TU Delft

Delft University of Technology

Analysis of service diagnosis improvement through increased monitoring granularity

Chen, Cuiting; Gross, Hans Gerhard; Zaidman, Andy

DOI
10.1007/s11219-015-9286-2

Publication date
2017

Document Version
Final published version

Published in
Software Quality Journal

Citation (APA)
Chen, C., Gross, H. G., & Zaidman, A. (2017). Analysis of service diagnosis improvement through increased monitoring granularity. *Software Quality Journal*, *25*(2), 437-471. https://doi.org/10.1007/s11219-015-9286-2

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Software Qual J
DOI 10.1007/s11219-015-9286-2

Analysis of service diagnosis improvement through increased monitoring granularity

Cuiting Chen[1] · Hans-Gerhard Gross[2] · Andy Zaidman[1]

© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract 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 paper, 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.

✉ Cuiting Chen
cuiting.chen@tudelft.nl

Hans-Gerhard Gross
hans-gerhard.gross@hs-esslingen.de

Andy Zaidman
a.e.zaidman@tudelft.nl

[1] Software Engineering Research Group, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

[2] Faculty of Information Technology, Esslingen University, Flandernstrasse 101, 73732 Esslingen, Germany

Published online: 14 August 2015

Keywords Residual defect · Fault localization · Online monitoring · Simulator · Service framework

1 Introduction

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.

Recent work (Chen et al. 2012) 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. 2009). 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 (Chen et al. 2012; 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 (Chen et al. 2012) 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. However, 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. 2010). However, in the case of a service-based system acting on 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 (Chen et al. 2012) reveals that the difficulty of tight coupling for the SFL approach can be resolved either by the architecture of the system and how services interact or by the granularity of the observations used for SFL. However, 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 paper* was to explore the trade-off between increasing the accuracy of the diagnosis in the case of tightly interacting faulty services on the one hand and the performance penalty on the running service system on the other hand. This current paper is an extension of our previous work presented in (Chen et al. 2013). 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 the current paper is the addition of a detailed analysis of runtime overhead caused by the different levels of monitoring. In the current paper, we concentrate on the following concrete research questions:

RQ1 How and to which extent can the monitoring granularity affect the correctness of SFL-based diagnosis for service-oriented systems?

RQ2 How can we increase the monitoring granularity for diagnosis in service-oriented systems?

RQ3 What is the overhead caused by the monitoring for diagnosis at 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 article is organized as follows: Sect. 2 presents the research field and techniques related to our approach. Sect. 3 outlines why tight service interaction inhibits the calculation of a diagnosis by SFL, and why increased monitoring granularity is adequate to alleviate this problem. Sect. 4 introduces the SFL simulator and explains how it can be used to assess the performance of our proposed approach quickly. Sect. 5 describes the case study used to assess our proposed approach. Sect. 6 presents the experiments measuring the runtime overhead caused by the monitoring of different levels of granularity. Sect. 7 discusses the experimental results and the limitations. Finally, Sect. 8 presents related work, and Sect. 9 concludes the paper.

2 Background

2.1 Spectrum-based fault localization

SFL infers a diagnosis from symptoms. Diagnosis refers to a ranking of potentially faulty components (source code lines, blocks, etc.). Symptoms are observations about component involvement in system activations, plus pass/fail information about the executions (Gonzalez-Sanchez et al. 2011). Component involvement is expressed in the form of so-called block hit spectra (hence the name spectrum-based fault localization). It produces for each system activation a binary coverage value per component (Reps et al. 1997; Zoeteweij et al. 2007) with covered $= 1$ and uncovered $= 0$. Component coverage can be derived

from a coverage tool. Each system activation, which may be referred to as test, leads to a spectrum, and it is associated with a binary verdict (pass = 0, fail = 1) from an oracle (Weyuker 1982). Execution of several tests produces an activity matrix, representing activation of each component over time. The test verdicts lead to a binary output vector with pass/fail information. The diagnosis is calculated through applying a similarity coefficient (SC) to each component activity vector and the output vector. The similarity denotes the likelihood of a component being the faulty one and, therefore, determines its position in the diagnosis ranking. Any SC may be used; however, the Ochiai SC has been found to work best (Abreu et al. 2006). Intuitively, SFL works by comparing the different combinations of component involvements in the individual system operations. Components that have not taken part in a system activation or are used more in passing activations are less likely to be faulty in case a failure is observed.

The basic SFL approach is illustrated in Table 1 by means of a simple Java program. This example system is comprised of components C_0–C_{10} with a source code line denoting the component granularity. It is exercised with six system activations, i.e., test cases or transactions, leading to the corresponding 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's activity vector. Finally, the similarity values are brought in a descending order. This results in C_4 being ranked top with 100 % likelihood, which represents the location of the fault in this example system (fault marked in bold).

2.2 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

Table 1 Illustration of SFL

Comp.	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	

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,[1] Redhat's JBoss,[2], or Ebay's Turmeric[3] 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.

2.3 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. 1, and it is briefly summarized in the following [more details in (Chen et al. 2012)].

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 services, such as SoapUI[4] and JMeter.[5] 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 system is built on Ebay's open-source service framework Turmeric.

1 http://axis.apache.org.

2 http://www.redhat.com/products/jbossenterprisemiddleware/.

3 https://www.ebayopensource.org/index.php/Turmeric.

4 http://www.soapui.org.

5 http://jmeter.apache.org.

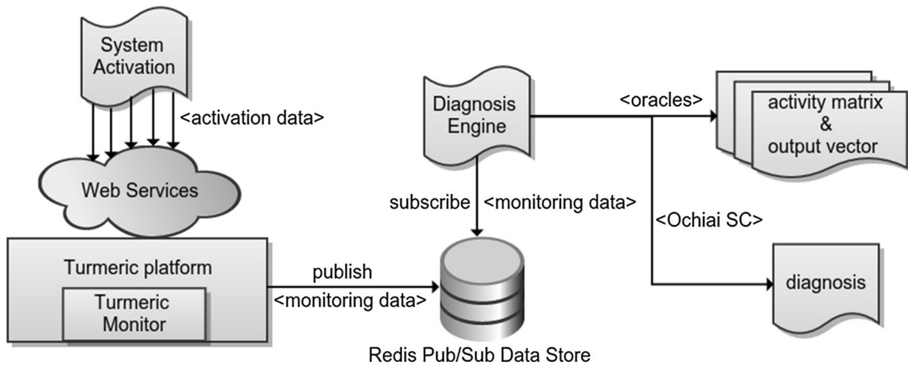


Fig. 1 Monitoring and diagnosis architecture based on Turmeric

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 database instance[6] 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 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. 1.

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

3 Problem statement and approach

One of the main targets of this paper was 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 Fig. 2. A topology is created by defining a number of components. Each component is defined by the component name and the component health (h). 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 Fig. 2, where each component has exactly one monitor, the monitoring topology corresponds to the service topology.

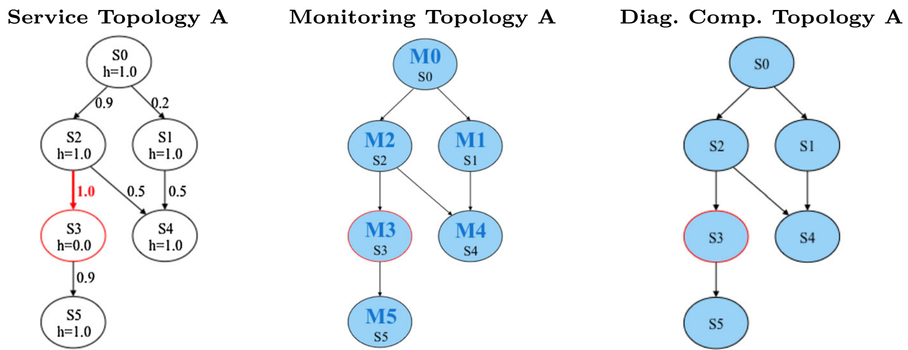
The *diagnosis component topology* then represents a virtual service topology in which the components of the service topology are split up in subcomponents in case multiple monitors per component are placed. This diagnosis component topology can discern multiple calling paths within a component in the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

3.1 The problem of tight service interaction

First, we explain how tight interaction aggravates diagnosis.

Consider the service topology on the left-hand side in Fig. 2, which is comprised of six services, S_0–S_5, with service S_3 being the faulty one with low health probability ($h = 0.0$). All other services are set to be 100 % healthy (healthy probability $h = 1.0$). 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. 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. 2.

The activity matrix and diagnosis results for this monitoring setup (produced with the SFL simulator, described later in Sect. 4) are presented in the table in Fig. 2. 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 Fig. 2, 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, in our approach, only a result that ranks the real faulty service uniquely highest in the diagnosis can be considered as a *correct diagnosis*. On the other hand, a result that ranks any healthy services highest is categorized as an *incorrect diagnosis*. In this example, tight interaction between services produces an ambiguous



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	

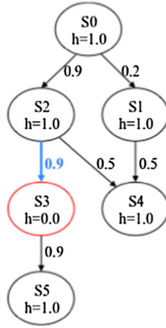
Fig. 2 Example topology illustrating tight service interaction

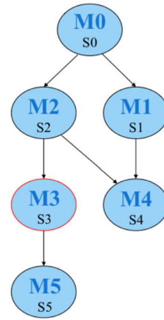
diagnosis, i.e., both a healthy service and the faulty service are ranked top, which is taken as an incorrect result by our definition.

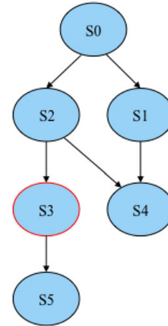
3.2 Solving tight service interaction: potential solution 1

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. 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 Fig. 3.

Service Topology B **Monitoring Topology B** **Diag. Comp. Topology B**







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	

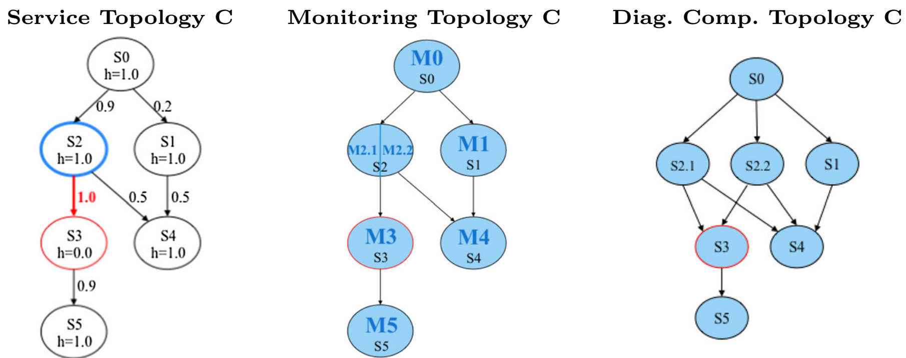
Fig. 3 Example topology illustrating potential solution 1

3.3 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 subcomponents, or simply adding components, and associating individual monitors to these subcomponents. 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 subcomponents, we retrieve finer-grained coverage information and finer-grained potential communication paths between the subcomponents, 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. 4. Here, service topology C corresponds to service topology A shown in Fig. 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 subcomponents: $S_{2.1}$ and $S_{2.2}$, as shown in the diagnosis component topology in Fig. 4. A possible way to realize this splitting is through code slicing. Both subcomponents lead to two separate observable paths from S_2 into S_3, and the corresponding activity matrix is also changed. 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 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 (Sect. 5)



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	

Fig. 4 Example topology illustrating potential solution 2

4 System simulations

4.1 SFL simulator

Performing experiments with a full-fledged case study is tedious. Every new experiment requires extensive adaptation to new experimental requirements. This leads us to the development of a simulator. It is 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.

Similarly to what we have explained in Sect. 3, 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. Different from the model that we used in Sect. 3, for the simulator we extend our model with a *failure probability*, which denotes the likelihood of a component to propagate an error into a failure, i.e., the fault observation. A failure probability between 0.0 and 1.0 means the likelihood for a component to issue a failure and terminate the transaction when it gets an error. The failure probability 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 activation will continue normally and issue a failed transaction at the end.

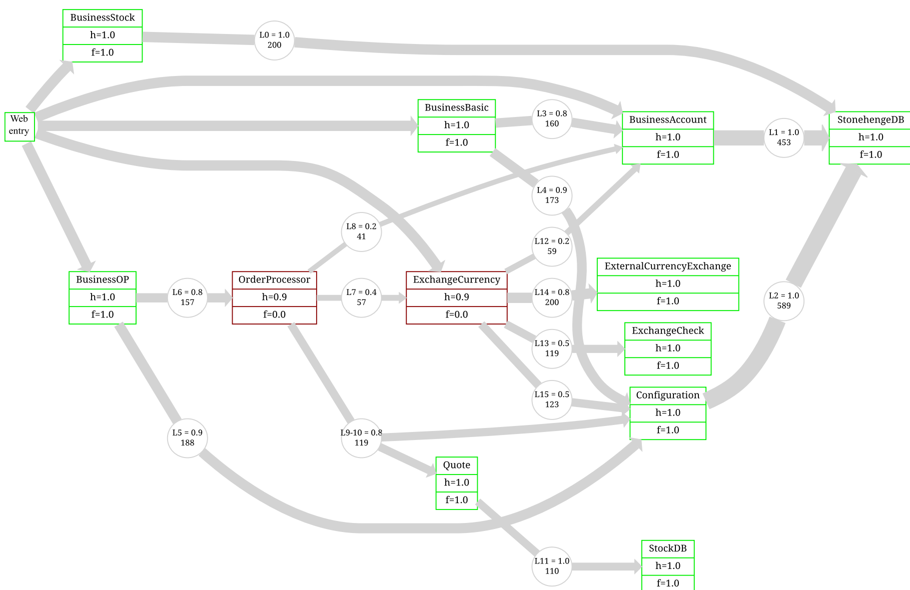


Fig. 5 Topology of the case study produced by the SFL simulator

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.

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, Fig. 5 displays an example topology of our case study system 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 5 also shows a particular instance after 200 transactions from the Web Application (denoted as “Web entry” 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 source code of the SFL simulator is available for download.[7] Its usage for the work described in this paper was twofold. First, we used it to develop our approach described in Sect. 3.3. Second, we applied it to simulate our original case system described in (Chen et al. 2012), for an initial assessment of our ideas in a more realistic setup (described below).

4.2 Simulation results

To assess our approach in a more realistic setup, we imitated our case study system with the SFL simulator. Different from the topology shown in Fig. 5, which is only 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 (Chen et al. 2012). 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 2. 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 2), and this corresponds to our original experiments described in (Chen et al. 2012). The second level is more detailed and separates service interfaces into finer-grained subcomponents (indicated as i_2 in Table 2). The ExchangeCurrencyService is split into five subcomponents and the OrderProcessorService is into seven subcomponents. The subcomponents, 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

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

Table 2 Simulation results for service diagnosis

Services	Component granularity	No. of activations	Diagnosis		% Correct diagnosis
			Correct	Incorrect	
ExchangeCurrencyService	i_1 Interface	50	8	42	16
	i_2 Subcomp	50	39	11	78
OrderProcessorService	i_1 Interface	50	13	37	26
	i_2 Subcomp	70	47	23	67

component/subcomponent can be set as faulty in one activation, so the number of activations in the simulation (Table 2) 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 2 illustrate the poor performance of SFL for tightly interacted services. A diagnosis is considered to be correct, if only the true faulty component is correctly and uniquely 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 interacted services. The simulation results for the finer-grained level of monitoring granularity (i_2, shown in Table 2) 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 6). This requires some explanation:

1. Compared to the case study, fewer faults are activated in the simulation (as shown in Table 3), 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 Sect. 5). The simulator allows to define topologies with finer-grained subcomponents; however, estimating the link probabilities and health values of these finer-grained subcomponents becomes increasingly difficult.

Table 3 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 Subcomp	11	5	6
OrderProcessorService	i_1 Interface	37	5	32
	i_2 Subcomp	23	5	18

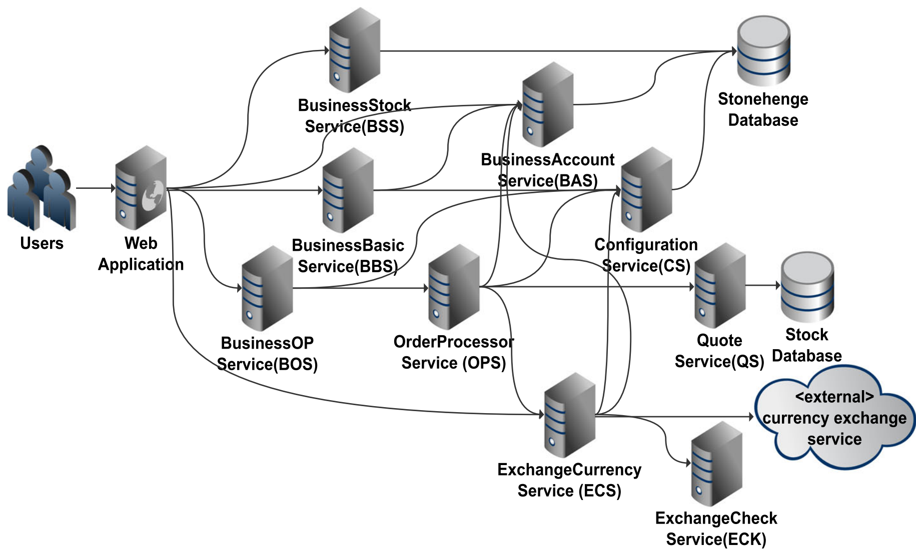


Fig. 6 Case study system: SFL stonehenge

All in all, the simulator always produces worse results when compared to the real case study, i.e., an approach being tested positive in simulation is more likely to receive positive results in real system. This is mainly due to the fact that it builds system topologies based on probabilities. 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 on a real system.

5 Case study

5.1 Case system

After having demonstrated in the simulator how an increase in the monitoring granularity of a system can support the calculation of a correct diagnosis, the next step is the evaluation of our proposed approach in a real service-based system. We use our original case study SFL Stonehenge[8] from (Chen et al. 2012; Espinha et al. 2012) and adapt it to the requirements implied by our problem statement. SFL Stonehenge is a service-based system simulating the stock market. It supports users in buying and selling of stocks, checking orders, and performing currency conversion operations for foreign stock acquisition.

Figure 6 illustrates the basic service architecture of the system. It is comprised of 10 web services including one *external currency exchange service*, plus a *web application* for user interaction. In addition, it accesses two data stores. The services provide the following operations. The *BusinessBasicService* and the *BusinessAccountService* provide the functions for user authentication, login, and the user account. The *BusinessOPService* and the

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

BusinessStockService are used for buying and selling stock, checking orders, and compiling market summaries. The *QuoteService* and the *OrderProcessorService* are used to process the stock orders placed by a user. The *ExchangeCurrencyService* and the *ExchangeCheckService* are responsible for the currency operations, and the *ConfigurationService* binds all the other services together, and acts like a registry.

In the following, we show typical service transactions that can be performed with our case system.

```
BusinessBasicService.login -->
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.getAccountProfile
    BusinessAccountService.updateAccountForLogin
BusinessBasicService.logout -->
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.updateAccount
BusinessBasicService.register -->
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.getAccountProfile
BusinessOPService.sell  -->
    ConfigurationService.getOPSLocations
    OrderProcessorService.submitOrder -->
        ConfigurationService.getQSLocations
        QuoteService.getQuotes
        ConfigurationService.getBSAccountLocations
        BusinessAccountService.updateWallet
ExchangeCurrencyService.exchCurrency -->
    ConfigurationService.getECheckLocations
    ExchangeCheckService.checkCurrency
    ExchangeCheckService.checkAmount
    ConfigurationService.getBSAccountLocations
    BusinessAccountService.updateWallet
```

5.2 Conducting the case study

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

Table 4 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 Mutators used in the two tightly interacted services

Services	Mutators (from Table 4)	No. of mutations
ExchangeCurrencyService (24 mutated versions)	1	5
	2	7
	4	12
OrderProcessorService (41 mutated versions)	1	15
	2	1
	3	1
	4	23
	5	1

which present tight interactions with other services. We apply the PIT mutation tool[9] in order to create 65 faulty system versions, 24 faulty versions for the *ExchangeCurrencyService*, and 41 faulty versions for the *OrderProcessorService*. Table 4 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 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 Sect. 2.3 and detailed in (Chen et al. 2012). In instance i_2, we invoke the same system and use the same Turmeric-based monitoring. Additionally, we also put monitors in the service

[9] http://pitest.org/.

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 subcomponents 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 subsection.

5.3 Case study results

Tables 6 and 7 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 6 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 subcomponents 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 6. Actually, the faults injected in both services can always be diagnosed correctly, leading to 100 % correct diagnoses. This

Table 6 Experimental results for service diagnosis

Services	Component granularity	No. of mutations	Diagnosis		% Correct diagnosis
			Correct	Incorrect	
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 7 Reasons for incorrect diagnoses in experiment

Services	Component granularity	Incorrect diagnoses	No activation	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

becomes apparent when we look at the reasons for the incorrect diagnoses shown in Table 7. In the first instance, i_1, 19 plus nine 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, two plus four 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 failure. The lower values of 92 and 90 % shown in Table 6 are a consequence of intermittent fault behavior of the services, a common property of software.

6 Runtime overhead

6.1 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, the analysis of diagnosis will not affect the system performance, and the main impact of our diagnosis approach on the runtime performance of service system is from the monitoring required for SFL. Therefore, we focus on determining the overhead of the online monitoring. In the experiments, we aim to measure the time overhead caused by the code block monitor for i_2 (subcomponent granularity), the time overhead caused by the Turmeric monitor for i_1 (service interface granularity), 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), the main function of which are introduced in Sect. 5.1. 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 setup. 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 Sect. 5). 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 8. Eventually, we decided to drop JMeter in favor of our own developed clients.

6.2 Overhead results

Table 9 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 9 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. 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 9.

The four center columns in Table 9 termed “Monitors” present the average response times for each service request to the service 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 and 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,[10], we observed that the first request that makes a service to invoke another new service always consumes extra overhead. However, 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 are 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 9, 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

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

Table 8 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

Table 9 Average end-to-end response time from ECS and OPS services in milliseconds over 1000 transactions

Serv. req.	No. of monitors		Monitors				Impact (%)		
	Turmeric	Code block	None	Code block	Turmeric	Turmeric and code block	Code block	Turmeric	Turmeric and 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

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. 7.2.

The overhead results presented in Table 9 are different from the results obtained in our previous overhead experiments outlined in our earlier article (Chen et al. 2013). 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 database maintaining the activity matrices, thereby adding considerable overhead through a suboptimal synchronous implementation. Moreover, earlier we used the EMMA coverage tool[11] 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 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.

The overhead measurements shown in Table 9 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 interservice 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 is listed in Table 10.

Table 10 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 9, 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[12] that our service 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.

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

[12] http://db.apache.org/derby/.

The impact percentages shown in Table 10 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.

Based on the results presented in Tables 9 and 10, we calculated the real value of overhead caused by the monitoring for each service. Table 11 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 0.8 ms; 10 code block monitors can cause overhead from 0.7 to 3.5 ms; and when the number of code block monitors is increased up to 100, the overhead also increases by 5.5 ms and 12.5 ms. Although the overhead from one and 10 code block monitors is 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 8. 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 12 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.

We also investigate the amount of monitoring data produced by each request, in order to see whether the throughput of monitors affects their overhead. Table 13 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 10, we notice that the data size and the impact of code block monitoring for BAS requests have exactly 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 are 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 14, 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.

The Turmeric monitor that we implemented for the experiments in (Chen et al. 2013) 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

Table 10 End-to-end response time from BAS service in milliseconds

Serv. req.	No. of monitors		Monitors				Impact (%)		
	Turmeric	Code block	None	Code block	Turmeric	Turmeric and code block	Code block	Turmeric	Turmeric and 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

Table 11 Monitoring overhead for code block monitor in milliseconds

Service	No. 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 12 Monitoring overhead for Turmeric monitor in milliseconds

Service	No. 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

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 15 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 Tables 9 and 10, and also presented in the Table 15. The last column of Table 15 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 13 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 B.	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 14 Data size versus 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

Table 15 Overhead for the logging part in Turmeric monitor in milliseconds

Service requests	No. of turm. moni.	With Turmeric, no code block monitoring data logging		Data logging over	Turmeric monitor overhead	%
		Acticated	Disabled			
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

7 Discussion and lessons learned

7.1 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 subcomponents would subsist across service boundaries, thereby invalidating our decoupling effort. This was not case. However, in the case study, some subcomponents within the services are still tightly interacted, so that the subcomponents 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 system. It is often not realistic that a software system only contains one fault. However, when applying online diagnosis for a service 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 system is practical and effective. Multiple faults in a service system can be found one by one as long as they cause a failure.

7.2 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 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 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 set up 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 10

shows negative impact by the code block monitors invoked during the execution of BAS_5. This becomes obvious, if we check Table 11. It demonstrates that the overhead caused by one code block monitor is less than 1 millisecond, Table 8, in which the standard deviation from the same request is nearly 10 ms. The same is true for the result of "101.09 %" for ECS_1 in Table 15, 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 9.

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 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 straightforward idea for reducing monitoring overhead, which is completely replacing the Turmeric monitors with code block monitors. Additionally, a code block monitor also produces 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.

7.3 Threats to validity

We are aware of a number of threats that might invalidate our findings. We use SFL Stonehenge 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.

Currently, we implement code block monitor with Redis pub/subfunctionality. 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 time stamps, this approach may not be applicable to service 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 cannot 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.

8 Related work

In this section, we briefly discuss the studies most relevant to diagnosis for service-based software systems. In particular, we start of by looking into other works that do diagnosis of service-based systems in Sect. 8.1. Subsequently, in Sect. 8.2, we look into whether alternative fault localization techniques are applied. Finally, in Sect. 8.3, 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.

8.1 Diagnosis for service-based systems

Chen et al. (2002) present *Pinpoint*, 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 interservice diagnosis, and the fact that services are used in different contexts.

Yan and Dague (2007), and 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. (2009, 2012) 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 was 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. 2010, 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.

8.2 Fault localization

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 (2007) use principal 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.

8.3 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. (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. (2010) 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 toward 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 toward contract violations and is more geared toward detecting the actual defect in a service composition.

9 Conclusion and future work

The goal of this paper was to investigate the trade-off between making the diagnosis of tightly interacting faulty services more accurate by increasing the monitoring granularity and the resulting performance penalty on the service system.

Referring to our research questions, we looked at:

RQ1: How and to what extent does the monitoring granularity affect the calculation of an SFL-based diagnosis? First, we used a simulator to reason over different service topologies. Second, we performed an actual case study on a SOA-based system, varying the level of monitoring granularity. The main conclusion 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.

RQ2: How can we increase the monitoring granularity for diagnosis? 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.

RQ3: What is the overhead caused by the monitoring for diagnosis at various levels of granularity? 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 toward 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 intraservice 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 system. Such monitor placement can achieve the highest accuracy of diagnosis and the least disturbance to the service 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.

Acknowledgments We would like to acknowledge NWO for sponsoring this research through the Jacquard ScaleItUp Project (Number 638.001.212). Also many thanks to our industrial partners Adyen and Exact.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

Abreu, R., Zoeteweij, P., & van Gemund, A. J. (2006). An evaluation of similarity coefficients for software fault localization. In *Proceedings of international symposium on dependable computing (PRDC), IEEE* (pp. 39–46).

Abreu, R., Zoeteweij, P., Golsteijn, R., & van Gemund, A. (2009). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, *82*(11), 1780–1792.

Baresi, L., & Guinea, S. (2013). Event-based multi-level service monitoring. In *IEEE 20th international conference on web services (ICWS), 2013* (pp. 83–90).

Baresi, L., Ghezzi, C., & Guinea, S. (2007). Towards self-healing composition of services. In B. J. Krämer & W. A. Halang (Eds.), *Contributions to Ubiquitous Computing, Studies in Computational Intelligence* (pp. 27–46). New York: Springer.

Bennett, K., Layzell, P., Budgen, D., Brereton, P., Macaulay, L., & Munro, M. (2000). Service-based software: the future for flexible software. In *Proceedings of asia-pacific software engineering conference (APSEC), IEEE* (pp. 214–221).

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

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

Chen, C., Gross, H. G., & Zaidman, A. (2012). Spectrum-based fault diagnosis for service-oriented software systems. In *Proceedings of the international conference on service-oriented computing and applications (SOCA), IEEE* (pp. 1–8).

Chen, C., Gross, H. G., & Zaidman, A. (2013). Improving service diagnosis through increased monitoring granularity. In *7th international conference on software security and reliability (SERE), IEEE* (pp. 129–138).

Chen, M., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: problem determination in large, dynamic internet services. In *Proceedings of international conference on dependable systems and networks (DSN), IEEE* (pp. 595–604).

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

Espinha, T., Chen, C., Zaidman, A., & Gross, H. G. (2012). Maintenance research in SOA—Towards a standard case study. In *Proceedings of European conference on software maintenance and reengineering (CSMR), IEEE* (pp. 391–396).

Feldman, A., Provan, G. M., & van Gemund, A. J. C. (2010). Approximate model-based diagnosis using greedy stochastic search. *Journal of Artificial Intelligence Research*, *38*, 371–413.

Gonzalez-Sanchez, A., Piel, E., Gross, H. G., & van Gemund, A. (2010). Prioritizing tests for software fault localization. In *International conference on quality software, IEEE* (pp. 42–51).

Gonzalez-Sanchez, A., Abreu, R., Gross, H. G., & van Gemund, A. J. (2011). Spectrum-based sequential diagnosis. In *Proceedings of international conference on artificial intelligence (AAAI)* (pp. 189–196), AAAI Press.

Grosclaude, I. (2004). Model-based monitoring of component-based software systems. In *International workshop on principles of diagnosis* (pp. 155–160).

Heward, G., Muller, I., Han, J., Schneider, J. G., & Versteeg, S. (2010). Assessing the performance impact of service monitoring. In *Software engineering conference (ASWEC), 2010 21st Australian* (pp. 192–201).

Keller, A., & 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.

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

Mayer, W., Friedrich, G., & Stumptner, M. (2010). Diagnosis of service failures by trace analysis with partial knowledge. In Service-oriented computing, LNCS (Vol. 6470, pp. 334–349). Berlin: Springer.

Mayer, W., Friedrich, G., & Stumptner, M. (2012). On computing correct processes and repairs using partial behavioral models. In *20th European conference on artificial intelligence (ECAI)* (pp. 582–587).

Mohamed, A., & Zulkernine, M. (2008). On failure propagation in component-based software systems. In *Proceedings of international conference on quality software (QSIC), IEEE* (pp. 402–411).

Mosincat, A. D., & Binder, W. (2011). Automated maintenance of service compositions with SLA violation detection and dynamic binding. *International Journal on Software Tools for Technology Transfer*, *13*(2), 167–179.

Repp, N., Berbner, R., Heckmann, O., & Steinmetz, R. (2007). A cross-layer approach to performance monitoring of web services. In *Emerging web services technology* (pp. 21–32).

Reps, T., Ball, T., Das, M., & Larus, J. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *European software engineering conference symposium on foundations of software engineering (ESEC/FSE), LNCS* (Vol. 1301, pp. 432–449). Springer.

Weiser, M. (1981). Program slicing. In *Proceedings of the international conference on software engineering (ICSE)* (pp. 439–449). IEEE Press.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, *25*(4), 465–470.

Wong, W. E., Debroy, V., & 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., & Dague, P. (2007). Modeling and diagnosing orchestrated web service processes. In *Proceedings of international conference on web services (ICWS), IEEE* (pp. 51–59).

Yan, Y., Dague, P., Pencole, Y., & 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.

Zhang, J., Chang, Y., & Lin, K. J. (2009). A dependency matrix based framework for QoS diagnosis in SOA. In *Proceedings of international conference on service-oriented computing and applications (SOCA), IEEE* (pp. 1–8).

Zhang, J., Huang, Z., & Lin, K. (2012). A hybrid diagnosis approach for QoS management in service-oriented architecture. In *International conference on web services (ICWS), IEEE* (pp. 82–89).

Zoeteweij, P., Abreu, R., Golsteijn, R., & van Gemund, A. J. (2007). Diagnosis of embedded software using program spectra. In *Proceedings of international conference and workshops on engineering of computer-based systems (ECBS), IEEE* (pp. 213–220).

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

Cuiting Chen graduated with a Computer Science Master's degree from Dresden University of Technology, Germany, in 2010. She currently works as a PhD student at Delft University of Technology, the Netherlands. Her main research interests include software testing and software evolution.

Hans-Gerhard Gross received an MSc in Computer Science (1996) from the Beuth University of Applied Sciences, Berlin, Germany, and a PhD in Software Engineering (2000) from the University of Glamorgan, Wales, UK. Following his PhD, Dr. Gross joined the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany, where he was responsible for a number of public research projects and consulting projects with German software organizations. From 2005 to 2013, Dr. Gross was employed as Assistant Professor at Delft University of Technology, the Netherlands. Since 2013, Dr. Gross is working as Professor at Esslingen University, Germany. His research interests encompass all phases of software development, and software evolution, and software testing, in particular.

Andy Zaidman is an Associate Professor at the Delft University of Technology, the Netherlands. He obtained his MSc (2002) and PhD degree (2006) in Computer Science from the University of Antwerp, Belgium. His main research interests are software evolution, program comprehension, mining software repositories, and software testing. He was the general chair of the 15th Working Conference on Reverse Engineering (WCRE 2008) held in Antwerp, Belgium, program co-chair of WCRE 2009 held in Lille, France, and program co-chair of VISSOFT 2014 held in Victoria, BC, Canada. In 2013, Andy Zaidman was the laureate of a NWO Vidi career grant.