of observation points (monitors), and at a low number of observations (spectra). We are aware of the fact that the number of observations and observation points affect the achievable results, but we decided to treat the generation of variable numbers of observation points as a problem in its own right, to be addressed in the future.

4.5 Related Work

Literature describing the application of genetic algorithms to the optimization of topologies is abundant. For instance, Kumar et al. (Kumar et al., 1995) propose a general approach based on GA to design network topologies for distributed systems, in order to achieve network reliability; Madeira et al. (Madeira et al., 2005)

develop a computational model to optimize topologies of linear elastic structures with GA; the authors of (Granelli et al., 2006) use GA to optimize the topology of hardware circuit against parallel flows.

In software engineering, the authors of (Hadaytullah et al., 2010), (Räihä et al., 2008), and (Räihä et al., 2011) propose to apply multi-objective GA to automatically synthesize software architectures. The architectural patterns are used for mutations and the quality metrics are used as fitness function to assess each architecture. Their research results conclude that their approach of architecture synthesis based on GA is able to produce a set of reasonable architectural solutions. However, only two quality attributes, i.e. modifiability and efficiency, were considered in their approach to generate software architectures. Lutz (Lutz, 2001) use meta-heuristics to evolve good hierarchical decompositions. Decomposition is related to our problem of placing monitors at strategically optimal locations.

Harman (Harman and Clark, 2004) states that “metrics are fitness functions too". We acknowledge this by defining fitness functions for diagnosability. Kim and Park (Kim and Park, 2009) propose the application of reinforcement learning in self-managing systems. In particular, they mention software architecture. Our approaches are intended to contribute to self-adaptive and self-managing systems.

Piel et al. (Piel et al., 2011) apply spectrum-based fault localization techniques together with online monitoring to recover health information and pinpoint problematic components for self-adaptive systems. Abreu et. al. (Abreu et al., 2009b) present a diagnosis approach combining spectrum-based fault localization and model-based diagnosis techniques, which is able to locate multiple faulty components with relatively low cost.

4.6 Summary

In this chapter, we outlined how genetic algorithms can be used to study the effects of monitoring topologies on SFL-based diagnoses. We defined a simple one-to-one mapping between the chromosome of a genetic algorithm and an activity matrix to be used by SFL, plus several fitness functions representing diagnosability. Activity matrices were used as simplifying models for real topologies. Explorative experiments revealed a number of general characteristics of topologies that support diagnosability, and we learned to better understand how topology affects the calculation of diagnoses.

The vision of our research is that, eventually, we would like to be able to have a search heuristic generate the most optimal monitoring topology in terms of high diagnosability for any arbitrary existing system. In the future, therefore, we will have to look at how real topologies can be encoded for GA, instead of merely using activity matrices representing topologies. This can be done either with the help of a topology simulator(Section 2.2), or with real systems. Other issues to be addressed in the future are the inclusion of context information (derived from traces) in the

calculation of the diagnosis, and the inclusion of more monitors. This last aspect represents a multi-objective optimization problem in its own right, i.e. generate topologies for optimal diagnosability with minimal monitoring overhead.

5

Diagnosis Improvement Through Increased Monitoring Granularity

Due to their loosely coupled and highly dynamic nature, service-oriented systems offer many benefits for realizing fault tolerance and supporting trustworthy computing. They enable automatic system reconfiguration when a faulty service is detected. Spectrum-based fault localization (SFL) is a statistics-based diagnosis technique that can be effectively applied to pinpoint problematic services. However, SFL exhibits poor performance in diagnosing services which are tightly interacted. Previous research suggests that an increase in the number of monitoring locations may improve the diagnosability for tight interaction.

In this chapter, we analyze the trade-offs between the diagnosis improvement through increased monitoring granularity and the overhead caused by the introduction of more monitors, when diagnosing tightly-interacted faulty services. We apply SFL in a service-based system, for which we show that 100% correct identification of faulty services can be achieved through the increased monitoring granularity. We assess the overhead with increased monitoring granularity and compare this with the original monitoring setup. Our experimental results show that the monitoring at the service communication level causes relatively high overhead, whereas the monitoring overhead at a finer level of granularity, i.e. at the service implementation level, is much lower, but highly dependent on the number of monitors deployed.[1]

5.1 Background **59**
5.2 Problem Statement and Approach **62**
5.3 System Simulations **65**
5.4 Case Study **67**
5.5 Runtime Overhead **70**
5.6 Discussion and Lessons Learned **77**
5.7 Related work **80**
5.8 Summary **82**

[1]This chapter comprises our findings submitted to the *Software Quality Journal*. An earlier version of this chapter was published at the *7th International Conference on Software Security and Reliability (SERE'13) with the most distinguished paper award* (Chen et al., 2013a).

The dynamic features inherent to service-oriented software systems, such as online deployment of services, and runtime reconfiguration and evolution, facilitate fault tolerance mechanisms in a natural way, and it makes the handling of emerging problems straightforward. If a faulty service misbehaves during operation, it can be exchanged for another healthy service through simple runtime reconfiguration (Bennett et al., 2000; Canfora and Di Penta, 2006). However, before a service may be exchanged, it must be determined with certainty that this service, indeed, represents the root cause of the failing system, and that it is not merely propagating an error from somewhere else (Mohamed and Zulkernine, 2008). Even though service-oriented systems provide all the ingredients necessary to recover from and adapt to operation time failures (Di Nitto et al., 2008), adequate runtime diagnosis approaches that accurately identify a faulty service are still missing. Diagnosis for services has been proposed in the past (Yan and Dague, 2007; Yan et al., 2009), but the techniques are mainly based on static system modeling, disregarding the dynamic nature of service-based systems.

Chapter 3 demonstrates that spectrum-based fault localization (SFL), which is a statistics-based diagnosis technique, can be applied effectively to pinpoint faulty components in service-based systems. SFL works by automatically inferring a diagnosis from observed symptoms (Abreu et al., 2009a). The diagnosis is a ranking of potentially faulty components, i.e. the services in a service-based system, and the symptoms are observations about service involvement in system activation, i.e. the service transactions, plus pass/fail information for each transaction (Chapter 3 and (Gonzalez-Sanchez et al., 2011)). SFL is based on the assumption that a service is more likely to be faulty, if it participates more in failing transactions, and it mimics how a human diagnostician would exonerate parts of a system that cannot be used to explain a particular failure observation.

Although SFL represents an adequate technique for diagnosing faulty services, experiments performed for our previous work in Chapter 3 show that incorrect diagnoses are more likely, if services are tightly interacted. In other words, if a service S_1 always invokes another service S_2 and one of the services is faulty, the diagnosis would be such that both services S_1 and S_2 will be convicted, leading to incorrect or inconclusive diagnoses. In a traditional setting with a human diagnostician, this is not so much of an issue. Since it would mean that more services would have to be inspected, in order to determine the true root cause of failure, thereby merely increasing the residual diagnosis cost (Gonzalez-Sanchez et al., 2010a). However, in the case of a service-based system acting fault-tolerance autonomously, it would mean that reconfiguration or other self-healing activities would have to be applied to more suspects, thereby unnecessarily treating services that are actually healthy.

Careful analysis of the experiments performed for Chapter 3 reveals that the difficulty of tight coupling for the SFL approach can be resolved either by the ar-

chitecture of the system and how services interact or by the granularity of the observations used for SFL. Whereas, in the first instance, it would be rather difficult to try and rearrange the architecture in order to decouple services for any individual system configuration; in the second instance, it would be relatively easy to introduce more monitoring points in the architecture, and thus increase the level of monitoring granularity, that would be sufficient to support the calculation of a conclusive diagnosis.

As a consequence, the *goal of this chapter* is to explore the effects of changing the level of monitoring granularity, and assess its impact on the calculation of a diagnosis and on the performance of a running service-oriented system. In this chapter, we concentrate on the following concrete research questions:

RQ5.1: How and to which extent does the monitoring granularity affect the calculation of a diagnosis with spectrum-based fault localization?

RQ5.2: How can we increase the monitoring granularity?

RQ5.3: What is the overhead caused by the monitoring of various levels of granularity?

We make the following contributions. We describe an approach and implementation for increasing the monitoring granularity in services, and show how this can improve the accuracy of diagnosing faulty services. We use a SFL simulator to study the effects of changing the monitoring granularity on the calculation of the diagnosis in many different system configurations. We assess the overhead of our approach and implementation in a real case study and discuss its implications.

The remainder of this chapter is organized as follows: Section 5.1 presents the research field and techniques related to our approach. Section 5.2 outlines why tight service interaction inhibits the calculation of a diagnosis by SFL, and why increased monitoring granularity is adequate to alleviate this problem. Section 5.3 introduces the SFL Simulator and explains how it can be used to assess the performance of our proposed approach quickly. Section 5.4 describes the case study used to assess our proposed approach. Section 5.5 presents the experiments measuring the runtime overhead caused by the monitoring of different levels of granularity. Section 5.6 discusses the experimental results and the limitations. Finally, Section 5.7 presents related work and Section 5.8 concludes the chapter.

5.1 Background

SFL for Service-based Systems

Applying SFL in service-based systems requires the SFL concepts to be adapted to the service context. This has implications in terms of the component granularity, system activation, component coverage and the verdicts. The service represents the

natural component granularity. It is the basic unit that can be restarted, exchanged, or otherwise treated, in case an error is detected. Alternatively, a service operation, which represents a business functionality of a service, may denote a finer level of granularity.

Due to the loosely-coupled nature of services, activation in service-based systems is not so obvious. A service instance may serve many application contexts. In other words, a service will not be exclusively activated from within one application context, but from a potentially arbitrary number of other applications operating in other contexts, i.e. the contexts of all clients that depend on a service. Applying SFL in a service-based system, therefore, requires a system activation to be made explicit through a unique transaction ID, which separates the service activations of different application contexts.

Component involvement in transactions is typically measured through coverage tools. However, since there is no single controlling authority that can produce service coverage information, involvement of a service in a transaction must be produced differently. To apply SFL in service-based systems requires dedicated monitors, which observe the service communication and associate the services or their operations with the corresponding transactions invoking the services or their operations. This can either be done by the services themselves or through modern service frameworks. For example, Apache's Axis2, Red Hat's JBoss, or eBay's Turmeric come well-equipped with extensive monitoring facilities that can be adopted to producing service involvement information.

A transaction's pass/fail information comes from an oracle. Runtime errors, exceptions, warnings and logs are natural choices for realizing oracles in service-based systems. They are readily available through the platforms managing the communication between services, or they are initiated through the business logic, i.e., the services themselves.

Implementation of SFL for Service-based Systems

This section presents the implementation of the aforementioned SFL concepts for service-based systems. Firstly, the service operation is set as component granularity for diagnosis, because it permits a more fine-grained diagnosis. Secondly, activation of the service-based system used for our experiments is outlined. Thirdly, online monitoring is required, in order to recover the service involvement in transactions, and in order to calculate the verdicts. In addition, a diagnosis engine is built in order to maintain the SFL activity matrices and calculate the diagnoses. The organization of our SFL implementation for service-based systems is presented in Fig. 5.1, and it is briefly summarized in the following (more details in Chapter 3).

Typically, services would be activated at the application interface through user interaction. However, in our case, system activation is automated through various third-party tools for evaluation purposes, or through custom-built clients for assessing overhead. There are some existing tools, which provide easy access to

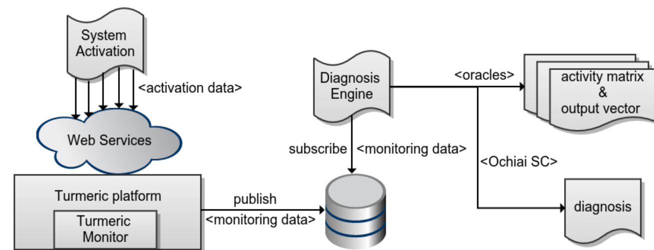


Figure 5.1: Monitoring and diagnosis architecture based on Turmeric

services, such as SoapUI and JMeter. Such tools are used to create SOAP messages and execute them automatically, thereby mimicking real user interaction coming from different application contexts. On top of that, our service-oriented system is built on eBay's open source service framework Turmeric[2]. This framework provides stub code for each service, which allows developers to build customized client applications to invoke the services.

Turmeric also provides many inbuilt features to support the (online) collection of system data required for applying SFL in service-based systems. These features facilitate the integration of online monitoring code, in order to record the component coverage for SFL with minimum amendments, resulting in a slender monitoring design. The message-handling mechanism of Turmeric is based on a specific pipelined architecture. All incoming and outgoing messages will go through the pipelines and will be processed by a group of default handlers. The default handlers can be extended by adding custom-built handlers for monitoring, i.e. our Turmeric monitors, dedicated to obtaining transaction information required by SFL. For each service message, the Turmeric monitors will parse the message context to get the transaction ID, the message content, the service and operation names and other information referring to the transaction. The custom-built monitors in the pipelines publish to a Redis in-memory data base instance[3] in order to forward the collected data asynchronously to the diagnosis engine. The diagnosis engine subscribes to the respective monitoring data via Redis and performs the SFL calculations offline. That way, the monitoring data from messages belonging to the same transaction can be easily traced, resulting in the involvement of service operations in a unique transaction to be used in the diagnosis.

Verdicts are generated based on the monitoring data from Turmeric monitors. A set of oracles is applied to determine the result of each transaction with pass or fail, based on the message content. The monitors also check upcoming exceptions, or other noteworthy events and outcomes during system operation. Any of these noteworthy occurrences can be associated with a unique transaction ID, and used to judge the transaction.

The actual diagnosis is conducted offline in a diagnosis engine. It is designed

[2]https://github.com/ebayopensource/turmeric-runtime

[3]We use the *publish/subscribe* feature for optimal performance; see `http://redis.io/`.

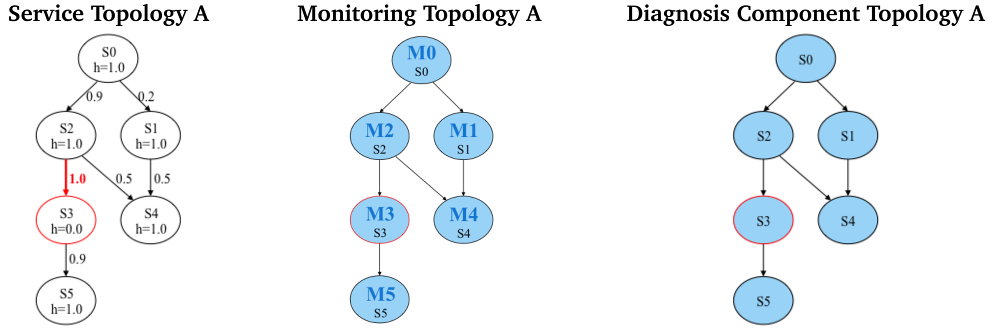


Figure 5.2: Example topology illustrating tight service interaction

Table 5.1: Activity Matrix for Topology A

Component	Activity for Topology A (fatal failure)	Ochiai SC
S5	00000000000000000000	0.000
S1	00000000110000000100	0.280
S4	10111000000110001110	0.728
S0	11111111111111111111	0.922
S3	**10111011101111111111**	**1.000**
S2	**10111011101111111111**	**1.000**
Output	10111011101111111111	

as a separately operating application that collects all monitoring data to get service activities and produce verdicts by applying oracles. Activities and verdicts are transformed into an activity matrix and an output vector for further calculation of a diagnosis. This implementation is summarized in Fig. 5.1.

5.2 Problem Statement and Approach

One of the main targets of this chapter is to study how tight service interaction inhibits the calculation of a diagnosis, and how adjusting the monitoring granularity can help overcome this limitation. In order to explain the tight service interaction problem we make use of a *service topology*. An example can be found on the left-hand side in Figure 5.2. A topology is created by defining a number of components. Each component is defined by the component name, component health, and failure probability. Health denotes the probability that a component will not produce an error when it is executed: 1.0 represents a healthy component, while a value in the range (0.0, 1.0) represents a faulty component with intermittent fault behavior. A health value of 0.0 denotes no fault intermittency, i.e., the component will always produce an error if activated. Components in a topology can be connected through defining a link between them with an associated invocation probability.

Besides the service-topology, we also look at the *monitoring topology*, which is basically a representation of where the monitors are in the service topology. In the most basic case of Figure 5.2, where each component has exactly one monitor, the monitoring topology corresponds to the service topology.

The *diagnosis component topology* then represents a virtual service topology in

Table 5.2: Activity Matrix for Topology B

Component	Activity for Topology B (fatal failure)	Ochiai SC
S5	00000000000000000000	0.000
S1	01000000000001010001	0.471
S4	11001001000111110100	0.745
S0	11111111111111111111	0.949
S2	11101111111111111111	0.973
S3	**11101111110111111111**	**1.000**
Output	11101111110111111111	

which the components of the service topology are split up in subcomponents in case multiple monitors per component are placed. This diagnosis component topology can discern multiple calling paths within a component in the service topology.

The Problem of Tight Service Interaction

First, we explain how tight interaction aggravates diagnosis. Consider the topology on the left-hand side in Fig. 5.2, which is comprised of six services, S_0 – S_5, with service S_3 being the faulty one with low health (h=0.0). All other services are set to be 100% healthy. Services S_2 and S_3 are tightly interacted, indicated through the 1.0 invocation probability between them. It means once service S_2 is invoked, service S_3 will also be invoked, leading to the same activity status for the two services. This creates a problem for the diagnosis, when each service gets only one monitor, as illustrated in the monitoring topology shown in the middle of Fig. 5.2. There is a one-to-one mapping between the service topology and the topology of the monitors, hence the topology of the diagnosis components, shown on the right-hand side of Fig. 5.2. There is a one-to-one mapping between the service topology and the topology of the monitors, hence the topology of the diagnosis components, shown on the right-hand side of Fig. 5.2.

The activity matrix and diagnosis results for this monitoring setup (produced with the SFL simulator[4], described in Section 2.2) are presented in the table in Tab. 5.1. Due to the tight interaction between services S_2 and S_3, the diagnosis not only convicts the real faulty service, S_3, but also its tightly-interacted peer, the service S_2. As indicated by the Ochiai Similarity Coefficients (SC) in Tab. 5.1, the two services are assigned the same values (SC = 1.0), and thus, the same rank in the diagnosis. In this diagnosis, both services are, in fact, treated as one single diagnosis component. This ambiguity would bring extra effort to service maintainers to identify the real faulty service, however, in case of automatic service recovery, both services would have to be treated, thereby treating an otherwise healthy service (S_2). Therefore, only the results, which rank the real faulty service highest and uniquely in the diagnosis, can be considered correct. In this example, tight interaction between services produces an ambiguous result, however, in some cases, tight interaction can also lead to incorrect diagnoses.

[4]https://github.com/SERG-Delft/sfl-simulator

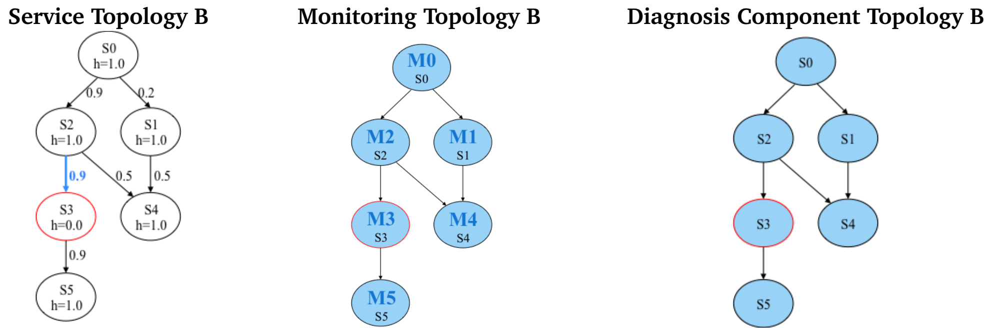


Figure 5.3: Example topology illustrating potential solution 1

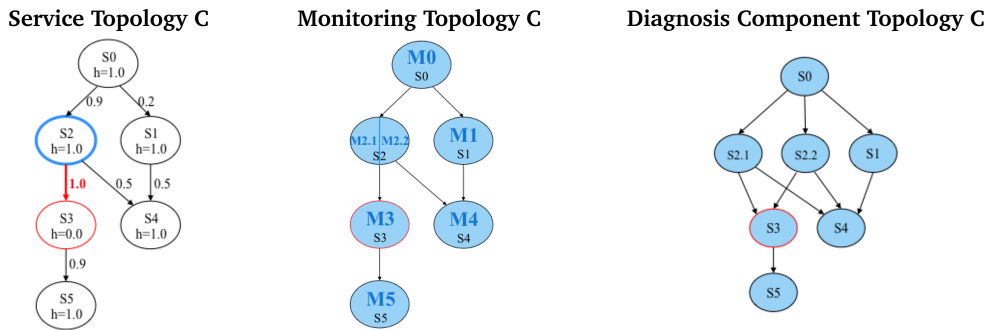


Figure 5.4: Example topology illustrating potential solution 2

Table 5.3: Activity Matrix for Topology C

Component	Activity for Topology C (fatal failure)	Ochiai SC
S5	00000000000000000000	0.000
S1	00000000000000010100	0.000
S2.1	10001101001001000000	0.679
S4	00000001011111011000	0.686
S0	11111111111111111111	0.806
S2.2	10110001111111001000	0.920
S3	**10111101111111001000**	**1.000**
Output	10111101111111001000	

Solving Tight Service Interaction – Potential Solution 1

A possible solution to deal with this insufficiency of diagnosis in the case of tight service interactions would be to reduce the invocation probabilities between such services. In other words, create a system, in which not every invocation of service S_2 will subsequently lead to the invocation of service S_3. Service Topology B in Fig. 5.3 illustrates such an architecture. The invocation probability between the two initially tightly-interacted services is reduced to 0.9. Without having to change the monitoring setup, this slight adjustment in the invocation probability leads to enough decoupling of the services, and to the introduction of sufficiently more discriminative information in the observations. Thus, a correct diagnosis can be calculated in the related activity matrix for the Diagnosis Component Topology B in the table shown in Tab. 5.2.

Solving Tight Service Interaction – Potential Solution 2 (Our Approach)

In real systems, the invocation probabilities between individual services cannot be adjusted arbitrarily, because they are determined by the business logic, and the input parameters coming from the external system context, i.e. the system's usage profile. In order to retrieve similar discriminative power in the observations, a feasible adjustment in the monitoring topology must be invented that leads to similar results as shown for Service Topology B. Experiments with the SFL Simulator suggest that this may be achieved through increasing the number of observation points (monitors) in the service topology. This boils down to logically splitting services into sub-components, or simply adding components, and associating individual monitors to these sub-components. This increases the level of detail, i.e. the monitoring granularity used for the similarity coefficients, and helps discriminate service invocations that follow different internal invocation paths. By defining a monitoring topology that separates services into finer-grained sub-components, we retrieve finer-grained coverage information, and finer-grained potential communication paths between the sub-components, with potentially different invocation probabilities between them. The assumption that we do make here is that we have access to the internals of the services to actually implement this finer-grained monitoring.

This increase in the monitoring granularity is illustrated in Fig. 5.4. Here, Service Topology C corresponds to Service Topology A shown in Fig. 5.2, with S_2 and S_3 being tightly interacted, and S_3 being the faulty service. In contrast to Monitoring Topology A, the new Monitoring Topology C is changed in such a way, that, instead of using only one monitor, two monitors ($M_{2.1}$ and $M_{2.2}$) are associated with service S_2. Each of the monitors is in charge of different paths through Service S_2. So, in terms of monitoring, Service S_2 is split into two sub-components: $S_{2.1}$ and $S_{2.2}$, as shown in the diagnosis component topology in Fig. 5.4. Both sub-components lead to two separate observable paths from S_2 into S_3, and the corresponding activity matrix is also changed, as shown in Tab. 5.3. In this way, the diagnosis is able to produce a correct and unambiguous result. This example illustrates that adding more observation points can improve diagnosis for service-oriented systems with tight interactions. However, whether, and to which extent the increasing of monitoring granularity can affect diagnosis depends on careful selection of the observation locations. This requires further investigation when performing a case study (Section 5.4)

5.3 System Simulations

Simulation Results

To assess our approach in a more realistic setup, we imitated our case study system with the SFL Simulator. In contrast to the topology shown in Fig. 2.3, which is only

Table 5.4: Simulation Results for Service Diagnosis

Services	Component Granularity	# of Activations	Diagnosis Correct	Diagnosis Incorrect	% Correct Diagnosis
ExchangeCurrencyService	i_1 Interface	50	8	42	16%
	i_2 Sub-comp	50	39	11	78%
OrderProcessorService	i_1 Interface	50	13	37	26%
	i_2 Sub-comp	70	47	23	67%

Table 5.5: Reasons for Incorrect Diagnoses in Simulation

Services	Component Granularity	Incorrect Diagnoses	Fault not Activated	Other Reasons
ExchangeCurrencyService	i_1 Interface	42	16	26
	i_2 Sub-comp	11	5	6
OrderProcessorService	i_1 Interface	37	5	32
	i_2 Sub-comp	23	5	18

displaying top-level services (due to space limitations), in the simulator, we used a more detailed system model that includes the service interface level. This follows the original design of the case study system presented in Section 2.2. In addition, the link probabilities used in the simulations are based on the service implementation logic plus test data applied. The system health (or failure intermittency) is determined based on the number of fault activations during testing of the real system.

In the original experiments, two services could be identified to exhibit the problem of tight service interaction, i.e. the *ExchangeCurrencyService* and the *OrderProcessorService*, resulting in incorrect diagnoses. The results of the simulations performed for these two services are shown in Table 5.4. The simulations are based on two levels of detail. The first level of granularity assessed is the service interface level (indicated as i_1 in Table 5.4), and this corresponds to our original experiments described in Chapter 3. The second level is more detailed and separates service interfaces into finer grained sub-components (indicated as i_2 in Table 5.4). The ExchangeCurrencyService is split into 5 subcomponents, and the OrderProcessorService is into 7 subcomponents. The sub-components, which are associated with individual monitors, are determined following roughly the main execution paths through these services. Their respective invocation probabilities defined in their links are derived experimentally from the original system in the case study. Since the simulation is made for single-fault case, i.e., only one component/subcomponent can be set as faulty in one activation, so the number of activations in the simulation (Table 5.4) is set to 50 and 70 for two services, respectively, in order to retrieve sufficient fault coverage.

The low values for correctly performed diagnoses for granularity i_1 shown in

Table 5.4 illustrate the poor performance of SFL for tightly coupled services. A diagnosis is considered to be correct, if the true faulty component is correctly and unambiguously identified by SFL. In the initial setup (with interface-level granularity, i_1), this can only be achieved in 16% and 26% of the cases for the two tightly coupled services. The simulation results for the finer-grained level of monitoring granularity (i_2, shown in Table 5.4) are much improved, up to 78% and 67%. However, the improvement is poorer than expected. In fact, they are worse than the results from the experiments performed for the real case study described later (Table 5.8). This requires some explanation:

1. Compared to the case study, fewer faults are activated in the simulation (as shown in Table 5.5), leading to missing diagnoses. The chance of executing some faults is low through the combination of failure and invocation probabilities defined in the simulation. In other words, some faults that are activated in the case study are not activated in the simulation.

2. Even though the number of activations corresponds to the real system, the random activations between the components is more diverse. The simulation uses random invocations according to predefined probabilities in order to exercise the topology. The probabilities are retrieved experimentally from the real case study, but they cannot absolutely reflect the usage profile imposed by the real test cases. This leads to statistically significant deviations of the executions in the simulation compared to the real system.

3. The monitoring granularity in the real case system is increased compared with the simulation (see Section 5.4). The simulator allows to define topologies with finer-grained sub-components, however, estimating the link probabilities and health values of these finer-grained sub-components becomes increasingly difficult.

All in all, the simulator always produces worse results when compared to the real case study. This is mainly due to the fact that it builds system topologies based on probabilities, i.e., an approach being tested positive in simulation is more likely to receive positive results in real system. Therefore, using the simulator for trial test can easily assess an approach without implementing it in a real system. In our experiment, the simulations confirm the positive effect of introducing more observation points for the calculation of the diagnosis. In the following section, we describe how our approach is evaluated in a real system.

5.4 Case Study

Conducting the Case Study

Because the focus in this chapter is on tight service interaction, in the case study, again, we look at the two services, the *ExchangeCurrencyService* and the *OrderPro-*

Table 5.6: Active Mutators in the Experiment

ID	Mutator	Error in the system
1	Negate Conditionals	wrong internal state or response, null or runtime exception
2	Return Values	wrong response, null or runtime exception
3	Conditionals Boundary	wrong internal state or response
4	Void Method Call	wrong internal state
5	Math Mutator	wrong internal state

Table 5.7: Mutators used in the two Tightly Interacted Services

Services	Mutators (from Table 5.6)	# of Mutations
ExchangeCurrencyService	1	5
(24 mutated versions)	2	7
	4	12
OrderProcessorService	1	15
(41 mutated versions)	2	1
	3	1
	4	23
	5	1

cessorService, which present tight interactions with other services. We apply the PIT mutation tool in order to create 65 faulty system versions, 24 faulty versions for the *ExchangeCurrencyService*, and 41 faulty versions for the *OrderProcessorService*. Table 5.6 summarizes the type of mutations applied with PIT, and it briefly states the purpose of each mutator used, and the error it generates in the system. Table 5.7 illustrates the kind of mutators applied to the two services. The different numbers of mutations per mutator come from the presence or absence of specific code features in the service implementations that PIT manipulates.

For each of the 65 faulty system versions, we use JMeter to execute 48 web service requests as test scenarios in order to cover all service operations. Upon completion of all transactions for one faulty system version, the diagnosis engine is invoked to parse the monitoring data, identify the failures in the system, and create an activity matrix with an output vector. Then, it is assessed whether the resulting diagnosis pinpoints the service correctly that contains the seeded fault. The whole experiment is designed for the single fault case. We ensure that each of the 65 versions of the system contains only one fault, either in the *ExchangeCurrencyService* or in the *OrderProcessorService*.

The conduction of the case study is split up into two instances, i_1 and i_2. In instance i_1, we invoke the original case system with monitoring enabled at the service interface level of granularity. The monitoring is provided through the Turmeric framework, mentioned in Section 5.1 and detailed in Chapter 3. In instance i_2, we invoke the same system and use the same Turmeric-based monitoring. Additionally, we also put monitors in the service implementation codes at the code block

level of granularity. Basically, we split the service implementation into several code blocks, and put an observation point at the end of each block. The observation point is also a Redis-based publisher. Once a code block is executed to the end, the ID of the code block will be published to Redis. Based on the time sequence, the application is able to associate the monitoring data from the code block monitors with the transaction information from Turmeric monitors. We determine the code blocks based on the internal control-flow structure of the service implementations. In some cases, we separate the blocks for better isolation of tightly-interacted code sections. This results in 10 monitored sub-components for each of the two services under consideration. That way, we are able to increase the number of observation points in instance i_2 to the finer level of granularity required for correct diagnoses. The additional monitoring introduces more and more diverse coverage information, which we expect will yield better suited activity matrices, thus, leading to better diagnoses. The results of these experiments are presented in the following sub-section.

Case Study Results

Table 5.8 and Table 5.9 summarize the results of the case study for both instances, i.e. i_1 for service interface monitoring granularity and i_2 for code block monitoring granularity. Table 5.8 shows the correctness of diagnoses at both levels of monitoring granularity for each faulty service version. A diagnosis is considered correct, if the faulty service or one of its sub-components is ranked top, and no other service receives the same ranking, i.e. the diagnosis is correct and unique.

The improvement of the finer-grained monitoring granularity over the original coarser-grained granularity is substantial. Both services with incorrect diagnoses in our original case study can now be diagnosed correctly and unambiguously as the faulty services to a very high degree, i.e. 92% and 90% shown in Table 5.8. Actually, the faults injected in both services can always be diagnosed correctly, leading to 100% correct diagnoses. This becomes apparent when we look at the reasons for the incorrect diagnoses shown in Table 5.9. In the first instance, i_1, 19 plus 9 out of the total number of incorrect diagnoses of the two services produced wrong results because of tight interaction on failure. This represents our original problem, and the table indicates that it can be resolved entirely through increasing the monitoring granularity for the considered services in the second instance, i_2. In both instances, i_1 and i_2, 2 plus 4 out of the total number of incorrect diagnoses are due to the faults in the services *not* being activated. In other words, in these cases no test execution was able to cover the faults introduced through the mutations. In general, diagnosis can only be initiated when a fault is actually detected. This is not attributable to our diagnosis technique, but a fundamental problem of all coverage-based quality assurance approaches.

Therefore, we can claim that all faults can be diagnosed correctly and unambiguously in our case study, if they can be detected, i.e. they are propagated into

Table 5.8: Experimental Results for Service Diagnosis

Services	Component Granularity	# of Mutations	Diagnosis Correct	Diagnosis Incorrect	% Correct Diagnosis
ExchangeCurrencyService	i_1 Service Interface	24	3	21	13%
	i_2 Code Block	24	22	2	92%
OrderProcessorService	i_1 Service Interface	41	28	13	68%
	i_2 Code Block	41	37	4	90%

Table 5.9: Reasons for Incorrect Diagnoses in Experiment

Services	Component Granularity	Incorrect Diagnoses	No Acti-vation	Tight Interaction on Failure
ExchangeCurrencyService	i_1 Service Interface	21	2	19
	i_2 Code Block	2	2	0
OrderProcessorService	i_1 Service Interface	13	4	9
	i_2 Code Block	4	4	0

failure. The lower values of 92% and 90% shown in Table 5.8 are a consequence of intermittent fault behavior of the services, a common property of software.

5.5 Runtime Overhead

Experimental Setup

An important aspect of our proposed diagnosis technique is the runtime overhead it imposes on the service-based system. Since the diagnosis engine is detached from the executing system, we focus on the overhead of the runtime monitoring required for SFL. In the experiments, we aim to measure the time overhead caused by the code block monitor, the time overhead caused by the Turmeric monitor, and the time overhead caused by the data-logging (publishing to Redis) in the Turmeric monitor.

We chose a set of requests based on diversity in service interactions that they will create, to invoke the ExchangeCurrencyService (ECS) and the OrderProcessorService (OPS). Both services have four fundamentally different associations with other services, e.g. the BusinessAccountService or the ConfigurationService, which are interesting for performance measurements. Additionally, we also add the BusinessAccountService (BAS) to the overhead experiments, in order to measure overhead under diverse scenarios. This service does not invoke any other subsequent services. That way, we can collect performance data for a range of different scenarios, i.e. with a variable number of services involved in various shorter and more extensive transactions.

The service-based system is repeatedly invoked with diverse requests and under

various monitoring configurations set up. For each invocation, we measure the end-to-end response time for the request. Then we compare the response time of the exactly same request under different monitoring setups. Therefore, we are able to observe the time overhead caused by Turmeric monitor or code block monitor.

For service activation, we used self-created service clients to invoke the services, instead of JMeter (which we used in the case study described in Section 5.4). The reason is that service clients are able to produce more reliable performance measurement. When we compare the standard deviations of 15 requests over 1000 runs for both JMeter and self-developed service clients, it becomes apparent that for 12 requests the spread obtained from our own service clients is much smaller than when using JMeter. These results are shown in Table 5.10. Eventually, we decided to drop JMeter in favor of our own developed clients.

Table 5.10: Standard Deviation of Experimental Results in Milliseconds

Tool	BAS_1	BAS_2	BAS_3	BAS_4	BAS_5	BAS_6	BAS_7	ECS_1	ECS_2	ECS_3	ECS_4	OPS_1	OPS_2	OPS_3	OPS_4
Client	3.383	7.501	16.498	4.165	9.906	14.360	9.346	178.954	16.622	21.408	12.340	99.929	22.185	37.281	26.561
Jmeter	11.108	28.237	22.445	21.238	32.805	42.031	47.468	209.220	9.143	26.714	13.545	113.760	28.661	23.106	19.369

Overhead Results

Table 5.11 shows the average response times for activating the *ECS* and *OPS* services 1000 times. The requests to both the ECS and OPS services may involve other services to complete. In other words, the request will initially invoke the ECS or the OPS, but the invoked service will continue to call other services, in order to complete a transaction. Thus, part of end-to-end response time from the ECS or OPS services can be attributed to the communication between all involved services. The total number of invoked Turmeric monitors depends on the number of involved services. When the Turmeric monitors are enabled, a request to a service will activate two Turmeric monitors, namely (1) one at the side of service request and (2) the other one at the side of service response. If the first service invokes another subsequent service, four additional Turmeric monitors will be activated to handle the message at (1) the side of the client request for the invoking service, (2) the side of service request for the invoked service, (3) the side of service response for the invoked service, and (4) the side of client response for the invoking service. Table 5.11 lists the number of activated Turmeric monitors for each service request. Among the listed requests, ECS_2 only gets two Turmeric monitors, that is because this request only invokes the ECS, it does not make the ECS invoke other services. When code block monitors are enabled in the system, there will be 10 code block monitors deployed for each of the two services, in order to improve the diagnosis accuracy for the services as detailed in Sec 5.4. However, different requests will activate different parts of service implementation, so that different

code block monitors will be invoked. The numbers of actually invoked code block monitors for each request are also listed in Table 5.11.

The four center columns in Table 5.11 termed "Monitors", present the average response times for each service request to the service-oriented system according to four monitoring strategies, i.e. all monitors disabled ("None"), only code block monitors enabled ("Code Block"), only Turmeric monitors enabled ("Turmeric"), both monitoring strategies enabled ("Turmeric & Code Block"). Notable are the relatively long response times for the requests ECS_1 and OPS_1. Based on a further investigation into network traffic during an experiment with Wireshark[5], we observed that the first request that makes a service to invoke another new service always consumes extra overhead. Since for the first request the service needs to establish a connection to the other service, and the following requests can directly reuse the connection if they are invoking the same service and the connection data is still buffered in the system memory. Both ECS_1 and OPS_1 requests are the first ones that the ECS and OPS services start with, respectively, and both requests invoke a large set of services as compared with their following requests. Therefore, the response times from both requests are much longer.

The three columns on the right-hand side in Table 5.11, termed "Impact (%)", show the impact of monitoring overhead for various monitoring setups compared to the system without any monitoring at all ("None"). The values indicate that Turmeric monitoring causes the most overhead in the system, while the overhead from code block monitoring is minute and may be ignored. An outlier case is the service request ECS_2, in which the impact from only Turmeric monitors is slightly larger than the impact from both Turmeric and code block monitors. In addition, we also observed two negative impact results from the service request ECS_4 and OPS_2. They are caused by the limitation of overhead measurement in our experiments, which is discussed in Sec. 5.6.

The overhead results presented in Table 5.11 are different from the results obtained in our previous overhead experiments outlined in our earlier article (Chen et al., 2013a). In this other article, the experiments were only aimed at getting an initial feeling of the potential overhead caused by various monitoring strategies, and we had to circumvent a few flaws in the implementation. The monitors were not decoupled from the data base maintaining the activity matrices, thereby adding considerable overhead through a sub-optimal synchronous implementation. Moreover, earlier we used the EMMA coverage tool[6] for realizing the code block monitors. However, it also causes overhead in itself, because it uses code instrumentation, plus coverage information could only be generated when the application server was shutting down, which lead to an awkward data collection procedure at the end of each experiment. Both implementation issues are now being resolved by using the publish/subscribe facility of Redis. Now, coverage information is simply

[5]http://www.wireshark.org/

[6]http://emma.sourceforge.net/

published to Redis the moment it is available, and a monitor is realized through a single ultra-fast Redis operation. In our opinion, the application of an in-memory publish/subscribe tool like Redis represents an optimal monitoring solution.

Table 5.11: Average End-to-End Response Time from ECS and OPS Services in Milliseconds over 1000 Transactions

Service Request	# of		Monitors				Impact (%)		
	Turm. Moni.	C.B. Moni.	None	Code Block	Turmeric	Turmeric & Code Block	Code Block	Turmeric	Turmeric & Code Block
ECS_1	14	6	2996.034	3002.367	3055.052	3065.618	0.21%	1.97%	2.32%
ECS_2	2	2	49.664	50.657	56.928	56.927	2.00%	14.63%	14.62%
ECS_3	14	5	72.58	74.456	118.256	120.189	2.58%	62.93%	65.60%
ECS_4	10	4	47.577	47.357	66.477	66.878	-0.46%	39.72%	40.57%
OPS_1	18	8	870.442	878.675	987.537	995.058	0.95%	13.45%	14.32%
OPS_2	18	7	135.504	130.494	177.714	180.371	-3.70%	31.15%	33.11%
OPS_3	18	8	310.94	320.227	351.423	353.64	2.99%	13.02%	13.73%
OPS_4	18	8	147.765	152.587	202.53	206.669	3.26%	37.06%	39.86%

The overhead measurements shown in Table 5.11 are also influenced by communication between several involved services which leads to a large spread for the overhead values measured. Furthermore, the number of code block monitors is fixed for the concern of diagnosis. We conduct a similar experiment with the *BAS* service, because the requests to the BAS service will not cause it to invoke subsequently associated other service(s). This experiment helps us foresee the likely impact of inter-service communication overhead. For the request to the BAS service, two Turmeric monitors handle the service messages at the side of service request and service response, respectively. When code block monitoring is enabled, we deploy different numbers of code block monitors in various service interfaces of BAS, in order to discover the relation between the number of code block monitors and the overhead they cause. For instance, the request *BAS_1* will invoke a service interface, which contains 10 code block monitors, and the request *BAS_3* will invoke another service interface with 100 code block monitors. The number of activated monitors for each request to the BAS service are listed in Table 5.12.

Table 5.12 presents the average end-to-end response times of 1000 invocations of BAS. Since the requests only invoke one service, the response times are much lower than those found in Table 5.11, with the exception of the first service request (*BAS_S*). The BAS_S request invokes the same service interface as the request *BAS_1*, however, it is the first request that the service client starts with in each experiment. As the first request in the whole experiment, it requires the service client to load the runtime libraries offered by the Turmeric platform to initialize the communication with a Turmeric service, and it establishes the connection to the derby database that our service-oriented system is using. These two parts consumes the major part of the time overhead from the BAS_S request. Due to the unreliable deviation caused by the initialization step, we exclude the results from the BAS_S

request in the following analysis.

The impact percentages shown in Table 5.12 expose more details about the monitoring overhead. The impact through Turmeric monitoring is still obvious to see. However, the impact of code block monitoring increases with the number of code block monitors, which is to be expected. The overhead of a single code block monitor is relatively low and may be ignored. However, using many monitors, i.e. up to 100, in the same service, increases the overhead from the code-block monitors to values similar to the ones exhibited by the Turmeric monitors.

Table 5.12: Average End-to-End Response Time from BAS Service in Milliseconds over 1000 Transactions

Service Request	# of Turm. Moni.	C.B. Moni.	Monitors None	Code Block	Turmeric	Turmeric & Code Block	Impact (%) Code Block	Turmeric	Turmeric & Code Block
BAS_S	2	10	1113.402	1146.469	1309.721	1315.575	2.97%	17.63%	18.16%
BAS_1	2	10	12.967	15.278	22.027	24.165	17.82%	69.87%	86.36%
BAS_2	2	1	45.087	45.851	60.424	60.606	1.69%	34.02%	34.42%
BAS_3	2	100	34.709	45.985	47.437	59.931	32.49%	36.67%	72.67%
BAS_4	2	10	28.63	30.229	34.876	35.619	5.59%	21.82%	24.41%
BAS_5	2	1	49.45	48.868	53.709	54.341	-1.18%	8.61%	9.89%
BAS_6	2	10	47.722	50.738	63.41	66.886	6.32%	32.87%	40.16%
BAS_7	2	100	25.637	32.611	39.17	44.635	27.20%	52.79%	74.10%

Based on the results presented in Table 5.11 and Table 5.12, we calculated the real value of overhead caused by the monitoring for each service. Table 5.13 presents the overhead for code block monitors. In the BAS service, the overhead corresponds to the number of code block monitors. The maximum overhead caused by one code block monitor is 0.8ms; 10 code block monitors can cause overhead from 0.7ms to 3.5ms; and when the number of code block monitors is increased up to 100, the overhead also increases by 5.5ms and 12.5ms. Although, the overhead from one and 10 code block monitors are similar, we can still see a linear increase in overhead with an increase in the number of code block monitors. In the ECS and OPS services, the number of activated code block monitors is very low, i.e., less than 10. In four out of six cases, the total overhead from code block monitor is small. However, in two cases, the caused overhead is comparable to the overhead of 100 code block monitors in the BAS service. These two cases come from the results of ECS_1 and OPS_1, respectively. As mentioned before, both requests cause very long response times. Furthermore, the deviations of response times caused by both requests are also very large, i.e., 178.954 ms for ECS_1 and 99.929 ms for OPS_1, as shown in Table 5.10. Although the results for code block monitoring from both requests are relatively larger than that of other requests, they can be ignored, when compared to the base response time results and their deviations. Therefore, it is possible that the large deviations may influence the results for code block monitoring.

Table 5.13: Monitoring Overhead for Code Block Monitor in Milliseconds

Service	# of Code Block Monitors	Minimum Overhead	Maximum Overhead
BAS	1	-0.582	0.764
BAS	10	0.743	3.476
BAS	100	5.465	12.494
ECS	2	-0.001	0.993
ECS	4	-0.401	-0.22
ECS	5	1.876	1.933
ECS	6	6.333	10.566
OPS	7	-5.01	2.657
OPS	8	2.217	9.287

Table 5.14 shows the overhead results for Turmeric monitors. Compared with the overhead for code block monitors, it is more obvious to see the overhead of Turmeric monitors increases along with the number of activated Turmeric monitors.

Table 5.14: Monitoring Overhead for Turmeric Monitor in Milliseconds

Service	# of Turmeric Monitors	Minimum Overhead	Maximum Overhead
BAS	2	4.259	16.148
ECS	2	6.27	7.264
ECS	10	18.9	19.521
ECS	14	45.676	63.251
OPS	18	33.413	117.095

We also investigate the amount of monitoring data produced by each request, in order to see if the throughput of monitors affects their overhead. Table 5.15 presents the total size of monitoring data from two levels of monitoring for each request. Combined with the impact percentages of code block monitoring shown in Table 5.12, we notice that the data size and the impact of code block monitoring for BAS requests have the same tendency, i.e., when the data size is large, the impact percentage for the same request is also large, and vice versa. However, the main reason behind this situation is that both the data size and the impact of code block monitoring are tightly depending on the number of code block monitors. The content of monitoring data from a code block monitor is the id of this code block, so the monitoring data for all code block monitors in our system is always the same size. If more code block monitors are activated, more data will be generated. If we further calculate the data size and the impact per code block monitor for each BAS request, as shown in Table 5.16, we can more clearly see that larger data size does not cause larger impact (compare BAS_1 with BAS_4) for code block monitoring in BAS. We apply the same analysis to the rest of results, and our conclusion is that the size of monitoring data is not really a big issue in terms of overall monitoring overhead.

Table 5.15: The Size of Monitoring Data in Byte

Monitor	BAS_1	BAS_2	BAS_3	BAS_4	BAS_5	BAS_6	BAS_7	ECS_1	ECS_2	ECS_3	ECS_4	OPS_1	OPS_2	OPS_3	OPS_4
Code Block	190	19	3K	270	21	270	3K	44	15	36	29	62	68	76	76
Turmeric	707	2K	915	805	782	2K	503	5K	548	6K	4K	10K	10K	10K	10K

Table 5.16: Data size vs impact per code block monitor for BAS(Just for illustration)

Monitor	BAS_1	BAS_2	BAS_3	BAS_4	BAS_5	BAS_6	BAS_7
Data size	19	19	30	27	21	27	30
Impact %	1.7%	1.69%	0.32%	0.56%	-1.18%	0.63%	0.27%

The Turmeric monitor that we implemented for the experiments in (Chen et al., 2013a) caused a large amount of overhead. The major reason for this overhead was due to the use of synchronous database access to record the monitoring data. In the current implementation, we have changed the synchronous database access to a Redis-based Publish/Subscribe messaging mechanism for the logging of monitoring data, causing less overhead. The main function that Turmeric monitors perform is to handle the incoming and outgoing messages, parse the context of a message to get predefined data for SFL and log the monitoring data. In order to investigate how much of the total overhead can be attributed to just the logging of the data, we created two setups in which the Turmeric monitors are enabled to handle service messages and no code block monitoring was activated. In the first setup the Turmeric monitor is set without data logging, while in the second setup the monitor does publish the monitoring data.

The third and fourth columns in Table 5.17 show the end-to-end response time of each request measured in the system. The third column represents the case with data logging activated, while the fourth column shows the setup where data logging has been disabled. The overhead of the data logging part in the Turmeric monitors is calculated and presented in the fifth column. In order to assess how much the data logging part can impact the performance of the Turmeric monitor, we calculated the overhead of Turmeric monitors for each request based on the results in Table 5.12 and Table 5.11, and also presented in the Table 5.17. The last column of Table 5.17 presents the percentage of overhead caused by the data logging. In most cases, the data logging causes between 20% and 40% of the overhead in the Turmeric monitoring.

Table 5.17: Overhead for the logging part in Turmeric monitor in Milliseconds

Service Requests	# of Turm. Moni.	With Turmeric, No Code Block Monitoring Data logging Acticated	Disabled	Data Logging Over.	Turmeric Monitor Overhead	%
BAS_1	2	22.027	18.745	3.282	9.06	36.23%
BAS_2	2	60.424	52.828	7.596	15.337	49.52%
BAS_3	2	47.437	45.798	1.639	12.728	12.88%
BAS_4	2	34.876	33.018	1.858	6.246	29.74%
BAS_5	2	53.709	51.922	1.787	4.259	41.96%
BAS_6	2	63.41	60.167	3.243	15.688	20.67%
BAS_7	2	39.17	36.939	2.231	13.533	16.49%
ECS_1	14	3055.052	2995.389	59.663	59.018	101.09%
ECS_2	2	56.928	54.036	2.892	7.264	39.81%
ECS_3	14	118.256	104.477	13.779	45.676	30.17%
ECS_4	10	66.477	60.841	5.636	18.9	29.82%
OPS_1	18	987.537	956.688	30.849	117.095	26.35%
OPS_2	18	177.714	165.165	12.549	42.21	29.73%
OPS_3	18	351.423	335.981	15.442	40.483	38.14%
OPS_4	18	202.53	181.418	21.112	54.765	38.55%

5.6 Discussion and Lessons Learned

Diagnosis Observations

From the simulations and the case study, we conclude that the monitoring granularity has indeed an effect on the calculation of an SFL diagnosis. Furthermore, increasing the monitoring granularity facilitates the calculation of correct and unambiguous diagnoses through introducing more and more diverse observations into the statistics of the SFL diagnosis. The increase in coverage diversity has a positive effect on the similarity coefficients produced, because it helps convict components that participate more in failing transactions, and exonerate components that participate more in passing transactions.

Initially we expected that we would not be able to achieve 100% correct diagnoses in our case study system. We thought that some of the tight couplings between sub-components would subsist across service boundaries, thereby invalidating our decoupling effort. This was not case. However, in the case study, some sub-components within the services are still tightly interacted, so that the sub-components are assigned the same similarity coefficient in the diagnosis. In other words, even though we can pinpoint the faulty service correctly, and this was our original goal, in some cases, we cannot determine the location of the fault within the service correctly. This comes from how we determine the finer-grained monitoring locations according to the predicate nodes in the service implementations. Some of the monitored code blocks are still exercised in combination, and thus, are tightly linked.

Here, an important lesson learned is that we can reduce tight coupling on the higher level of granularity, i.e. between services, but we cannot remove it entirely

on the lower levels of granularity, e.g. within services. We acknowledge the fact that topology plays a major role in the successful application of spectrum-based fault localization in service-based systems. In the future, we will look at other methods of topological separation, for example program slicing techniques (Weiser, 1981).

In addition, all experiments with both the simulator and the case study were set up for diagnosing a single fault in a service-oriented system. It is often not realistic that a software system only contains one fault. However, when applying online diagnosis for a service-oriented system, the diagnosis is activated immediately once a system failure is observed, i.e., the monitoring data of the system for each round of diagnosis only contains one failure. Within this context of single failure, the approach of diagnosing a single fault for a running service-oriented system is practical and effective. Multiple faults in a service-oriented system can be found one by one as long as they cause a failure.

Overhead Observations

In general, from the results of our overhead experiments, we observe that one Turmeric monitor can cause more overhead than one code block monitor. The overhead of Turmeric monitoring is always noticeable, whereas the overhead of code block monitoring is only visible when many monitors are activated. A small number of code block monitors in service-oriented system may be ignored in terms of a potential performance impact they create. On the other hand, if the number of code block monitors increases (e.g., 100), the caused overhead becomes comparable to Turmeric monitors.

We are aware of the fact that every type of monitoring comes at a cost. However, assessing the cost through measurement of overhead can be affected by various factors. From our experiments we found that the service-oriented system itself may influence the measurement. Basically, the response time of a request is a combination of service processing time, connection setup time, and message transmission time (Repp et al., 2007). Services which have interactions with other services always require more time in connection setup and message transmission. The connection setup depends on the activity state of both services and their underlying infrastructures. Transmission time depends on the quality of the network used. Thus, these two parts can be very dynamic and it may bring deviations to the overhead measurement. In our case system, most services are internal. They are running on the same computer system, so the message transmission time boils down to what is typically used in local socket communication. However, since our system is also based on the Turmeric platform, the connection to an internal service is setup with the Turmeric runtime library, we cannot guarantee that this third-party library will not bring any variation to the connection setup or transmission. Moreover, our system also uses an external service for real-time currency exchange, and we are not able to monitor the activity state of this external service;

plus all messages to the external service go through an external network connection. If the overhead caused by a monitor is too small, the connection setup or communication times can completely hide it. For example, Table 5.12 shows negative impact by the code block monitors invoked during the execution of BAS_5. This becomes obvious, if we check Table 5.13. It demonstrates that the overhead caused by one code block monitor is less than 1 millisecond, and Table 5.10, in which the standard deviation from the same request is nearly 10 milliseconds. The same is true for the result of "101.09%" for ECS_1 in Table 5.17 and the observation that the impact of Turmeric monitoring is larger than that of both Turmeric and code block monitoring for ECS_2 in Table 5.11.

We also determine that the data logging part inside the Turmeric monitoring is less than half of overall performance impact of the Turmeric monitors. The rest goes into intercepting and parsing of all incoming or outgoing messages. Even though, it does not publish any data, the interception already causes a lot of overhead in the monitoring.

Our experimental results show that a code block monitor consumes much less overhead than a Turmeric monitor does. This finding leads to an straight-forward idea for reducing monitoring overhead, which is completely replacing the Turmeric monitors with code block monitors. Additionally, a code block monitor also produce much less monitoring data than a Turmeric monitor does, based on our current implementation. A code block monitor only logs out the id of a code block, while a Turmeric monitor offers service and operation data, transaction data, message content, etc. If a code block monitor is implemented to get all those data, its overhead will also increase. In addition, a Turmeric monitor spends more than half of overhead on obtaining the required information from the Turmeric framework, even though those data are readily inside the framework. The code block monitor is staying inside the service implementation, where to fetch those required data and how to keep them would be a set of new problems for code block monitor. If code block monitors are equipped with all those functionalities, it will generate more overhead than it currently does, and its overhead may become comparable with or even more than that of Turmeric monitor. Therefore, replacing Turmeric monitor with code block monitor is not a good solution to deal with monitoring overhead.

Threats to Validity

We are aware of a number of threats that might invalidate our findings. We use SFL-Stonehenge[7] as case study. Although it is a realistic system, our results may not be applicable to any arbitrary service-based system. In fact, the topology of a system may have an effect on how well monitoring can be applied and diagnosis can be performed, e.g., in the case of very few independent paths through the logic. We see the topology problem as an important avenue for future work.

[7]https://github.com/SERG-Delft/sfl-stonehenge

Currently we implement code block monitor with Redis pub/sub functionality. It enables the diagnosis engine to receive the monitoring data from code block monitors at runtime. However, the association between the monitoring data from code block monitor and Turmeric monitor is based on timestamps, this approach may not be applicable to service-oriented systems allowing concurrent transactions.

A threat to our overhead experiments is the involvement of the external service for currency exchange in our system. This service is out of our control. The connection to the external service highly depends on its activity state. Its response can be very slow if it is overloaded. Correspondingly, the performance of the external service can affect the measurement of the end-to-end response time for those requests which invoke the external service. In addition, the Turmeric runtime library may also have an influence on the connection setup of services built on Turmeric platform.

Another potential threat comes from the tools used for our work. We have tested our own implementation as much as possible and compared the results of our case study with the outcome obtained from the simulator. Although the results are not the same, they are in a similar league, reassuring us that there are no major flaws in our case study implementation.

Another important threat to external validity is that the results for the overhead experiment might be dependent on the underlying technology, e.g., Turmeric or the way that the code block monitor is implemented. In future work we will replicate our experiment with different underlying technology to establish whether the obtained overhead results are generalizable.

We are also aware of the fact that code block monitors can not be inserted into the service implementation without access to the source code, which in turn typically entails the ownership of the service. Service-based systems can integrate external services that are not owned, thus precluding the application of our approach. However, for those companies which own large enterprise IT infrastructure and a lot of internal services running on it, such as eBay, Amazon and Google, the placement of monitors inside services is both possible and useful.

5.7 Related work

In this section we briefly discuss the studies most relevant to diagnosis for service-oriented software systems. In particular, we start of by looking into other work that do diagnosis of service-based systems. Subsequently, we look into whether alternative fault localization techniques are applied. Finally, we look into monitoring for service-based systems and measurements for overhead of monitoring. Based on this small survey, we believe that we are the first to study the combination of (1) spectrum-based fault localization, (2) multi-level monitoring to overcome the fault localization problem for tightly interacted services and (3) a detailed analysis

of overhead of multi-level monitoring for diagnosis.

Diagnosis for service-based systems

Chen et al. present *Pinpoint* (Chen et al., 2002), a similar diagnosis approach plus a tool using similarity coefficients in order to infer a diagnosis from system activation and component involvement. However, even though their title suggests otherwise, they do not address the specific issues of diagnosing services, i.e. the problems of inter-service diagnosis, and the fact that services are used in different contexts.

Yan, et al. (Yan and Dague, 2007; Yan et al., 2009), propose a model-based approach to diagnose orchestrated Web service processes. Modeling is done through discrete event systems, which imposes a heavy burden on the user of the technique. Zhang et al. (Zhang et al., 2009, 2012a) describe approaches for diagnosing quality-of-service problems in service-oriented architectures. However, their diagnosis approaches cannot adapt well to the dynamic nature of SOA, due to the static information they used. Moreover, their bayesian-based approaches are more heavyweight compared to spectrum-based approaches. Additionally, the authors measure the execution time for diagnosis, but their main purpose is to compare the performance of their two approaches, and they did not assess the overhead caused by diagnosis to the performance of service system. Mayer and colleagues (Mayer et al., 2010a, 2012), describe a similar diagnosis approach that is based on analyzing execution traces of failed transactions. However, the models they used for diagnosis are rather complex, and proper evaluation is still pending.

Fault localization

Wong et al. (Wong et al., 2010) discuss a number of code coverage-based heuristics to be used in fault localization. Grosclaude describes a model-based monitoring approach for diagnosing component-based systems, and suggests to use transactions IDs in order to associate messages sent between components (Grosclaude, 2004). This is also proposed by (Chen et al., 2002), and we see it as a standard approach to determine which service takes part in which system transaction. Chatzigiannakis and Papavassiliou (Chatzigiannakis and Papavassiliou, 2007) use principle component analysis in order to identify faulty nodes in sensor networks.

Spectrum-based fault localization is a lightweight technique, but alternatives exists. One such alternative are techniques that are model-based. Although outside the realm of service-based computing, Feldman et al. have proposed a greedy stochastic algorithm for computing diagnoses within a model-based diagnosis framework (Feldman et al., 2010). An important drawback of these model-based approaches is that we need to provide a correct model of the nominal behavior of the entire service-based application, which is daunting. A second issue is the combinatorial explosion in the reasoning of model-based diagnosis that inhibits the diagnosis of very large systems.

Monitoring for service based systems

There are a large number of papers about monitoring for service systems, however, most of them are missing overhead measurements, e.g., (Zulkernine et al., 2008; Keller and Ludwig, 2003). Furthermore, among those that do have monitoring overhead measurements, most of them are lacking a real and proper service system for evaluation, e.g., (Baresi and Guinea, 2013). In what follows, we present some of the monitoring solutions that have been presented.

Lin et al. (Lin et al., 2009) implement a middleware to monitor and diagnose service systems. They use a self-created example business process to measure the overhead of data collection. They do not provide detailed analysis of monitoring impact and types of monitor. Heward et al. in (Heward et al., 2010b) quantify and assess the performance impact of monitoring on a web service. Although they measure the performance impact under various monitoring setups, the testing vehicle they used is a single service.

Moscinat and Bonder present ADULA, a framework for automated maintenance of BPEL (Business Process Execution Language) processes (Mosincat and Binder, 2011). ADULA automatically detects and repairs service-level agreement (SLA) violations caused by service performance degradation in a way transparent to the user and to the BPEL engine. Their approach uses lightweight sampling monitoring and allows for customizable violation detection. They have also implemented repair policies, so that a service which violates the SLA can be replaced with another services that does adhere to the SLA violation. Their approach has a clear focus on performance and not on correctness.

Baresi et al. present a step towards self-healing compositions of service. Their approach is to monitor the execution of a service composition and trigger a suitable reaction so that the system can continue its execution (Baresi et al., 2007). The faulty behaviors that they consider are non-answering services and services violating their contracts. Their approach thus heavily relies on a contract violation being present. In contrast, our approach does not make assumptions towards contract violations and is more geared towards detecting the actual defect in a service composition.

5.8 Summary

The goal of this chapter is to investigate to which extent an increase in monitoring granularity supports the diagnosis of faulty services and its impact on service-oriented system. Referring to our research questions, we looked at:

RQ5.1: How and to which extent does the monitoring granularity affect the calculation of a diagnosis with spectrum-based fault localization? First, we used a simulator to reason over different service topologies. Second, we performed an actual case study on a service-oriented system, varying the level of monitoring granularity. The main conclusion from both experiments is that increasing the level of moni-

toring granularity can indeed improve diagnosis. More precisely, in our case study we could obtain up to 100% correct diagnoses. This comes through the increased variability in the observations used for the activity matrix of the SFL technique.

RQ5.2: How can we increase the monitoring granularity? The natural choice for placing monitors is at the service-level. However, this is so coarse-grained that many cases cannot be correctly diagnosed. Increasing the level of observation-granularity can then only be done by going into the services, changing their implementations. A brute force approach would be to monitor every single line of code. However, we restrict the monitoring to the code block level, representing unique execution branches through a service or proper isolation of tight coupling.

RQ5.3: What is the overhead incurred? Our case study demonstrates that we are able to diagnose all faulty services correctly through increasing the monitoring granularity. Yet, at the same time, we are also worried about the performance overhead that the entire infrastructure adds. The total impact of monitoring on the system performance depends on the number of used monitors. In detail, the monitoring at the service level, i.e., Turmeric monitoring, always causes more overhead than the monitoring at a finer-grained level, i.e., code block monitoring. On the other hand, when the number of code block monitors is small, the caused overhead can be negligible, however, the overhead can also become comparable with Turmeric monitoring if the number of code block monitors is increased.

Contributions Our work makes the following contributions:

1. We apply spectrum-based fault localization in the area of service-oriented systems in order to pinpoint problematic services.

2. We introduce the problem of tight service interaction, an inhibiting factor towards obtaining a good diagnosis of where the problematic service is located.

3. We present the SFL simulator, a simulation environment in which we can simulate faulty behavior of services with a certain probability and which allows us to study many service topologies with regard to the tight service interaction problem.

4. We introduce the idea of intra-service fine-grained monitoring to overcome the tight service interaction problem.

5. We present a case study with SFL-Stonehenge, a small real-world and open-source case study to illustrate that fine-grained monitoring can indeed help overcome the tight service interaction problem.

6. We perform an in-depth study on the performance overhead of our fine-grained monitoring approach.

Future work Based on the finding that the overhead of code block monitoring is tightly related to the number of its monitors and its overhead can become comparable with that of Turmeric monitoring, we plan to study where would be the best place for monitors in a service-oriented system. Such monitor placement can achieve the highest accuracy of diagnosis and the least disturbance to the service-oriented system at runtime. In the case study, we did the placement of monitors manually, but in future work, we would like to use some techniques, such as code slicing, to make it automatic. Currently, the monitors for different granularities are also deployed at compile time, we would like to enable dynamic monitoring in the future. This can also facilitate the automation of monitor placement.

Another area of future research is verifying whether our approach would also work for component-based systems.

6 Diagnosis Improvement Through Invocation Monitoring

Service oriented architectures support software runtime evolution through reconfiguration of misbehaving services. Reconfiguration requires that the faulty services can be identified correctly. Spectrum-based fault localization is an automated diagnosis technique that can be applied to faulty service detection. It is based on monitoring service involvement in passed and failed system transactions.

Monitoring only the involvement of services sometimes leads to inconclusive diagnoses. In this chapter, we propose to extend monitoring to include also the invocation links between the services. We show through simulations and a case study with a real system under which circumstances service monitoring alone inhibits the correct detection of a faulty service, and how and to which extent the inclusion of invocation monitoring can lead to improved service diagnosis.[1]

6.1 Problem Statement and Approach **87**
6.2 System Simulations **89**
6.3 Case Study **94**
6.4 Discussion **96**
6.5 Related work **97**
6.6 Summary **98**

Service-oriented architecture (SOA) is an architectural style that supports the construction of dynamic, adaptable, and evolvable systems well (Canfora and Di Penta, 2009b). Evolution takes place simply through runtime reconfiguration and versioning of the services involved in a SOA, e.g. through exchanging a faulty service for a healthy one (Bennett et al., 2000). Due to their highly dynamic nature, and the ultra-late binding of the service instances, which is one of the inherent characteristics of SOA (Lewis and Smith, 2008), traditional development-time quality assurance approaches must be superseded by techniques targeting operation time.

[1] This chapter contains material published at the *13th International Conference on Quality Software (QSIC'13)* (Chen et al., 2013b).

Spectrum-based fault localization (SFL) is a statistics-based, automatic diagnosis technique that has been demonstrated to perform well in pinpointing critical services during runtime (Chapter 3). It works by automatically inferring a diagnosis from observed symptoms (Abreu et al., 2009a). A diagnosis is a ranking of the potentially misbehaving services, and the symptoms are observations about service involvement in system transactions, plus pass/fail information for each transaction (Chapter 3 and (Gonzalez-Sanchez et al., 2011)).

Even though SFL was found to perform well in service-oriented systems, there are specific circumstances under which suboptimal diagnoses are achieved. A particular issue arises when service instances are tightly coupled. This means that several services are (almost) always invoked in combination, thereby inhibiting discriminative information to be produced in observations used by SFL, leading to inconclusive or ambiguous diagnoses. In other words, tightly coupled services are correctly pinpointed, but assigned the same rank in the diagnosis, as if they were one single service in its own right. Another issue arises when services exhibit fault intermittency. This means a service only fails sometimes when invoked.

Experiments performed for earlier work in Chapter 5 suggest that ambiguity and intermittency can be resolved through incorporating more detailed information to be used by SFL. One approach is the instrumentation of the services in order to retrieve finer grained observations on the code block level of a service implementation. This has been demonstrated to be successful in Chapter 5, but with the limitation that each service must support the instrumentation. Another approach would be the inclusion of information expressing the invocations between the services. We refer to these observations as *invocation link activation observations*. That way, the observations used by SFL do not only incorporate which services have been involved in a particular transaction, but, additionally, which routes through which invocation links between services were taken in a transaction. Our hypothesis is that this additional information can help to improve SFL-based diagnosis.

From these considerations, we can formulate the following research questions to be addressed in this chapter:

RQ6.1: To which extent can the usage of information expressing activation of links between services improve diagnosis?

RQ6.2: How does topology, i.e. the organization of the invocation links between services, affect diagnosis, and are there general characteristics of topology that improve diagnosis?

The main contributions of this work are an approach and algorithm that can be used to incorporate link invocation information in SFL, and a case study to demonstrate the extent to which invocation link information can improve SFL-based diagnoses.

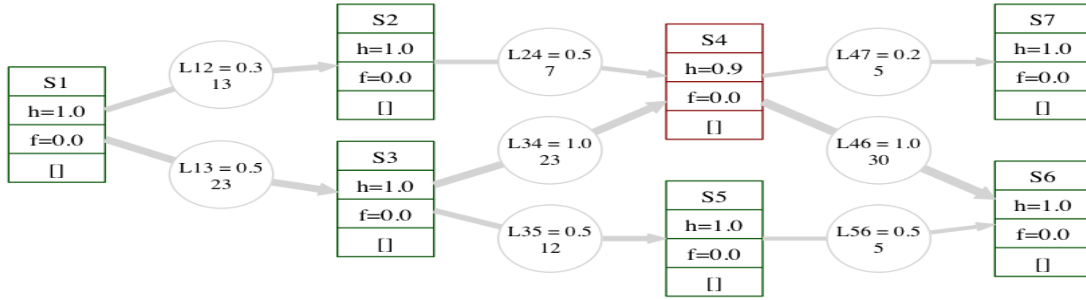


Figure 6.1: Example topology: illustration of the problem

The remainder of the chapter is organized as follows. Section 6.1 explains the problem in detail, and how it can be addressed. Section 6.2 shows how system simulations can be used to outline the approach and make an initial assessment. Section 6.3 evaluates our approach with a real service-based system, and Section 6.4 discusses the results of the experiments performed. Finally, Sections 6.5 and 6.6 present the related work and conclude the chapter.

6.1 Problem Statement and Approach

The topology shown in Fig. 6.1 and its corresponding activity matrix and diagnosis in Table 6.1 illustrate the problem addressed in this chapter. This topology is comprised of six healthy services with $h = 1.0$ and one faulty service S_4 with $h = 0.9$, representing low intermittent fault behavior. Intermittency of 0.9 means that if S_4 is invoked, it will fail in 10% of the cases. Failure probability is set to $f = 0.0$ in all services, meaning that once a fault is activated, it will not be detected immediately (i.e., turn into failure), making diagnosis more realistic and difficult. Services S_3, S_4 and S_6 are tightly coupled, indicated through the highest possible invocation probabilities of their respective links between them ($L_{34} = L_{46} = 1.0$). It means when S_3 is invoked, its subsequent tightly linked services S_4 and S_6 will also always be invoked.

A diagnosis in which only the activity of the services was considered would lead to S_4 and S_6 being ranked top with $SC_o = 0.34$, leading to an ambiguous result. However, after introducing invocation link information into the calculation of the diagnosis, as demonstrated in Table 6.1, the service S_4 becomes more suspicious, since it is associated with the top ranked invocation link L_{24}. This is reasonable, because all incoming and outgoing invocation links associated with a service represent more precise information about the activity of a service over time. Each invocation link represents a specific path leading into a service or leaving the service. If there are more paths to be observed, this leads to more varied activation observations for the service through the different paths, and, therefore, to more accurate information about which path lead to the activation of a fault. In the case of

Table 6.1: Activity matrix and diagnosis: illustration of the problem

Nodes	Activity Matrix	SC_o
L24	0000001001010000000100000000100000000010000010000000	0.436
S6	101001111101001111110111001010100001100011100100000	0.340
L46	101001111101001111110111001010100001100011100100000	0.340
S4	101001111101001111110111001010100001100011100100000	0.340
L12	000000100101000100110000011010000000100000100011000	0.320
S2	000000100101000100110000011010000000100000100011000	0.320
L56	000000000100000001000000100100000000000000100000000	0.258
S1	11111111111111111111111111111111111111111111111111	0.245
L13	101001111101001111010111001010100000100010100100000	0.241
S3	101001111101001111010111001010100000100010100100000	0.241
L34	101001111101001111010111001010100000100010100100000	0.241
L35	000000001100001110000111001000100000000010000100000	0.167
S5	000000001100001110000111001000100000000010000100000	0.167
S7	000000101000000001000010000000000000000000000010000	0.000
L47	000000101000000001000010000000000000000000000010000	0.000
Output	000000000000000000100000001000000000100000000000000	

tightly-coupled services, but also when services exhibit intermittent fault behavior, the additional information is beneficial for the diagnosis.

In other words, any of the links associated with a service represents a potentially different path through the service, thereby increasing the observation granularity. This is similar to adding monitors inside the services without touching their implementations, and we expect it can lead to similar results as reported in Chapter 5, without having to instrument the service implementation. In fact, this exploits topological information of the system, and it may be regarded as a first step towards combining SFL with model-based diagnosis (de Kleer and Kurien, 2003).

The specification shown in Alg. 1 defines the algorithm used to exploit the additional information introduced by the invocation link observations. It takes as input the ranking R of services and invocation links produced by SFL and the topological information A of the system, i.e. which service is associated with a link, and returns a set of potentially faulty services as diagnosis D. If all SC in R are 0.0, there was no observed failure. All services are considered to be healthy. If there was a failure and each item in R with the highest SC is a service, it returns these services as the diagnosis D. Otherwise, it means that some links ranked higher than or equal to services, and we can exploit the invocation link information. In this case, it extracts all links L that are ranking higher than or equal to the top-ranked service, and then it checks which services have the highest number of associations with those links. These services are stored in S. If S only contains one service, i.e. $|S| == 1$, then the algorithm returns D with the service as potentially faulty service. Otherwise, if there are more services with the same highest number of associations, it selects the ones with the highest SC and returns them as diagnosis D.

The algorithm determines the services with the highest number of associations with higher- or equally highly-ranked links. An invocation link ranking higher indicates that it is more likely to activate the fault and cause the failure. Therefore, a service which is associated more with these higher-ranked links is more likely to contain the fault than other services. In other words, services that are more associated with higher-ranked links are more related to the paths traversing those links which were covered when a fault was activated. Since a link cannot be faulty, the service is convicted that participates more in these paths that lead to fault activation.

In a nutshell, components that are more activated in failing transactions are more likely faulty. Invocation links are components that cannot be faulty. Services that are more associated with those assumed faulty links, are more likely faulty.

For example, service S_4 in Fig. 6.1 has two associations with the higher-ranked links L_{24} and L_{46}, S_6 has one association with the higher-ranked links, i.e. zero associations with L_{24}, and one association with L_{46}. That way, we can say that service S_4 is more suspicious to be faulty than service S_6, because service S_4 is participating more in transactions involving the assumed more likely faulty links L_{24} and L_{46}. Because links cannot be faulty, service S_4 becomes the most likely convict in this example case, which represents a correct diagnosis.

Algorithm 1 Diagnosis with invocation link information

```
function DIAGNOSE(R, A)
    T, L, S, D ← ∅
    sc_top = getHighestSC(R)
    if (sc_top != 0.0) then
        T ← {i | i.sc == sc_top and i ∈ R}
        if (∀i ∈ T and i is service) then
            D ← T
        else
            service_tr = getTopRankedService(R)
            L ← {l | l.sc ≥ service_tr.sc and l is link and l ∈ R}
            S ← getServicesWithHighestNumOfAssoc(L, A)
            if (|S| == 1) then
                D ← S
            else
                sc_max = getHighestSC(S)
                D ← {i | i.sc == sc_max and i ∈ S}
    return D
```

6.2 System Simulations

In order to validate our approach quickly and easily, we performed the initial assessment with our SFL simulator[2]. It provides functions for setting up component

[2]https://github.com/SERG-Delft/sfl-simulator

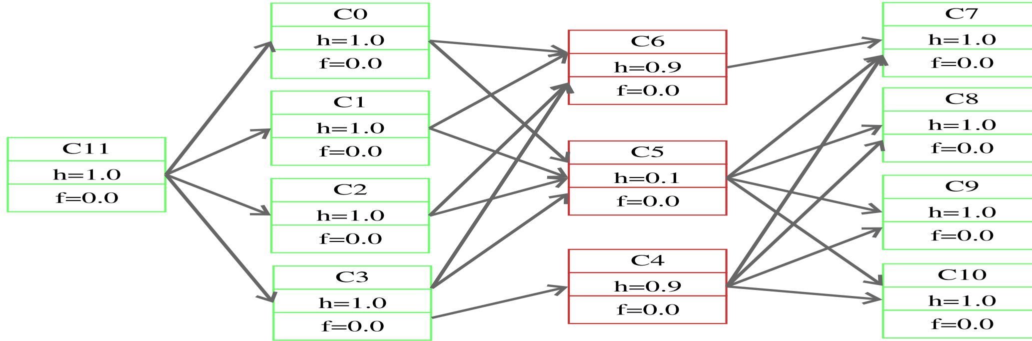


Figure 6.2: Topology for the pilot simulation

topologies, executing the topologies thereby gathering coverage information, and calculating diagnoses.

Pilot Simulation

In order to investigate how invocation link activation information influences the diagnosis for a service-based system, we used the SFL simulator to build a trial topology (Fig. 6.2). It is comprised of 12 components with different incoming and outgoing numbers of invocation links between them. Components C_4, C_5, and C_6 are set to be faulty, and they represent the study subjects on which we focus our interest. From initial experiments, performed for Chapter 5, we figured that the number of incoming and outgoing links might be significant for improving diagnosis through adding invocation link information (compare with Fig. 6.1). This comes from how additional invocation link monitors can separate the specific invocation paths leading into and coming out of components.

The three faulty components shown in Fig. 6.2 represent extreme cases, i.e. a component with one incoming link and several outgoing links (C_4), a component with several incoming links and one outgoing link (C_6), and a component with several incoming and outgoing links (C_5). In order to study the effects of invocation link activation information on diagnosis, the topology is executed according to different criteria.

In each experiment, the failure probabilities of the components are varied, i.e. $P_f = 0.0$ or $P_f = 1.0$, representing the probability that a failure can be detected when a fault was triggered. In addition, the invocation probabilities P_i between the concerned (faulty) components and their peers are varied, i.e. high interaction probability $P_i = 0.9$, low interaction probability $P_i = 0.1$. This represents the probability that a component associated with this link is activated. In each experiment, one of the components is set to be faulty with intermittency, i.e. low health $h = 0.1$ and high health $h = 0.9$, and it represents the probability that a faulty component will fail when invoked.

Table 6.2: Pilot simulation (500 activations)

Topology Setup			C_4		C_5		C_6	
P_f	P_i	h	better	worse	better	worse	better	worse
0.0	0.9	0.9	8.1%	3%	15%	0%	0.4%	0%
0.0	0.1	0.9	24.1%	3.4%	28.6%	2.9%	5.4%	0%
0.0	0.9	0.1	0%	0%	89%	0%	0.2%	0%
0.0	0.1	0.1	17.6%	2.7%	10.7%	0.2%	4.9%	2.2%
1.0	0.9	0.9	0%	47.9%	9.8%	0%	2.6%	0%
1.0	0.1	0.9	0%	3.6%	1.2%	1.2%	1.3%	1.3%
1.0	0.9	0.1	0%	0%	0%	0.2%	0%	0%
1.0	0.1	0.1	0.5%	8.7%	0%	1.7%	0%	0.7%

Table 6.2 summarizes the results of the experiments performed with these diverse topology setups. Every line in the table represents three experiments comprised of 500 diagnoses each. Every experiment was carried out with a specific topology setup, indicated in the first three columns, and with every of the three concerned components, C_4, C_5, or C_6 set to be faulty. For each of the 500 activations, the simulator was set to calculate one diagnosis based on only component activation observations, and another diagnosis based on both component and invocation link activation observations. The result of a diagnosis can be classified as *correct*, *ambiguous* or *incorrect*. A *correct diagnosis* pinpoints the faulty component correctly and uniquely (no duplicate top rankings). An *ambiguous diagnosis* pinpoints the faulty component but includes other healthy components on the same rank (duplicate top rankings). An *incorrect diagnosis* ranks any arbitrary healthy component higher than the faulty component.

Table 6.2 shows for each of the concerned components the percentage of how much better or worse the diagnoses become through incorporating invocation link activation information compared with merely using component activation information. The percentage is calculated based on the total number of failed transactions. *Better* means that an initially incorrect or ambiguous diagnosis can be performed correctly, through including invocation link information. *Worse* means that an initially correct diagnosis would become ambiguous or incorrect through including invocation link information.

From Table 6.2, we can see that if the failure probability is low, i.e. $P_f = 0.0$ (top part of the table), using invocation link activation information is more beneficial, in general. All concerned components show more *better* than *worse* results. Component C_5 scores the highest improvements, which, we believe, is attributable to its high number of incoming *and* outgoing invocation links.

An interesting result that we did not anticipate initially is the poor performance when the failure probability is high, i.e. $P_f = 1.0$. This is shown in the bottom part of Table 6.2. In this case, invocation observation carries not merely useless, but even misleading information. This comes from how the simulator treats failure

probability. It stops a transaction if a failure is detected in a component, thereby dismissing all information about its outgoing invocation links. This leads to component C_4 issuing the worst results, because of its low overall number of considered invocation links, i.e. only one incoming link. Because C_5 and C_6 have more incoming links, that can be considered in the diagnosis, their results are not so bad. This suggests that for the sake of diagnosability, real systems should have more invocation links between their components/services, and they should be built to recover from failure and continue operation.

Other interesting observations are the effects of health on the calculation of diagnoses. When failure probability is high, i.e. 1.0, and health is low, i.e. 0.1, it means an activation always causes a failure immediately. In this case, C_5 and C_6 are only becoming worse, i.e. invocation link activation information has no improvement at all. In addition, the overall worst case can be observed for component C_4 when failure probability, invocation probability and health probability are all high, i.e. $P_f = 1.0$, $P_i = 0.9$ and $h = 0.9$.

From these simulations, we can conclude that using invocation link observations in SFL is beneficial if the topology is highly interconnected (many invocation links between the services), and if a failure is detected, the system should recover and continue its operation, if possible.

Simulation with a Real System

After having established a strong empirical relation between a high number of incoming and outgoing invocation link activation observations and the quality of an SFL-based diagnosis, the next step is to assess our approach through simulation with a real system, which represents a more realistic setup compared to the pilot simulation. We use a simulation of our case study system presented in Section 6.3.

The simulated system consists of a number of components, i.e. service interfaces, and invocation links between them. Two of the components that exhibit poor diagnosability in the real system are set to be faulty with low intermittency of $h = 0.8$, all other components are set to be healthy. The two poorly diagnosable components are *ExchangeCurrencyService* and *OrderProcessorService*. In the following, we refer to them as *ECS* and *OPS*, respectively. The invocation probabilities between the components used for simulation are determined experimentally, based on the implementation logic plus the test data used in order to execute the real system. Failure probability in the simulation is set to 0.0, reflecting the behavior of the real system, i.e. faults are not detected immediately.

The number of simulations is set to a high value, i.e. 2000, in order to create a statistically significant data set. One problem with simulating real systems is that the simulation of service and invocation link activation is completely random, solely based on the predetermined probabilities, whereas, in the real system, invocations follow distinct patterns coming from the system's usage profile and the business logic of the services. In order to retrieve a meaningful dataset in the simu-

Table 6.3: Determining the number of realistic simulation activations

Simulated system activations	Number of failed activations, measured range (min – max)	Deviation from num. of failures in the real System
100	62 – 82	10.00%
500	393 – 416	4.60%
1000	776 – 818	4.20%
2000	1585 – 1614	1.45%
5000	3985 – 4046	1.22%

lation, it is, therefore, essential to generate many activations. Table 6.3 shows how an increase in the number of simulated activations eventually leads to a decrease in deviation from the number of failures in the real system. A low number of activations in the simulation results in high deviation of the number of failed transactions compared to the real system. Only at 2000 activations, the simulations lead to a number of failures that is comparable to the failures generated in the case study system, i.e. an acceptable deviation of 1.45% compared to executing the real system. Any more activations in the simulation do not improve the deviation from the real system significantly. Hence our choice of 2000 activations for the simulation.

In the simulation, the system is exercised with component activation observation enabled, and then with both, component and invocation link activation observation enabled. Table 6.4 presents the total number of failures in the simulation, and how many of the failed activations lead to incorrect (inc), ambiguous (amb) and correct (cor) diagnoses. For both services, diagnosis improves considerably, when invocation activation information is included, i.e. an improvement from 49.5% to 63.4% correct diagnoses for component ECS, and from 24.1% up to 52.6% correct diagnoses for component OPS.

Table 6.5 shows more details about how the inclusion of invocation link activation information makes diagnoses better or worse in the simulations. For service ECS, 235 diagnoses (out of 1616) are better, of which 132 ambiguous diagnoses can be turned into correct diagnoses (Amb→Cor), and also 103 incorrect diagnoses can be turned into correct ones (Inc→Cor). However, 11 correct diagnoses are turned into incorrect diagnoses (Cor→Inc). For service OPS, the improvement is much better. Inclusion of invocation link activation information improves the diagnoses in 485 cases (out of 1703), 115 ambiguous diagnoses can be resolved, and 370 diagnoses can be corrected. We did not find any worse diagnoses for service OPS. In future work, we will analyze these results carefully and try to determine why some cases issue worse results. This may indicate a limitation of our approach in terms of which kind of topology could be misleading the diagnosis.

Table 6.4: Simulation results for 2000 activations

Comp.	# of Fail.	Component Activation Inc	Amb	Cor	Cor-%	Comp. + Invocation Activation Inc	Amb	Cor	Cor-%
ECS	1616	624	192	800	49.5%	577	15	1024	63.4%
OPS	1703	1165	127	411	24.1%	785	22	896	52.6%

Table 6.5: Detailed Distribution of Better and Worse Diagnoses through Invocation Coverage

Services	Better Diagnoses Total	Amb→Cor	Inc→Cor	Worse Diagnoses Total	Cor→Amb	Cor→Inc
ECS	235	132	103	11	0	11
OPS	485	115	370	0	0	0

6.3 Case Study

In order to evaluate our approach more thoroughly, we conducted an experiment on our original case study SFL-Stonehenge[3] introduced in Chapter 2, and adapted it to the requirements implied by our problem statement. The system was extended to deal with invocation link activation information.

Conducting the Case Study

The case study system is the same system that we used in the simulations with two faulty services exhibiting poor diagnosability, i.e. *ExchangeCurrencyService* (ECS) and *OrderProcessorService* (OPS). Both services also exhibit tight coupling with their peers, and intermittent fault behavior. The goal of the case study is to assess to which extent the inclusion of invocation link activation information can improve their diagnosability.

We applied the PIT mutation tool[4] in order to create 65 faulty service versions, 24 faulty versions of *ECS* and 41 faulty versions of *OPS*. For each of the 65 faulty system versions, we use JMeter to execute 48 web service requests as test scenarios in order to cover all service operations. Upon completion of all transactions for one faulty system version, the diagnosis engine is invoked to parse the monitoring data, identify the failures in the system, and create an activity matrix with an output vector. The monitoring is provided through the Turmeric framework[5], mentioned in Section 3.1. Turmeric already logs all required transaction information, e.g., the traces of a service invoking other services. In other words, the invocation link activation information is readily available in the existing monitors.

[3]https://github.com/SERG-Delft/sfl-stonehenge
[4]http://pitest.org/
[5]https://github.com/ebayopensource/turmeric-runtime

Table 6.6: Diagnosis Results for SFL-Stonehenge

Service	Pass	Fail	Serv. Interface Activation Inc	Amb	Cor	Cor-%	Serv. Iface + Link Activation Inc	Amb	Cor	Cor-%
ECS	2	22	19	0	3	13.6%	3	0	19	86.4%
OPS	4	37	9	2	26	70.3%	3	0	34	91.9%

In order to assess to which extent the additional invocation link activation information makes diagnoses better or worse for the two faulty services, we invoked the diagnosis engine twice per execution. First, it creates activity matrices that are only comprised of service interface activation data. Second, it creates activity matrices that include both service interface activation data plus invocation link activation data. The two data sets can then be compared. The whole experiment is designed for the single fault case. We ensure that each of the 65 versions of the system contains only one fault, either in *ECS* or in *OPS*.

Case Study Results

Table 6.6 summarizes the case study results. It shows for each of the two faulty services, ECS and OPS, the total number of passed and failed transactions (pass/-fail) in the experiment. Some transactions pass, because the faults introduced by the mutations are not triggered. Then, it shows for the failed transactions, the incorrect, ambiguous and correct diagnosis results based on the two activation criteria, i.e., for service interface activation information on the left hand side, and for both service interface plus invocation link activation information on the right hand side. The results indicate considerable improvements in diagnoses that are based on service activation information plus invocation link activation. ECS improves from 13.6% up to 86.4% correct diagnoses, and OPS improves from 70.3% up to 91.9% correct diagnoses.

Table 6.7 shows details on how the diagnoses in the case study become better or worse after including the invocation link activation information. For ECS, 17 diagnoses are improved from incorrect to correct. For OPS, 2 diagnoses are improved from ambiguous to correct, 6 are improved from incorrect to correct. OPS does not receive any worse result, while for service ECS, one diagnosis deteriorates, i.e. from correct to incorrect. Careful analysis of this single worse diagnosis leads us to an explanation. The faulty service ECS is not only invoked by other services, but also directly from the user. Since we did not take the invocation link activations between users and services into account, this missing invocation link, which actually always activates the fault, cannot help to improve the diagnosis. This indicates the importance of including all the invocation links of the topology in the calculation of diagnoses. Once this link is added, the incorrect diagnosis can be corrected.

Table 6.7: Detailed Results for Better and Worse Diagnoses

	Better Diagnoses			Worse Diagnoses		
Service	Total	Amb→Cor	Inc→Cor	Total	Cor→Amb	Cor→Inc
ECS	17	0	17	1	0	1
OPS	8	2	6	0	0	0

6.4 Discussion

In the simulations and the case study we could identify considerable improvements by incorporating invocation link activation information into the calculation of SFL-based diagnoses. Our approach works, because it applies the same rules of the basic SFL that work for component activation information, also to the invocation link activation information. That is, services that participate more in failing transactions are more likely faulty, plus services that are more associated with *links* participating more in failing transactions, are more likely faulty.

Revisiting the Research Questions

***RQ6.1:* To which extent can the usage of information expressing activation of links between services improve diagnosis?** Both simulation and case study demonstrate that incorporating invocation link activation information in addition to service interface activation information can significantly improve diagnoses performed by spectrum-based fault localization. In the simulations of the case system, correct diagnoses for service ECS could be improved by around 14 percent through the additional observations, and for service OPS by around 29 percent. Interestingly, the overall improvement in correct diagnoses in the real system is higher than for the simulation, i.e. improvement for ECS by around 73 percent, and for OPS by around 22 percent, when including the additional observations. We believe this much better result in the real service-based system compared to its simulation comes from the fact that the simulator generates completely random invocation combinations between services, whereas, in the real system, service invocations follow less dynamic combinations, according the system's typical usage patterns. In other words, in the real system, much less different paths are exercised leading to a few prominent invocation patterns, whereas the simulation produces many more different service invocation combinations (compare with Table 6.3). This is an interesting observation which will be further researched in the future, i.e. can the combination of usage profile plus its associated invocation patterns be used as information in order to improve diagnosis?

When we compare the current results with the results of our earlier work presented in Chapter 5, in which we instrument the services themselves with additional observation points, it becomes apparent that including information about the invocation links between services is inferior (about 10%–15% worse). For the

same case study system, this other approach could achieve 100% correct and unambiguous service diagnoses in Chapter 5. However, the huge advantage of invocation link observations is that they can be retrieved through the service platform, whereas, for our earlier approach, the service implementations had to be amended, which is not always possible. Therefore, it is essential to find other information external to services that can be used to achieve 100% correct diagnoses.

RQ6.2: **How does topology, i.e. the organization of the invocation links between services, affect diagnosis, and are there general characteristics of topology that improve diagnosis?** Through the simulations and the case study, we can demonstrate that the topology of the observation points has, indeed, an effect on the quality of the diagnoses calculated by SFL. We compared the number of correctly performed diagnoses for monitoring service interface activation vs. interface plus invocation link activation, and observed considerable improvements. The improvements come from how the additional invocation link activation information helps split the topology into finer grained and separable units thereby helping to discriminate better the various service invocation paths. The simulations suggest that a high number of incoming and outgoing invocation links is beneficial, however, through the case study we found that any more than one incoming and outgoing link is already sufficient to improve the results. We believe, it is not so much the total number of incoming and outgoing links which makes a difference, but how those links lead to more diverse activation of the execution paths within a service, thereby exploiting information similar to that generated by service-internal monitors, as demonstrated in Chapter 5. These effects will be studied in future work.

As general guideline for determining the monitoring topology of a service-based system, we propose to split the monitoring of services into finer grained units representing better the service's different functions, and exploit additional information better that is suitable to separate the execution paths of the transactions flowing through the services.

6.5 Related work

Chen et al. present *Pinpoint* (Chen et al., 2002), a similar diagnosis approach plus a tool using similarity coefficients in order to infer a diagnosis from system activation and component involvement. However, even though their title suggests otherwise, they do not address the specific issues of diagnosing services, i.e. the problems of inter-service diagnosis, and the fact that services are used in different contexts. Yan, et al. (Yan and Dague, 2007; Yan et al., 2009), propose a model-based approach to diagnose orchestrated Web service processes. Modeling is done through discrete event systems, which imposes a heavy burden on the user of the technique. Zhang et al. (Zhang et al., 2009, 2012a) describe approaches for di-

agnosing quality-of-service problems in service-oriented architectures. Mayer and colleagues (Mayer et al., 2010a, 2012), describe a similar diagnosis approach that is based on analyzing execution traces of failed transactions.

Wong et al. (Wong et al., 2010) discuss a number of code coverage-based heuristics to be used in fault localization. Grosclaude (Grosclaude, 2004) describes a model-based monitoring approach for diagnosing component-based systems, and suggests to use transactions IDs in order to associate messages sent between components. This is also proposed by (Chen et al., 2002), and we see it as a standard approach to determine which service takes part in which system transaction. Chatzigiannakis and Papavassiliou (Chatzigiannakis and Papavassiliou, 2007) use principle component analysis in order to identify faulty nodes in sensor networks.

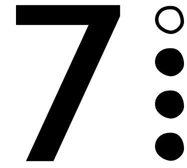
Heward et al. in (Heward et al., 2011) describe an algorithm for optimization of monitoring configurations for web services. They use their optimization algorithm in order to reduce the monitoring overhead in a service-based system, something that would also benefit our proposed techniques.

Li et al. (Li et al., 2008) describe an approach for control flow analysis and coverage for web services. They use their approach for testing purposes. Bartolini et al. (Bartolini et al., 2008) propose service coverage criteria that are based on service invocation monitoring. Their approach is also used for testing. Baresi et al. (Baresi et al., 2004b) introduce smart monitors for composed services, and Moser et al. (Moser et al., 2008) and Spanoudakis et al. (Spanoudakis and Mahbub, 2006) describe non-intrusive monitors for service-based systems.

6.6 Summary

In this chapter, we demonstrate how the monitoring of invocation link activation improves the performance of spectrum-based fault localization for diagnosing service-based systems. We devised an algorithm to deduce the faulty service from invocation link activation information based on the same assumptions that hold for the basic SFL approach. That is, a service that is associated more with an invocation link that participates more in failing transactions is more likely faulty. The pilot simulation revealed that the invocation links together with our algorithm can improve the diagnosis for components with more diverse interactions. This is even more the case, if the fault does not cause a failure immediately. Experiments with simulations, and a case study, confirm that the invocation link information can improve the diagnosis, in particular for real systems.

In the future, we are going to explore for which type of a topology can the invocation link information used for better diagnosis, and which other context information, such as the system's usage profile, can be also used for diagnosis.



7 Conclusion

In this thesis, we have focused on applying Spectrum-based Fault Localization(SFL) to diagnose Service-Oriented Systems at runtime. We have adapted the concepts of SFL to the context of service-oriented systems, and we have realized our SFL approach in service-oriented systems. To validate the performance of SFL in diagnosing service-oriented systems, we have conducted an experiment with a case study on a service-oriented system.

With the preliminary attempt of applying SFL to service-oriented systems, we discovered that the monitoring topology influences the accuracy of diagnosis for service-oriented systems. Therefore, we have proposed to apply Genetic Algorithms(GA) to find optimal monitoring topologies for diagnosing service-oriented systems with SFL.

After carefully investigating failed diagnoses from the initial step of applying SFL to service-oriented systems, we found that the main reasons for failed diagnoses can be attributed to tight interactions between services and fault intermittency of services. In order to improve diagnosis for such service-oriented systems, we have proposed two possible solutions. One solution is to increase the monitoring granularity by adding monitors at the code block level into the service implementation. The other solution is to include the monitoring of invocation links between services into the SFL diagnosis.

Furthermore, we also measured the runtime overhead caused by the diagnosis for service-oriented systems. Since the diagnosis engine in our implementation is detached from the service-oriented system, the only part of the diagnosis that can affect the runtime performance of service-oriented systems is from the online monitoring for the service-oriented system. Hence, we have measured the monitoring overhead at different levels of granularity. Below we present our contributions, answers to research questions and suggestions for future work.

7.1 Summary of Contributions

The main contributions of this dissertation are:

- An open-source service-based Java software system *SFL-Stonehenge*, which can be used as a possible standard case study for researchers working in the

area of SOA (Chapter 2).

- A brief survey of existing research initiatives in the area of SOA from which we extract criteria that need to be specified when performing a case study in order to allow future comparison and/or replication (Chapter 2).
- An approach for applying online SFL to service-oriented systems (Chapter 3).
- An evaluation of the performance of online SFL for service-oriented systems (Chapter 3).
- An approach for applying Genetic Algorithms to study the optimality of monitoring topologies that make service-oriented systems better diagnosable when applying SFL (Chapter 4).
- A set of general characteristics of monitoring topologies that improve SFL-based diagnoses (Chapter 4).
- A simulator for SFL-based diagnoses in various system topologies (Chapter 5).
- An approach to improve the accuracy of diagnosis for service-oriented systems by increasing the monitoring granularity (Chapter 5).
- A simulation and a case study to validate the approach of increasing the monitoring granularity (Chapter 5).
- A measurement of overhead caused by different levels of monitoring for service diagnosis (Chapter 5).
- An approach and algorithm of including invocation link information to improve the accuracy of SFL diagnosis for service-oriented systems (Chapter 6).
- A case study and a simulation to demonstrate the extent to which invocation link information can improve SFL-based diagnoses (Chapter 6).

7.2 The Research Questions Revisited

RQ3.1: How can a failure be detected in an operational service-oriented system?

In Chapter 3, we have reused the existing framework-based monitoring technique from the underlying service platform to obtain the information of service transactions at runtime. To determine a transaction pass or fail, we have devised an oracle together with the online monitoring to associate failure information with the transaction traces.

RQ3.2: How can spectrum-based fault localization be applied in a service-oriented system in order to trace a failure back to its respective root cause?

In Chapter 3, we have used a monitor and oracle to collect component involvement and pass/fail information. Based on this information, we have implemented a dedicated (external) diagnosis engine to generate an activity matrix and an output vector required by SFL and then calculate the diagnosis.

RQ3.3: How well does spectrum-based fault localization perform in a service-oriented system in terms of correctness of the diagnosis?

In Chapter 3, we performed a case study to assess how many correct diagnoses SFL is able to achieve in a service-oriented system. The results confirm the feasibility of the SFL approach, and indicate a high success rate of the diagnoses, i.e., 72% correctness.

RQ4.1: How can genetic algorithms be used to optimize monitoring topologies for spectrum-based diagnosis?

In Chapter 4, we applied genetic algorithms to find monitoring topologies that make service-oriented systems better diagnosable when applying SFL. The monitoring topology of a system is represented by an activity matrix, which can be expressed as a binary string and transformed into a GA-chromosome in a straightforward way. We also proposed a number of fitness functions. First, a function that expresses the diagnosability of a monitoring topology, i.e., the extent to which all diagnoses carried out on an activity matrix coming from that topology, are correct diagnoses. In the fitness function, each component is set to be faulty per diagnosis. Then, the fitness function calculates to which extent all similarity coefficients combined from all runs represent correct and distinguishable diagnoses. This yields a value representing how well a topology facilitates the discovery of each potential fault in every component. This basic fitness function can be amended in order to address the different optimization criteria required in the different experiments, e.g. favor high or low differences in the SC_o, or favor output vectors with low number of failures.

RQ4.2: What are characteristics of monitoring topologies that are optimal for spectrum-based diagnosis?

In Chapter 4, we defined a better diagnosable topology as a topology which allows all faults in a system to be detected in an unambiguous manner. The application of GA uncovers a number of routes to meet this fitness goal. Our results show that

- being able to invoke components in isolation facilitates diagnosability, because it helps separate component involvement in system executions better.
- adding monitoring points in the system and including the monitoring of inactivity, helps separating system executions, which is also beneficial for the diagnosability of the system.

- including monitoring of the system context (external components from other systems, incoming and outgoing activations) can support diagnosability through incorporating different invocation routes.

- including tracing information which represents combinations or distinct patterns of component coverage, may support SFL-based diagnosis.

RQ5.1: How and to which extent does the monitoring granularity affect the calculation of a diagnosis with spectrum-based fault localization?

In Chapter 5, we studied the effects of the monitoring granularity on performing SFL diagnosis. First, we performed simulations to reason over different service topologies with a simulator. Second, we conducted an actual case study on a service-oriented system, changing the level of monitoring granularity. The main finding from both experiments is that increasing the level of monitoring granularity can indeed improve diagnosis. More precisely, in our case study we could obtain up to 100% correct diagnoses. This comes through the increased variability in the observations used for the activity matrix of the SFL technique.

RQ5.2: How can we increase the monitoring granularity?

In Chapter 5, we increased the level of monitoring granularity by going into the service implementations, since the instinctive choice of placing monitors at the level of service operation is too coarse-grained, resulting in many cases that cannot be correctly diagnosed. A brute force approach of placing monitors inside the service implementation is to monitor every single line of code. However, in our approach, we restricted the monitoring to the code block level, representing unique execution branches through a service or proper isolation of tight interaction.

RQ5.3: What is the overhead caused by the monitoring of various levels of granularity?

In Chapter 5, we measured the monitoring overhead of different levels of granularity. The total impact of monitoring on the system performance depends on the number of used monitors. In detail, the monitoring at the level of service operation, i.e., Turmeric monitoring, always causes more overhead than the monitoring at a finer-grained level, i.e., code block monitoring. On the other hand, when the number of code block monitors is small, the caused overhead can be negligible, however, the overhead can also become comparable with Turmeric monitoring if the number of code block monitors is increased.

RQ6.1: To which extent can the usage of information expressing activation of links between services improve diagnosis?

In Chapter 6, we performed simulations and a case study to demonstrate that including the activation information of invocation links can significantly improve diagnoses performed by spectrum-based fault localization. In the simulations of the case system, correct diagnoses for the ECS service could be improved by around 14

percent through the additional monitoring, and for the OPS service by around 29 percent. In the real case system, the overall improvement in correct diagnoses is higher than for the simulation, i.e. improvement for the ECS by around 73 percent, and for the OPS by around 22 percent, when including the additional observations. Therefore, incorporating the activation information of invocation link can indeed improve SFL diagnosis when the diagnosed system has tight interaction problems. Furthermore, the huge advantage of this approach is that it does not require instrumenting the service implementations.

RQ6.2: How does topology, i.e. the organization of the invocation links between services, affect diagnosis, and are there general characteristics of topology that improve diagnosis?

In Chapter 6, we conducted simulations and a case study to compare the number of correct diagnoses for monitoring only the service interface versus the service interface plus the invocation link, and observed considerable improvements. The improvements came from how the additional activation information from the invocation link helps split the topology into finer grained and separable units, thereby helping to discriminate better the various service invocation paths. The simulations suggested that a high number of incoming and outgoing invocation links is beneficial, however, the case study illustrated that any more than one incoming and outgoing link is already sufficient to improve the results. We believe, the total number of incoming and outgoing links does not make so much difference, but how those links lead to more diverse activation of the execution paths within a service does. In addition, in Chapter 6, we proposed two general guidelines for determining the monitoring topology of a service-based system. One is to split the monitoring of services into finer grained units representing better the service's different functions. The other is to exploit additional information better that is suitable to separate the execution paths of the transactions flowing through the services.

7.3 Recommendations for Future Work

There are a number of interesting open issues for future work related to the topic of this thesis. In the following we suggest several recommendations:

1. Improve Diagnosis Accuracy

Our work with diagnosis improvement has highlighted the fact that including the activation information of invocation links can affect the diagnosis for service-oriented systems. This calls for an investigation on adding other context information, such as the system's usage profile, into the calculation of the diagnosis.

2. Optimize Monitoring Topology

We have shown that the overhead of code block monitoring is tightly related to the number of its monitors and its overhead can become comparable with that of

Turmeric monitoring. Future work should study where would be the best place for monitors in a service-oriented system, in order to achieve the highest accuracy of diagnosis and the least disturbance to the service-oriented system at runtime.

3. Automate Monitor Placement

In our case studies, the monitors for different granularities were deployed into the service-oriented system manually at compile time. Therefore, an issue left for future work is to use some techniques, such as code slicing, to make the placement of monitors automatic.

4. Explore Topology Effects

In Chapter 6, we have shown that the activation information of invocation links can affect diagnosis. In the future, it is necessary to explore for which type of topologies the invocation link information can be used for better diagnosis. This issue is also open for other context information.

5. Detect Tight Interaction

Our experimental results have revealed a fact that tight interactions between services can cause diagnosis to fail. A step before dealing with tight interaction is to discover whether a service-oriented system has tight interactions. Thus, a task for future work is to automatically detect tight interactions for a service-oriented system.

6. For Component-based Systems

All research work done in this dissertation is for service-oriented systems. This opens the door to verify whether our approaches would also work for component-based systems.

Bibliography

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

Abreu, R., Zoeteweij, P., and van Gemund, A. J. (2006). An evaluation of similarity coefficients for software fault localization. In *Proc. Int'l Symp. on Dependable Computing (PRDC)*, pages 39–46. IEEE.

Abreu, R., Zoeteweij, P., and van Gemund, A. J. (2009b). Spectrum-based multiple fault localization. In *Proc. Int'l Conference on Automated Software Engineering*, pages 88–99. IEEE.

Ahmad, A. and Pahl, C. (2011). Customisable transformation-driven evolution for service architectures. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 373–376. IEEE Computer Society.

Allauddin, M., Farooque, A., and Mehmooda, J. Z. (2011). A survey of quality assurance frameworks for service oriented systems. *International Journal of Advancements in Technology*, 2(2):188–198.

Ardissono, L., Furnari, R., Goy, A., Petrone, G., and Segnan, M. (2006). Fault tolerant web service orchestration by means of diagnosis. In *Proceedings of the Third European Workshop on Software Architecture (EWSA)*, volume 4344 of *LNCS*, pages 2–16. Springer.

Barbon, F., Traverso, P., Pistore, M., and Trainotti, M. (2006). Run-time monitoring of instances and classes of web service compositions. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 63–71. IEEE Computer Society.

Baresi, L., Ghezzi, C., and Guinea, S. (2004a). Smart monitors for composed services. In *Proceedings of the 2Nd International Conference on Service Oriented Computing*, ICSOC '04, pages 193–202. ACM.

Baresi, L., Ghezzi, C., and Guinea, S. (2004b). Smart monitors for composed services. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, pages 193–202. ACM.

Baresi, L., Ghezzi, C., and Guinea, S. (2007). Towards self-healing composition of services. In *Contributions to Ubiquitous Computing*, pages 27–46.

Baresi, L. and Guinea, S. (2013). Event-based multi-level service monitoring. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 83–90.

Bartolini, C., Bertolino, A., and Marchetti, E. (2008). Introducing service-oriented coverage testing. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 57–64.

Benatallah, B. and Motahari Nezhad, H. (2008). Service oriented architecture: Overview and directions. In Börger, E. and Cisternino, A., editors, *Advances in Software Engineering*, volume 5316 of *LNCS*, pages 116–130. Springer.

Benbernou, S., Hacid, L. C. M. S., Kazhamiakin, R., Kecskemeti, G., Poizat, J.-L., Silvestri, F., Uhlig, M., and Wetzstein, B. (2008). State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs. Deliverable # PO-JRA-1.2.1 of the S-Cube project.

Bennett, K., Layzell, P., Budgen, D., Brereton, P., Macaulay, L., and Munro, M. (2000). Service-based software: the future for flexible software. In *Proc. Asia-Pacific Software Engineering Conference (APSEC)*, pages 214–221. IEEE.

Bertolino, A., Inverardi, P., Pelliccione, P., and Tivoli, M. (2009). Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 141–150. ACM.

Brenner, D., Atkinson, C., Paech, B., Malaka, R., Merdes, M., and Suliman, D. (2006). Reducing verification effort in component-based software engineering through built-in testing. In *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International*, pages 175–184.

Bruning, S., Weissleder, S., and Malek, M. (2007). A fault taxonomy for service-oriented architecture. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 367–368.

Canfora, G. and Di Penta, M. (2006). Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10 –17.

Canfora, G. and Di Penta, M. (2009a). Service-oriented architectures testing: A survey. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer.

Canfora, G. and Di Penta, M. (2009b). Service-oriented architectures testing: A survey. In *Software Engineering*, volume 5413 of *LNCS*, pages 78–105. Springer.

Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwé, T. (2006). Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231.

Chapman, C., Saitou, K., and Jakiela, M. (1994). Genetic algorithms as an approach to configuration and topology design. *Mech. Des.*, 116(4):1005–1012.

Chatzigiannakis, V. and Papavassiliou, S. (2007). Diagnosing anomalies and identifying faulty nodes in sensor networks. *Sensors Journal, IEEE*, 7(5):637 –645.

Chen, C., Gross, H.-G., and Zaidman, A. (2012). Spectrum-based fault diagnosis for service-oriented software systems. In *Proc. of the Int'l Conf. on Service-Oriented Computing and Applications (SOCA)*. IEEE.

Chen, C., Gross, H.-G., and Zaidman, A. (2013a). Improving service diagnosis through increased monitoring granularity. In *7th Intl Conf. on Software Security and Reliability*, page to appear, Washington, DC.

Chen, C., Gross, H.-G., and Zaidman, A. (2013b). Improving service diagnosis through invocation monitoring. In *13th International Conference on Quality Software (QSIC)*, pages 85–94.

Chen, C., Gross, H.-G., and Zaidman, A. (2013c). Using genetic algorithms to study the effects of topology on spectrum based diagnosis. In *Proceedings of the 24th International Workshop on Principles of Diagnosis (DX)*, pages 166–173.

Chen, C., Omoro, B., Gross, H.-G., and Zaidman, A. (2013d). Comparing diagnostic performance of ochiai and relief in service-oriented systems. In *Proceedings of the 24th International Workshop on Principles of Diagnosis (DX)*, pages 39–44.

Chen, C., Zaidman, A., and Gross, H.-G. (2011). A framework-based runtime monitoring approach for service-oriented software systems. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications (QASBA)*, pages 17–20. ACM.

Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. (2002). Pinpoint: problem determination in large, dynamic internet services. In *Prod. Int'l Conf on Dependable Systems and Networks (DSN)*, pages 595–604. IEEE.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702.

Creswell, J. W. and Clark, V. L. P. (2010). Designing and conducting mixed methods research.

de Kleer, J. and Kurien, J. (2003). Fundamentals of model-based diagnosis. In *Fault Detection, Supervision and Safety of Technical Processes*, pages 25–36. IFAC.

Demeyer, S., Mens, T., and Wermelinger, M. (2002). Towards a software evolution benchmark. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 172–175. ACM.

Denaro, G., Pezzè, M., and Tosi, D. (2009). Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–262. ACM.

Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., and Pohl, K. (2008). A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341.

Domenico, B. and Carlo, G. (2007). Monitoring conversational web services. In *Proceedings of the 2nd international workshop on Service oriented software engineering (IW-SOSWE)*, pages 15–21. ACM.

Espinha, T., Chen, C., Zaidman, A., and Gross, H.-G. (2012a). Maintenance research in soa - towards a standard case study. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 391–396. IEEE.

Espinha, T., Chen, C., Zaidman, A., and Gross, H.-G. (2012b). Spicy stonehenge: Proposing a soa case study. In *Principles of Engineering Service Oriented Systems (PESOS), 2012 ICSE Workshop on*, pages 57–58.

Feldman, A., Provan, G. M., and van Gemund, A. J. C. (2010). Approximate model-based diagnosis using greedy stochastic search. *J. Artif. Intell. Res. (JAIR)*, 38:371–413.

Fidge, C. J. (1988). Timestamp in message passing systems that preserves partial ordering. In *Proceedings of the Australian Computing Conference*, pages 56–66.

Gold, N., Knight, C., Mohan, A., and Munro, M. (2004). Understanding service-oriented software. *IEEE Software*, 21(2):71–77.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison Wesley.

Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2011). Spectrum-based sequential diagnosis. In *Proc. Int'l Conf. on Artificial Intelligence (AAAI)*, pages 189–196. AAAI Press.

Gonzalez-Sanchez, A., Piel, E., Gross, H.-G., and van Gemund, A. (2010a). Prioritizing tests for software fault localization. In *Int'l Conf. on Quality Software*, pages 42–51. IEEE.

Gonzalez-Sanchez, A., Piel, E., Gross, H.-G., and van Gemund, A. (2010b). Runtime testability in dynamic high-availability component-based systems. In *Proc. Int'l Conf. Advances in System Testing and Validation Lifecycle (VALID)*, pages 37 –42. IEEE.

Granelli, G., Montagna, M., Zanellini, F., Bresesti, P., and Vailati, R. (2006). A genetic algorithm-based procedure to optimize system topology against parallel flows. *Power Systems, IEEE Transactions on*, 21(1):333–340.

Greiler, M., Gross, H.-G., and Nasr, K. (2009). Runtime integration and testing for highly dynamic service oriented ICT solutions – an industry challenges report. In *Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, pages 51–55.

Grosclaude, I. (2004). Model-based monitoring of component-based software systems. In *Int'l Workshop on Principles of Diagnosis*, pages 155–160.

Hadaytullah, Vathsavayi, S., Raiha, O., and Koskimies, K. (2010). Tool support for software architecture design with genetic algorithms. In *Proc. International Conference on Software Engineering Advances (ICSEA)*, pages 359–366. IEEE CS.

Harman, M. and Clark, J. A. (2004). Metrics are fitness functions too. In *Proc. of the Int'l Symp. on Software Metrics (METRICS)*, pages 58–69. IEEE.

Heward, G., Han, J., Schneider, J.-G., and Versteeg, S. (2011). Run-time management and optimization of web service monitoring systems. In *Proc. Int'l Conf on Service-Oriented Computing and Applications (SOCA)*, pages 1–6. IEEE.

Heward, G., Mueller, I., Han, J., Schneider, J.-G., and Versteeg, S. (2010a). Assessing the performance impact of service monitoring. In *Australian Software Engineering Conference (ASWEC)*, pages 192–201. IEEE Computer Society.

Heward, G., Mueller, I., Han, J., Schneider, J.-G., and Versteeg, S. (2010b). Assessing the performance impact of service monitoring. In *Software Engineering Conference (ASWEC), 2010 21st Australian*, pages 192–201.

Josuttis, N. M. (2007a). Soa in practice – the art of distributed system design.

Josuttis, N. M. (2007b). *SOA in Practice: The Art of Distributed System Design*. O'Reilly.

Keller, A. and Ludwig, H. (2003). The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81.

Kim, D. and Park, S. (2009). Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Proceedings of ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, volume 0, pages 76–85. IEEE.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734.

Kumar, A., Pathak, R. M., Gupta, Y. P., and Parsaei, H. R. (1995). A genetic algorithm for distributed system topology design. *Comput. Ind. Eng.*, 28(3):659–670.

Lamport, L. (1978). Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Lewis, G. and Smith, D. (2008). Service-oriented architecture and its implications for software maintenance and evolution. In *Frontiers of Software Maintenance (FOSM)*, pages 1–10. IEEE.

Li, L., Chou, W., and Guo, W. (2008). Control flow analysis and coverage driven testing for web services. In *Int'l Conf. on Web Services (ICWS)*, pages 473–480. IEEE.

Lin, K.-J., Panahi, M., Zhang, Y., Zhang, J., and Chang, S.-H. (2009). Building accountability middleware to support dependable soa. *Internet Computing, IEEE*, 13(2):16–25.

Lutz, R. (2001). Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47(7):613–634.

Madeira, J. A., Rodrigues, H., and Pina, H. (2005). Multi-objective optimization of structures topology by genetic algorithms. *Advances in Engineering Software*, 36(1):21–28.

Mahbub, K. and Spanoudakis, G. (2005). Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. *Proceedings of the International Conference on Web Services (ICWS)*, pages 257–265.

Marconi, A. and Pistore, M. (2009). Synthesis and composition of web services. In Bernardo, M., Padovani, L., and Zavattaro, G., editors, *Formal Methods for Web Services*, volume 5569 of *LNCS*, pages 89–157. Springer.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier.

Mayer, W., Friedrich, G., and Stumptner, M. (2010a). Diagnosis of service failures by trace analysis with partial knowledge. In *Service-Oriented Computing*, volume 6470 of *LNCS*, pages 334–349. Springer Berlin Heidelberg.

Mayer, W., Friedrich, G., and Stumptner, M. (2010b). Diagnosis of service failures by trace analysis with partial knowledge. In *ICSOC*, pages 334–349.

Mayer, W., Friedrich, G., and Stumptner, M. (2012). On computing correct processes and repairs using partial behavioral models. In *20th European Conference on Artificial Intelligence (ECAI)*, pages 582–587.

Mayer, W. and Stumptner, M. (2008). Evaluating models for model-based debugging. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 128–137.

Miller, B. and Goldberg, D. (1995). Genetic algorithms, tournament selection and the effects of noise. Technical Report 95006, IlliGAL Report, Dept. General Engineering, University of Illinois at Urbana Campaign.

Moe, J. and Carr, D. A. (2001). Understanding distributed systems via execution trace data. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 60–67. IEEE CS.

Mohamed, A. and Zulkernine, M. (2008). On failure propagation in component-based software systems. In *Proc. Int'l Conf. on Quality Software (QSIC)*, pages 402–411. IEEE.

Momm, C., Malec, R., and Abeck, S. (2007). Towards a model-driven development of monitored processes. *Internationale Tagung Wirtschaftsinformatik (WI2007), Karlsruhe*.

Moser, O., Rosenberg, F., and Dustdar, S. (2008). Non-intrusive monitoring and service adaptation for ws-bpel. In *Proc. Int'l Conf. on World Wide Web (WWW)*, pages 815–824. ACM.

Mosincat, A. D. and Binder, W. (2011). Automated maintenance of service compositions with sla violation detection and dynamic binding. *Int J Softw Tools Technol Transfer*, 13(2):167–179.

Nasr, K. A., Gross, H.-G., and van Deursen, A. (2011). Realizing Service Migration in Industry - Lessons Learned. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*.

Natis, Y. V. (2003). Service-oriented architecture scenario. Website last visited November 30th, 2011.

Novotny, P., Wolf, A. L., and Ko, B. J. (2012). Fault localization in manet-hosted service-based systems. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 243–248.

Papazoglou, M. (2008). The challenges of service evolution. In *Advanced Information Systems Engineering*, volume 5074 of *LNCS*, pages 1–15. Springer.

Papazoglou, M., Traverso, P., Dustdar, S., and Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45.

Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F., and Krämer, B. J. (2006). Service-oriented computing: A research roadmap. In Cubera, F., Krämer, B. J., and Papazoglou, M. P., editors, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings.

Piel, E., Gonzalez-Sanchez, A., Gross, H., and van Gemund, A. (2011). Spectrum-based health monitoring for self-adaptive systems. In *Proc. Int'l Conf. Self-Adaptive and Self-Organizing Systems (SASO)*, pages 99–108. IEEE.

Pistore, M. and Traverso, P. (2007). Assumption-based composition and monitoring of web services. In Baresi, L. and Di Nitto, E., editors, *Test and Analysis of Web Services*, pages 307–335. Springer.

Räihä, O., Koskimies, K., and Mäkinen, E. (2008). Genetic synthesis of software architecture. In *Proc. of the International Conference on Simulated Evolution and Learning (SEAL)*, volume 5361 of *LNCS*, pages 565–574. Springer.

Räihä, O., Koskimies, K., and Mäkinen, E. (2011). Generating software architecture spectrum with multi-objective genetic algorithms. In *Third World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pages 29–36. IEEE.

Repp, N., Berbner, R., Heckmann, O., and Steinmetz, R. (2007). A cross-layer approach to performance monitoring of web services. In *Emerging Web Services Technology*, pages 21–32.

Reps, T., Ball, T., Das, M., and Larus, J. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *European Softw. Engineering Conf. & Symp. on Foundations of Softw. Engineering (ESEC/FSE)*, volume 1301 of *LNCS*, pages 432–449. Springer.

Schmerl, B. R., Garlan, D., Dwivedi, V., Bigrigg, M. W., and Carley, K. M. (2011). Sorascs: a case study in soa-based platform design for socio-cultural analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 643–652. ACM.

Sim, S. E., Easterbrook, S. M., and Holt, R. C. (2003). Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 74–83. IEEE Computer Society.

Snell, J., Tidwell, D., and Kulchenko, P. (2001). *Programming Web Services with SOAP*. O'Reilly Media.

Spanoudakis, G. and Mahbub, K. (2006). Non-intrusive monitoring of service-based systems. *International Journal of Cooperative Information Systems*, 15(03):325–358.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In *Third International Conference on Genetic Algorithms*, pages 2–9.

Turner, M., Budgen, D., and Brereton, P. (2003). Turning software into a service. *Computer*, 36(10):38–44.

Weiser, M. (1981). Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE Press.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer.

Wong, W. E., Debroy, V., and Choi, B. (2010). A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208.

Yan, Y. and Dague, P. (2007). Modeling and diagnosing orchestratedweb service processes. In *Proc. Int'l Conf on Web Services (ICWS)*, pages 51–59. IEEE.

Yan, Y., Dague, P., Pencole, Y., and Cordier, M.-O. (2009). A model-based approach for diagnosing fault in web service processes. *International Journal of Web Services Research (IJWSR)*, 6(1):87–110.

Yin, R. K. (2014). Case study research design and methods.

Zaidman, A., Pinzger, M., and van Deursen, A. (2010). Software evolution. In Laplante, P. A., editor, *Encyclopedia of Software Engineering*, pages 1127–1137. Taylor & Francis.

Zhang, J., Chang, Y., and Lin, K.-J. (2009). A dependency matrix based framework for QoS diagnosis in SOA. In *Proc. Int'l Conf on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE.

Zhang, J., Huang, Z., and Lin, K. (2012a). A hybrid diagnosis approach for QoS management in service-oriented architecture. In *Int'l Conf. on Web Services (ICWS)*, pages 82–89. IEEE.

Zhang, J., Huang, Z., and Lin, K. (2012b). A hybrid diagnosis approach for qos management in service-oriented architecture. In *Proc. Int'l Conf. on Web Service (ICWS)*, pages 82–89. IEEE.

Zoeteweij, P., Abreu, R., and A.J.C. van Gemund (2007a). Software fault diagnosis. In *IFIP Int'l Conf. on Testing of Communicating Systems: Hand-Outs for the Tutorial Day of TestCom/FATES*, pages 1–26. Tartu University Press.

Zoeteweij, P., Abreu, R., Golsteijn, R., and van Gemund, A. (2007b). Diagnosis of embedded software using program spectra. In *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 213–220.

Zoeteweij, P., Abreu, R., Golsteijn, R., and van Gemund, A. J. (2007c). Diagnosis of embedded software using program spectra. In *Proc. Int'l Conf. and Workshops on Engineering of Computer-Based Systems (ECBS)*, pages 213–220. IEEE.

Zulkernine, F., Martin, P., and Wilson, K. (2008). A middleware solution to monitoring composite web services-based processes. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pages 149–156.

Summary

Automated Fault Localization for Service-Oriented Software Systems

– Cuiting Chen –

In this thesis, we have focused on applying Spectrum-based Fault Localization (SFL) to diagnose Service-Oriented Systems at runtime. We reused a framework-based online monitoring technique to obtain the service transaction information. We devised a three-phased oracle and combined this with monitoring to detect system failures at runtime. Both monitor and oracle generate component involvement and pass/fail information required by SFL. We conducted an experiment with a case system to validate the performance of SFL in diagnosing service-oriented systems. The results show that SFL is able to identify faulty service operations in 73% of the cases correctly.

With the preliminary attempt of applying SFL to service-oriented systems, we discovered that the monitoring topology can influence the accuracy of diagnosis for service-oriented systems. Therefore, we applied Genetics Algorithms (GA) to find the optimal monitoring topologies for SFL diagnosis. With the assistance of GA techniques, we have identified the following characteristics of optimal monitoring topologies:

- invoking components in isolation
- more monitoring points, including the monitoring of inactivity
- including the monitoring of the system context
- including tracing information

Through a careful investigation of the failed diagnoses from the initial step of applying SFL to service-oriented systems, we found that the main reasons for

failed diagnoses can be attributed to (1) *tight interactions* between services and (2) fault intermittency of services. In order to improve the diagnosis, we have proposed two possible solutions to deal with tight interaction. One solution is to increase the monitoring granularity by adding monitors at the code block level in the service implementation. The other solution is to include the monitoring of invocation links between services into the SFL diagnosis. The former solution is able to achieve 100% correct diagnoses, however, it requires the ownership of services to place monitors inside the services. The latter solution can be done with a more realistic set-up and it can also significantly improve the diagnoses.

We have also assessed the runtime overhead caused by the diagnosis for service-oriented system. Since the diagnosis engine in our approach is detached from the service-oriented system, the overhead of diagnosis imposed on the running service-oriented system is from monitoring. We measured the monitoring overhead at different levels of granularity, and found out that the monitoring at the service communication level consumes high overhead, whereas the monitoring at the service implementation level is much lower, but highly depends on the number of monitors deployed.

Samenvatting

Geautomatiseerde Foutlocalisatie voor Service-geörienteerde Software Systemen

– Cuiting Chen –

In dit proefschrift hebben we ons gericht op het toepassen van Spectrum-gebaseerde Foutlocalisatie (SFL) om service-georiënteerde systemen te diagnosticeren tijdens runtime. We hebben hiervoor een framework-gebaseerde online monitoring techniek herbruikt om de service transactie informatie te verkrijgen. We hebben een drie-fase orakel bedacht en gecombineerd met monitoring om systeemfouten tijdens de reguliere uitvoering op te sporen. Zowel de monitor als het orakel genereren informatie met betrekking tot de betrokkenheid van de component en de pass/fail informatie van de uitvoering die nodig is voor SFL. We hebben een experiment uitgevoerd om de geschiktheid van SFL voor het diagnoseren van service-georiënteerde systemen te valideren. De resultaten tonen dat SFL in 73% van de gevallen in staat is om foutieve service operaties te identificeren.

Tijdens de eerste pogingen om SFL op service-georiënteerde systemen toe te passen, ontdekten we dat de topologie van de verschillende monitors de nauwkeurigheid van de diagnose kan beïnvloeden. Daarom hebben we genetische algoritmen (GA) toegepast om de optimale topologie van monitors voor een SFL diagnose te bepalen. Met de hulp GA technieken, hebben we de volgende kenmerken voor een optimale monitor topologie geïdentificeerd:

- componenten moeten geïsoleerd worden opgeroepen
- monitors op meer punten plaatsen en ook inactiviteit monitoren
- de systeemcontext monitoren
- het toevoegen van tracing informatie

Door een zorgvuldig onderzoek van de mislukte diagnoses uit de eerste stap van het toepassing van SFL op service-georiënteerde systemen, vonden we dat de belangrijkste redenen voor gefaalde diagnoses kunnen worden toegeschreven aan (1) sterke interacties tussen bepaalde services en (2) intermitterende fouten van services. Om de diagnose te verbeteren, hebben wij twee mogelijke oplossingen ontwikkeld om om te gaan met service topologiën die sterke interacties vertonen. Een eerste oplossing is om de granulariteit van het monitoren te verhogen door monitors op het niveau van code blokken toe te voegen. De andere oplossing bestaat erin om de invocatie links tussen services mee in rekening te brengen tijdens de SFL diagnose. De eerste oplossing laat toe om 100% correcte diagnoses te verkrijgen, echter ze heeft als belangrijke voorwaarde dat de eigendomssituatie van de services toelaat om monitors in de code te introduceren. De tweede oplossing is meer realistisch uitvoerbaar en kan de diagnose ook gevoelig verbeteren.

We hebben bovendien ook de kost voor het online toepassen van SFL-gebaseerde diagnose in kaart gebracht. Omdat de diagnose component van onze aanpak is losgekoppeld van het service-georiënteerde systeem dat we willen diagnosticeren, blijft de kost voor de diagnose beperkt tot de kost voor monitoring. We hebben de kost voor monitoring bepaald op verschillende niveaus van granulariteit. Onze resultaten laten zien dat monitoring op het niveau van service communicatie een hoge kost kent, terwijl monitoren op het niveau van implementatie veel minder kost, maar wel afhangt van het aantal monitors dat geplaatst wordt.

Curriculum Vitae

Cuiting Chen
Born: February 17th, 1984
in Fujian, China.

Education

2010 – 2015: Ph.D., Computer Science
Delft University of Technology, Delft, The Netherlands. Under the supervision of Prof.dr. Arie van Deursen.

2007 – 2010: M.Sc., Computer Science
Dresden University of Technology, Dresden, Germany.

2001 – 2005: B.Sc., Electronic Engineering
Beijing Jiaotong University, Beijing, China.

Work Experience

November 2010 – March 2015: Assistant in Opleiding (AIO). ***Research Trainee***
Software Technology Department, Delft University of Technology. Mekelweg 4, 2628CD Delft, The Netherlands.

July 2005 – March 2007: Electronic Engineer
MotherBoard R&D 1 Dept. Asustek(Suzhou), China

March 2005 – July 2005: Internship ***Field Application Engineer***
OEM Dept. Asus(Beijing), China.

Review Experience

- Served as PC member for the 29th AAAI Conference on Artificial Intelligence (AAAI'15)
- ACM Transactions on Software Engineering and Methodology (TOSEM)

Publications

- C. Chen, H.-G. Gross, A. Zaidman. Analysis of Service Diagnosis Improvement through Increased Monitoring Granularity. *Under Review*.
- C. Chen, H.-G. Gross, A. Zaidman. Using Genetic Algorithms to Study the Effects of Topology on Spectrum Based Diagnosis. In *Proceedings of the 24th International Workshop on Principles of Diagnosis (DX'13)*, 2013.
- C. Chen, B. Omoro, H.-G. Gross, A. Zaidman. Comparing Diagnostic Performance of Ochiai and Relief in Service-oriented Systems. In *Proceedings of the 24th International Workshop on Principles of Diagnosis (DX'13)*, 2013.
- C. Chen, H.-G. Gross, A. Zaidman. Improving Service Diagnosis through Invocation Monitoring. In *Proceedings of the 13th International Conference on Quality Software (QSIC'13)*, 2013.
- C. Chen, H.-G. Gross, A. Zaidman. Improving Service Diagnosis through Increased Monitoring Granularity. In *Proceedings of the 7th International Conference on Software Security and Reliability (SERE'13)*, 2013. Distinguished Paper Award.
- C. Chen, H.-G. Gross, A. Zaidman. Spectrum-based Fault Diagnosis for Service-Oriented Software Systems. In *Proceedings of the 5th International Conference on Service-Oriented Computing and Applications (SOCA'12)*, 2012.
- T. Espinha, C. Chen, H.-G. Gross, A. Zaidman. Spicy Stonehenge: Proposing a SOA Case Study. In *the International Workshop on Principles of Engineering Service-Oriented Systems (PESOS'12)*, 2012.
- T. Espinha, C. Chen, H.-G. Gross, A. Zaidman. Maintenance Research in SOA - Towards a Standard Case Study. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, 2012.
- C. Chen, A. Zaidman, H.-G. Gross. A Framework-based Runtime Monitoring Approach for Service-Oriented Software Systems. In *Proc. of the Intl Workshop on Quality Assurance for Service-Based Applications (QASBA'11)*, 2011.