Improving Service Diagnosis Through Increased Monitoring Granularity

Cuiting Chen, Hans-Gerhard Gross and Andy Zaidman
Delft University of Technology, the Netherlands
Email: {cuiting.chen;h.g.gross;a.e.zaidman}@tudelft.nl

***Abstract*—Due to their loose coupling and highly dynamic nature, service-oriented systems offer many benefits for realizing fault tolerance and supporting trustworthy computing. They enable automatic system reconfiguration in case that a faulty service is detected. Spectrum-based fault localization (SFL) is a statistics-based diagnosis technique that can effectively be applied to pinpoint problematic services. It works by monitoring service usage in system transactions and comparing service coverage with pass/fail observations.**

SFL exhibits poor performance in diagnosing faulty services in cases when services are tightly coupled. In this paper, we study how and to which extent an increase in monitoring granularity can help to improve correct diagnosis of tightly coupled faulty services. We apply SFL in a real service-based system, for which we show that 100% correct identification of faulty services can be achieved through an increase in the monitoring granularity.

***Keywords*-residual defect, fault localization, online monitoring, simulator, service framework;**

I. 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 for such systems in a natural way. This disposition of service-based systems makes the handling of emerging problems to support fault tolerance straightforward. If a service misbehaves during operation it can be exchanged for another healthy service through simple runtime reconfiguration [1], [2]. 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 [3]. Even though service-oriented systems provide all the ingredients necessary to recover from and adapt to operation time failures [4], adequate runtime diagnosis approaches that identify a faulty service accurately are still missing. Diagnosis for services has been proposed in the past [5], [6], but the techniques are based mainly on static system modeling, disregarding the dynamic nature of service-based systems.

Recent work [7] demonstrates that spectrum-based fault localization (SFL), a statistics-based diagnosis technique can effectively be used in order to pinpoint faulty components in service-based systems. SFL works by automatically inferring a diagnosis from observed symptoms [8]. 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 [7], [9]. 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 can effectively be used in diagnosing faulty services, experiments performed for our previous work [7] show that incorrect diagnoses are more likely, if services are tightly coupled. In other words, if a service S_1 always invokes another service S_2, the diagnosis is such that both services S_1 and S_2 will be convicted, leading to incorrect or inconclusive diagnoses. In a traditional setting with a human diagnostician, this is not so much of an issue. Since it would mean that more services would have to be inspected, in order to determine the true root cause of failure, thereby merely increasing the residual diagnosis cost [10]. However, in the case of an autonomously acting fault-tolerant system, it would mean that reconfiguration or other self-healing activities would be applied to more suspects, thereby unnecessarily treating healthy services.

Careful analysis of the experiments performed for [7] reveals that the difficulty of tight coupling for the SFL approach can be explained either by the architecture of the system and how services interact, or by the granularity of the observations used for SFL. Whereas, in the first instance, it would be rather difficult to try and rearrange the architecture in order to decouple services for any individual system configuration; in the second instance, it would be relatively easy to introduce more observation points in the architecture, and thus increase the level of granularity, in order to support the calculation of a conclusive diagnosis. As a consequence, the goal of this paper is to explore the effects of changing the level of observation granularity and assess its impact on the calculation of a diagnosis. We concentrate on the following concrete research questions:

- **RQ1** How does the observation granularity affect the calculation of an SFL-based diagnosis?
- **RQ2** How is the granularity determined? In other words, what are good locations for monitoring?
- **RQ3** To which extent can incorrect diagnoses be resolved, and what is the overhead incurred?

We make the following contributions. We describe an approach and implementation for increasing the observation granularity in services, and show how this can improve the accuracy of diagnosing faulty services. We introduce a novel SFL simulator. This is used to study the effects of changing the observation 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. II presents the research field and techniques related to our approach. Sect. III 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. IV introduces the SFL Simulator and explains how it can be used to assess the performance of our proposed approach quickly. Sect. V describes the case study used to assess our proposed approach. Sect. VI discusses the experimental results and the limitations. Finally, Sect. VII presents related work and Sect. VIII concludes the paper.

II. Background

A. Spectrum-based Fault Localization

SFL infers a diagnosis, i.e. a ranking of potentially faulty components (source code lines, blocks, etc.), from symptoms, i.e. observations about component involvement in system executions, plus pass/fail information about the executions [9]. Component involvement is expressed in the form of block-hit-spectra (hence the name Spectrum-based Fault Localization), producing for each execution a binary coverage value per component [11][12] with covered=1 and not covered=0. This can be derived from a coverage tool. Each system execution, i.e. a test, leads to a spectrum and is associated with a binary verdict (pass=0, fail=1) from an oracle. 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 ranking. Any SC may be used; however, the Ochiai SC has been found to work best [13]. Intuitively, SFL works by comparing the different combinations of component involvements in the individual system operations. Components that have not taken part in an activity or are used more in passing activities are less likely faulty in case a failure is detected.

The basic SFL approach is illustrated in Table I by means of a simple Ruby program. This example system is comprised of components $C_0 - C_{10}$ with a source code line as component granularity. It is exercised with 6 system executions, 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_3 being ranked top with 100% likelihood, which represents the location of the fault in this example system (fault marked in bold).

Table I
ILLUSTRATION OF BASIC SFL

C	Character counter	t_1	t_2	t_3	t_4	t_5	t_6	SC
	def count(string)	[Activity Matrix]						
C_0	let = dig = other = 0	1	1	1	1	1	1	0.87
C_1	string.each_char { |c|	1	1	1	1	1	1	0.87
C_2	if c===/[A-Z]/	1	1	1	1	0	1	0.93
C_3	**let += 2**	1	1	1	1	0	0	1.00
C_4	elsif c===/[a-z]/	1	1	1	1	0	1	0.93
C_5	let += 1	1	1	0	0	0	0	0.71
C_6	elsif c===/[0-9]/	1	1	1	1	0	1	0.93
C_7	dig += 1	0	1	0	1	0	0	0.71
C_8	elsif not c===/[a-zA-Z0-9]/	1	0	1	0	0	1	0.47
C_9	other += 1 }	1	0	1	0	0	1	0.47
C_{10}	return let, dig, other	1	1	1	1	1	1	0.87
	end							
	Output vector (verdicts)	1	1	1	1	0	0	

B. SFL for Service-based Systems

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

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

Activation in service-based systems is not so obvious, and it cannot be done through exercising test cases. Because a service instance can serve many application contexts, it will not be exclusively activated from within one application, but from a potentially arbitrary number of other applications operating in other contexts. Applying SFL in a service-based system requires a system execution to be made explicit through a unique transaction ID, which separates the service executions 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/operations with their corresponding transactions. This can either be done by the services themselves or through modern service frame-

works. 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.

C. Implementation of SFL for Service-based Systems

We base our implementation on Ebay's open source service framework Turmeric.[3] It offers many inbuilt features that support the (online) collection of system data required for applying SFL in service-based systems. This allows the implementation of online monitoring to record the block-hit spectra for SFL with minimum amendments, yielding a slender design. Our prototypical implementation is summarized briefly in the following (more details in [7]).

Typically, services would be activated at the application interface through user interaction. However, in our case, system activation is automated through SoapUI[4] combined with JMeter[5] for evaluation purposes. These tools are used to create SOAP messages and execute them automatically, thereby mimicking real user interaction.

Involvement of a service in a transaction is derived from Turmeric's online monitoring facilities [14]. It provides a specific pipeline message-handling mechanism, which can be extended by additional custom-built handlers dedicated to monitoring incoming and outgoing traffic of a service. Combined with a unique transaction ID, the monitors can associate transactions with the respective service handlers that process the transaction, resulting in service activation information per unique transaction.

Verdicts are also generated based on Turmeric's message handling facility. Dedicated monitors are used to log upcoming exceptions or other noteworthy events and outcomes into a data store for further analysis. Thus, any one of these noteworthy occurrences can be associated with a unique transaction ID and be used for calculating a diagnosis.

The actual diagnosis is conducted offline in a diagnosis engine. It is designed as a separately operating application that collects the service involvement information and verdicts produced by the 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.

[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

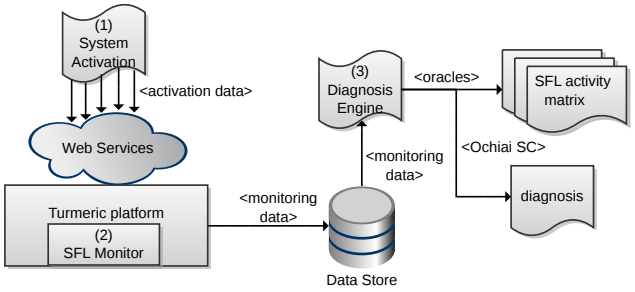

Figure 1. Monitoring and diagnosis architecture based on Turmeric

III. Problem Statement and Approach

A. The Problem of Tight Service Coupling

The goal of this paper is to study the effects of tight service coupling on the calculation of a diagnosis. Topology A displayed in Fig. 2 illustrates these effects. Topologies B and C demonstrate how tight coupling may be resolved in order to retrieve definite diagnoses. The topologies shown in Fig. 2 and their corresponding diagnoses (Table II) are produced with the SFL Simulator described later in Section IV. The topologies are comprised of six services, S_0 – S_5, with service S_3 being the faulty one with low health (h=0.0). All other services are set to be 100% healthy. Every service represents an observation point, which produces coverage information for the corresponding activity matrices.

In topology A (Fig. 2), services S_0, S_2, S_3 and S_5 are tightly coupled, indicated through the 1.0 invocation probabilities between them. Once S_0 is invoked, S_2, S_3 and S_5 will also always be invoked, leading to a diagnosis that blames not only the faulty service S_3, but also all of its tightly coupled peers. The first activity matrix in Table II illustrates this typical behavior of SFL. Three of the four tightly linked services are assigned the same similarity coefficient, and thus, the same rank in the diagnosis, with Ochiai SC = 1. Because tight coupling does not produce discriminative coverage information, it always leads to incorrect, and in this case, ambiguous diagnoses. It should be noted that service S_5 does not appear in the ranking shown in Table II, because it is not executed if S_3 fails, i.e. with a fatal failure. This simulation was set to stop a transaction in case of a fatal failure.

An easy way to relax tight interactions would be the reduction of the invocation probabilities between the services. Topology B in Fig. 2 shows invocation probabilities of the originally tightly-coupled services reduced to 0.9 each. This slight adjustment in the invocation probabilities leads to so much more decoupling of the services and introduction of more discriminative information in the observations, that a correct and unambiguous diagnosis can be calculated in the related activity matrix for Topology B in Table II. Both, fatal failure, i.e. stop the failing transaction, and warning,

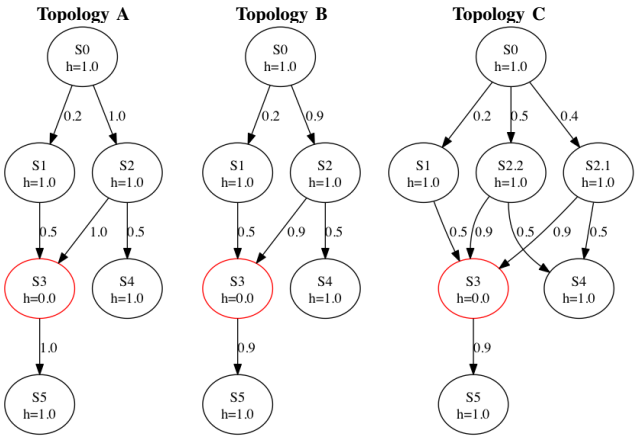

Figure 2. Example topologies illustrating tight service coupling (A) and potential decoupling (B, C)

i.e. continue the erroneous transaction, are simulated. In the warning case, this leads to occasional execution of service S_5, which represents a more difficult diagnosis. However, the diagnosis is correct and unambiguous in both cases.

B. 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 topology must be invented that leads to similar results as shown for Topology B. Experiments with the SFL Simulator suggest that this may be achieved through increasing the number of observation points in the topology, which boils down to splitting services into sub-components, or simply adding components. This increases the level of detail, i.e. the observation granularity used for the similarity coefficients. By splitting services into finer-grained components, we retrieve finer coverage information, plus finer grained communication between the sub-components, with different invocation probabilities between them.

This increase in observation granularity is illustrated in Topology C (Fig. 2) and its effect shown in its corresponding activity matrix. Service S_2 is split into two sub-components, $S_{2.1}$ and $S_{2.2}$, increasing the number of observation points. When looking at both sub-components of S_2 individually, we observe that the execution towards the faulty service S_3 can follow two internal routes with (hopefully) different invocation probabilities. In both topologies, B and C, S_3 can be reached through S_2, and this can be observed. However, in Topology C, two separate observable paths lead from S_2 into S_3. SFL exploits this additional information. Whether, and to which extent the invocation probabilities will change between the new sub-components through increasing the observation granularity depends on careful selection of the observation locations. This requires further investigation when performing a case study (Sect. V).

Table II
EXAMPLE ACTIVITIES/SIMILARITIES FOR THE TOPOLGIES IN FIG. 2

Service	Activity for Topology A (fatal failure)	Ochiai SC
	Topology A	
S5	00000000000000000000000000000000000000000000	0.000
S1	10001111010000011000010100101100000000100	0.592
S4	10011111011111001101100010101110001001 00	0.742
S3	**11111111111111111111111111111111111111111**	**1.000**
S2	**11111111111111111111111111111111111111111**	**1.000**
S0	**11111111111111111111111111111111111111111**	**1.000**
Errors	11111111111111111111111111111111111111111	
	Topology B	
Service	Activity for Topology B (fatal failure)	Ochiai SC
S5	00000000000000000000000000000000000000000000	0.000
S1	00000000000001000010000000110000100001 00	0.340
S4	00111010011000000011101010111001001010 01	0.668
S0	11111111111111111111111111111111111111111	0.949
S2	11111011111110111111111111111111111111111	0.973
S3	**11111011111110101111111111110111111111111**	**1.000**
Errors	11111011111110101111111111110111111111111	
Service	Activity for Topology B (warning)	Ochiai SC
S1	10000100000000100000000100100000011100000	0.485
S4	10111101001001001001100100000001111011001	0.590
S0	11111111111111111111111111111111111111111	0.922
S5	10101111100010111111111110110111111110010	0.924
S2	11111111101011111111111111111111111111111	0.946
S3	**11101111100010111111111111111111111111110**	**1.000**
E	11101111100010111111111111111111111111110	
	Topology C	
Service	Activity for Topology C (fatal failure)	Ochiai SC
S5	00000000000000000000000000000000000000000000	0.000
S1	10100000000000000010000001000000000000	0.344
S4	00010000001100011011100101001110000000000	0.445
S2.2	10010010000001110010111101101110011 10101	0.601
S0	11111111111111111111111111111111111111111	0.689
S2.1	10011001001110011101101110101100001 00110	0.851
S3	**10010001000010111101101110100100011 00110**	**1.000**
Errors	10010001000010111101101110100100011 00110	

IV. SYSTEM SIMULATIONS

A. SFL Simulator

Performing experiments with a fully fledged case study is tedious. Every new experiment requires extensive adaptation to new experimental requirements. This lead 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.

A topology is created by defining a number of components. Each component is defined by the component name, component health, and failure probability. Health denotes the probability that a component will not produce an error when it is executed. 1.0 represents a healthy component, while a value in the range (0.0, 1.0) represents a faulty

component with intermittent fault behavior. 0.0 denotes no fault intermittency, i.e., the component will always produce an error if activated. Failure probability denotes the likelihood of a component to propagate an error into a failure, i.e. the fault observation. 1.0 means that if a component encounters an error, this component will issue a failure, and the simulated execution will be stopped. This can also be used to discriminate fatal failures (i.e. component health < 1.0 and failure probability $= 1.0$) from warnings (i.e. failure probability = 0.0). In the case of a warning, the system execution will continue normally and issue a failed transaction at the end.

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

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

For illustration purposes, Figure 3 displays an example topology of our case study system produced by the SFL Simulator. It is a more elaborate diagram than the ones displayed in Fig. 2, and 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 3 also shows a particular instance after 200 transactions from the Web Application (denoted as "Web" at the left hand side of the figure). The whole numbers in the link nodes denote the frequencies of invocations, and the thickness of each line also indicates this.

The source code of the SFL Simulator is available for download.[6] Its usage for the work described in this paper was twofold. First, we used it to develop our approach described in Sect. III-B. Second, we applied it to simulate our original case system described in [7], for an initial assessment of our ideas in a more realistic setup (described below).

B. Simulation Results

To assess our approach in a more realistic setup, we imitated our case study system with the SFL Simulator. In contrast to the topology shown in Fig. 3, which is only displaying top-level services, we used a more detailed model, looking at the service interface level. This follows the original design of the case study system [7]. In addition, the link probability in the simulations is based on the service implementation logic plus the test data applied, and the health intermittency is determined based on the number of fault activations during test of the real system. Figure 3 shows a reduced topology of the simulated system (due to space limitations).

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

Table III
SIMULATION RESULTS FOR SERVICE DIAGNOSIS

Services	Component Granularity	# of Activ.	Diagnosis Correct	Diagnosis Incor.	Correct Diagn.
ExchangeCurrency Service	i_1 Interface	50	8	42	16%
	i_2 Sub-comp	50	39	11	78%
OrderProcessor Service	i_1 Interface	50	13	37	26%
	i_2 Sub-comp	70	47	23	67%

Table IV
REASONS FOR INCORRECT DIAGNOSES IN SIMULATION

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

In the original experiments, two services could be identified to exhibit the problem of tight service interaction, i.e. *ExchangeCurrencyService* and *OrderProcessorService*, resulting in incorrect diagnoses. The results of the simulations performed for these two services are shown in Table III. 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 III), and this corresponds to our original experiments described in [7]. The second level is more detailed and separates service interfaces into finer grained sub-components (indicated as i_2 in Table III). In order to obtain a finer level of granularity, the *ExchangeCurrencyService* is split into 2 sub-components and the *OrderProcessService* is split into 7 sub-components. The sub-components 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. The number of activations in the simulation (Table III) is set to 50 and 70, respectively, in order to retrieve sufficient fault coverage.

The low values for correctly performed diagnoses for granularity i_1 shown in Table III illustrate the poor performance of SFL for tightly coupled services. A diagnosis is considered to be correct, if the true faulty component is correctly and unambiguously identified by the SFL diagnosis. In the initial setup (with interface-level granularity, i_1), this can only be achieved in 16% and 26% of the cases for the

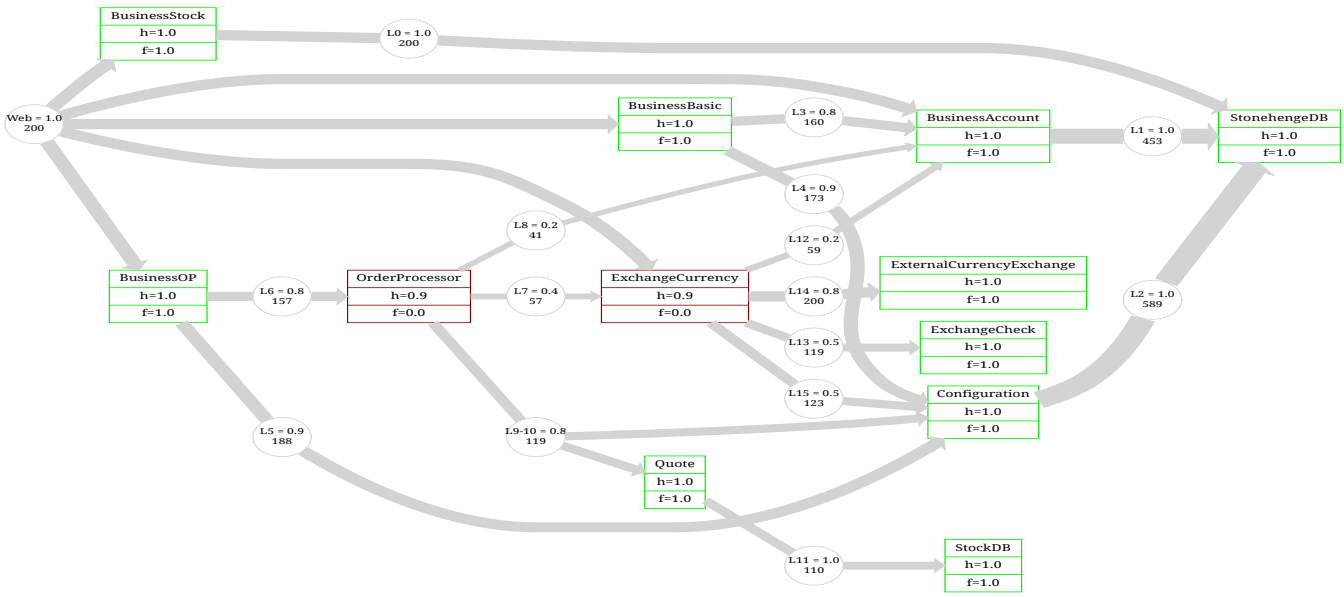

Figure 3. Topology of the case study produced by the SFL Simulator

two tightly coupled services. The simulation results for the finer-grained level of observation granularity (i_2, shown in Table III) 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 VII). This requires some explanation:

1) Compared to the case study, fewer faults are activated in the simulation (as shown in Table IV), 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 do not reflect the usage profile imposed by the real test cases accurately. This leads to statistically significant deviations of the executions in the simulation compared to the real system.
3) The observation granularity in the real case system is increased compared with the simulation (see Sect. V). The simulator allows to define topologies with finer-grained sub-components, however, estimating the link probabilities and health values of these finer-grained sub-components becomes increasingly more difficult.

All in all, the simulations suit our purpose in that they confirm a positive effect of introducing more observation points for the calculation of the diagnosis. In the following section, we describe how our approach is evaluated in a real system.

V. Case Study

A. Case System

After having demonstrated in the simulator how an increase in the observation granularity of a system can support the calculation of an unambiguous 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[7] from [7], 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 stock, checking orders, and performing currency conversion operations for foreign stock acquisition.

Figure 3 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. *BusinessBasicService* and *BusinessAccountService* provide the functions for user authentication, login, and the user account. *BusinessOPService* and *BusinessStockService* are used for buying and selling stock, checking orders, and compiling market summaries. *QuoteService* and *OrderProcessorService* are used to process the stock orders placed by a user. *ExchangeCurrencyService* and *ExchangeCheckService* are responsible for the currency operations, and the *ConfigurationService* binds all the other services together, and acts like a registry.

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

Table V
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 VI
MUTATORS USED IN THE TWO TIGHTLY COUPLED SERVICES

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

B. 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 tightly coupled services, *ExchangeCurrencyService* and *OrderProcessorService*. We apply the PIT mutation tool[8] in order to create 65 faulty service versions, 24 faulty versions of *ExchangeCurrencyService*, and 41 faulty versions of *OrderProcessorService*. Table V 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 VI 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[9] 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 *ExchangeCurrencyService* or in *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. II-C and detailed in [7]. In instance i_2, we invoke the same system and use the same Turmeric-based monitoring. Additionally, we also use the *EMMA* Java code coverage tool[10] to instrument the two services. *EMMA* is capable of providing source code line coverage information. However, we choose to retrieve code block coverage information. We determine the code blocks based on the internal control-flow structure of the service implementations. In addition, in some cases we separate the blocks for better isolation of tightly-coupled code sections. This results in 10 sub-components each for both services. We believe that code-block level of granularity is sufficient for our purpose since it separates a service into units that can be discriminated in the calculation of the diagnosis.

That way, we are able to increase the number of observation points in instance i_2 to the highest level of granularity required. The additional monitoring introduces more and more diverse coverage information, which we expect will yield better suited activity matrices, thus, leading to better diagnoses. The results of these experiments are presented in the following sub-section.

C. Case Study Results

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

The improvement of the finer-grained observation 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 VII. Actually, the faults injected in both services can always be diagnosed correctly, leading to 100% correct and unambiguous diagnoses. This becomes apparent when we look at the reasons for the incorrect diagnoses shown in Table VIII. In the first instance, i_1, 19 plus 9 out of the total number of incorrect diagnoses of the two services produced wrong results because of tight interaction on failure. This represents our original problem, and the table indicates that it can be resolved entirely through increasing the monitoring granularity for the considered services in the second instance, i_2. In both instances, i_1 and i_2, 2 plus 4 out of the total number of incorrect diagnoses are due to the faults in the services *not* being activated. In other words, in these cases no test execution was able to cover the faults introduced through the mutations. In general, diagnosis can only be initiated when a fault is actually detected. This is

[8]http://pitest.org/

[9]http://jmeter.apache.org

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

Table VII
EXPERIMENTAL RESULTS FOR SERVICE DIAGNOSIS

Services	Component Granularity	# of Mut.	Diagnosis Correct	Diagnosis Incor.	Correct Diagn.
ExchangeCurrency-Service	i_1 Service Interface	24	3	21	13%
	i_2 Code Block	24	22	2	92%
OrderProcessor-Service	i_1 Service Interface	41	28	13	68%
	i_2 Code Block	41	37	4	90%

Table VIII
REASONS FOR INCORRECT DIAGNOSES IN EXPERIMENT

Services	Component Granularity	Incorrect Diagnoses	No Activation	Tight Interaction on Failure
ExchangeCurrency-Service	i_1 Service Interface	21	2	19
	i_2 Code Block	2	2	0
OrderProcessor-Service	i_1 Service Interface	13	4	9
	i_2 Code Block	4	4	0

not attributable to our diagnosis technique, but a fundamental problem of all coverage-based quality assurance approaches.

We can, therefore, 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 VII are a consequence of intermittent fault behavior of the services, a common property of software.

VI. DISCUSSION AND LESSONS LEARNED

A. General 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. Further, 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 better convicting components that participate in failing transactions and exonerating components that participate in passing transactions.

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

Here, an important lesson learned is that we can reduce tight coupling on the higher level of granularity, i.e. between services, but we cannot remove it entirely on the lower levels of granularity, e.g. within services. We acknowledge the fact that topology plays a major role in the successful application of spectrum-based fault localization in service-based systems. In the future, we will look at other methods of topological separation, for example program slicing techniques [16].

B. Correct Diagnosis of Missing Faulty Components

Six mutations of instance two in our case study exhibit strange behavior. The faulty services are diagnosed correctly, but their corresponding faulty code blocks are not covered according to the code block monitors, even though the fault is triggered. Careful inspection of these six exceptions leads to an explanation.

The *EMMA* code coverage tool calculates code coverage based on the basic block level, instead of the code line level. It considers a block as executed, if the block's last byte code instruction is executed.[11] Unfortunately, the mutations injected by the PIT mutation tool may cause the service to be interrupted in the middle, when it is executing the faulty code block. It leads to the phenomenon that *EMMA* does not cover the faulty block while it is actually executed.

This unfortunate interference leads to an interesting lesson learned: tight coupling is not only a curse, but it can also be a cure, particularly in cases where not all sub-components can be monitored separately. If two components are tightly coupled, the monitored component may well hint to a problem in its tightly coupled but unmonitored peers. In such a case, two tightly coupled individual components are correctly treated as one big component in the diagnosis.

C. Runtime Overhead

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, in Table IX, we focus on the overhead of the runtime monitoring required for SFL. The table shows average end-to-end process response times of 100 transactions of four representative system requests. We chose the requests based on diversity in service interactions they will create. Both services exhibit four fundamentally different combinations of interactions with other services. The values are measured by JMeter on a standard (non-real-time) Linux PC platform with Turmeric and *EMMA* installed. The measurements indicate that the overhead for service-level monitoring introduced by Turmeric is huge, whereas the

[11] http://emma.sourceforge.net/faq.html#q.blockcoverage

Table IX
OVERHEAD: TURMERIC/EMMA MONITORING IN MILLISECONDS

Service Requests	No Turmeric Monitor		With Turmeric Monitor	
	No Emma	With EMMA	No Emma	With EMMA
Exch_Req_1	36	36	315	326
Exch_Req_2	7	7	64	69
Exch_Req_3	49	48	400	417
Exch_Req_4	30	31	249	246
Order_Req_1	77	77	514	501
Order_Req_2	86	88	459	470
Order_Req_3	54	49	271	281
Order_Req_4	39	38	276	271

additional block-level monitoring overhead introduced by *EMMA* is minute.

After having reviewed our case study, it becomes apparent that our admittedly poor prototypical implementation is responsible for the disproportionately high values measured for the service-level monitoring granularity in Turmeric. As major inhibiting factor, we can identify the large number of synchronous database accesses realized in our monitors. In future experiments, we will replace this inefficient storage implementation through an asynchronous publish-subscribe solution, e.g. based on Redis.[12]

D. 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 mentioned the topology problem earlier as future work.

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.

The PIT mutation testing tool is provided by a third party. PIT is still under constant development and being improved, but we consider it to be reliable.

VII. RELATED WORK

Chen et al. present *Pinpoint* [17], a similar diagnosis approach plus a tool using similarity coefficients in order to infer a diagnosis from system activation and component involvement. However, even though their title suggests otherwise, they do not address the specific issues of diagnosing services, i.e. the problems of inter-service diagnosis, and the fact that services are used in different contexts. Yan, et al. [5], [6], 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. [18], [19] describe approaches for diagnosing quality-of-service problems in service-oriented architectures. Mayer and colleagues [20], [21], describe a similar diagnosis approach that is based on analyzing execution traces of failed transactions.

Wong et al. [22] 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 [23]. This is also proposed by [17], and we see it as a standard approach to determine which service takes part in which system transaction. Chatzigiannakis and Papavassiliou [24] use principle component analysis in order to identify faulty nodes in sensor networks.

Heward et al. in [25] describe an algorithm for optimization of monitoring configurations for web services. They use their optimization algorithm in order to reduce the monitoring overhead in a service-based system, something that would also benefit our proposed techniques.

VIII. CONCLUSION AND FUTURE WORK

The goal of this paper is to investigate to which extent an increase in monitoring granularity supports the diagnosis of faulty services. Referring to our research questions, we looked at:

RQ1: How the observation granularity affects 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 observation 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 the granularity may be determined. In other words, what are good locations for monitoring? 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. The tool we use, *EMMA*, is able to do this. However, we restrict the monitoring to the code block level, representing unique execution branches through a service or proper isolation of tight coupling. In the case study, this was done manually, but the effort resembles code slicing techniques. In future work, we will assess

[12]http://redis.io

to which extent such techniques may be used in order to automate the placement of monitors.

RQ3: To which extent can incorrect diagnoses be improved, and what is the overhead incurred? Our case study demonstrates that we are able to diagnose all faulty services correctly through increasing the monitoring granularity. Yet, at the same time, we are also worried about the performance overhead that the entire infrastructure adds. In particular, we noted an overhead of around 700% for adding monitors to our services-framework. We believe, this is attributable to the inefficient prototypical implementation of our monitoring framework in Turmeric. Specifically, we use synchronous invocations in order to store the monitoring data for later analysis. In the future, we will change this implementation into a publish-subscribe architectural style and use asynchronous data logging. Adding finer-grained monitors through code instrumentation inside the services did not create any additionally measurable overhead.

ACKNOWLEDGMENT

We would like to acknowledge NWO for sponsoring this research through the Jacquard ScaleItUp project. Also many thanks to our industrial partners Adyen and Exact.

REFERENCES

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

[2] G. Canfora and M. Di Penta, "Testing services and service-centric systems: challenges and opportunities," *IT Professional*, vol. 8, no. 2, pp. 10 –17, march-april 2006.

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

[4] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, vol. 15, no. 3-4, pp. 313–341, 2008.

[5] Y. Yan and P. Dague, "Modeling and diagnosing orchestratedweb service processes," in *Proc. Int'l Conf on Web Services (ICWS)*. IEEE, 2007, pp. 51–59.

[6] Y. Yan, P. Dague, Y. Pencole, and M.-O. Cordier, "A model-based approach for diagnosing fault in web service processes," *International Journal of Web Services Research (IJWSR)*, vol. 6, no. 1, 2009.

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

[8] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

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

[10] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. van Gemund, "Prioritizing tests for software fault localization," in *Int'l Conf. on Quality Software*. IEEE, 2010, pp. 42–51.

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

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

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

[14] C. Chen, A. Zaidman, and H.-G. Gross, "A framework-based runtime monitoring approach for service-oriented software systems," in *Int'l Workshop on Quality Assurance for Service-Based Applications (QASBA)*. ACM, 2011, pp. 17–20.

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

[16] M. Weiser, "Program slicing," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 1981, pp. 439–449.

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

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

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

[20] W. Mayer, G. Friedrich, and M. Stumptner, "Diagnosis of service failures by trace analysis with partial knowledge," in *Service-Oriented Computing*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6470, pp. 334–349.

[21] ——, "On computing correct processes and repairs using partial behavioral models," in *20th European Conference on Artificial Intelligence (ECAI)*, 2012, pp. 582–587.

[22] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, vol. 83, no. 2, pp. 188–208, 2010.

[23] I. Grosclaude, "Model-based monitoring of component-based software systems," in *Int'l Workshop on Principles of Diagnosis*, 2004, pp. 155–160.

[24] V. Chatzigiannakis and S. Papavassiliou, "Diagnosing anomalies and identifying faulty nodes in sensor networks," *Sensors Journal, IEEE*, vol. 7, no. 5, pp. 637 –645, May 2007.

[25] G. Heward, J. Han, J.-G. Schneider, and S. Versteeg, "Runtime management and optimization of web service monitoring systems," in *Proc. Int'l Conf on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2011, pp. 1–6.