Delft University of Technology
Software Engineering Research Group
Technical Report Series

Spectrum-based Fault Diagnosis for Service-Oriented Software Systems

Cuiting Chen, Hans-Gerhard Gross, Andy Zaidman

Report TUD-SERG-2012-023

TUD-SERG-2012-023

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note:

Spectrum-based Fault Diagnosis for Service-Oriented Software Systems

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

This paper presents a lightweight, fully automated, spectrum-based diagnosis technique for service-oriented software systems that is combined with a framework-based online monitor. An experiment with a case system is set up to validate the feasibility of pinpointing problematic service operations. The results indicate that this approach is able to identify problematic service operations correctly in 72% of the cases.

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

I. Introduction

Service-oriented software systems offer many benefits in realizing flexible, interoperable, and adaptable distributed ICT infrastructures. These benefits are mainly attributable to the loose coupling of services, facilitated through modern underlying communication platforms, and their natural disposition to dynamic deployment, reconfiguration, and evolution. However, this nature of service-oriented system also presents many challenges [1], particularly concerning quality assurance. The fact that service-based applications only fully materialize when deployed in production, i.e., ultra-late binding [2], renders many of the traditional (offline) quality assurance methods useless [3]. In particular, many failures only emerge during operation time, triggered through runtime re-configuration or re-deployment of services [3], or resulting from incompatibilities in service versioning [4].

Although, by their very nature, service-oriented systems provide all the ingredients necessary to recover from and adapt to operation time failures [5], there is no standard means in those systems to detect and diagnose emerging problems automatically, and make propositions as to *what to recover* and *where to adapt?* The fact that a problem is detected in a particular service does not necessarily mean that this service is corrupt and should be exchanged. Faults located in other services may propagate through the system and cause an otherwise healthy service to break [6].

Automated software fault diagnosis can be applied to service-oriented systems in order to trace a detected problem back to the service where it originated. Fault diagnosis refers to the detection of a failure, i.e., a discrepancy between expected and observed behaviour, plus the localization of its root cause, i.e., the fault that caused an erroneous system state [7]. Being able to perform automated fault diagnosis in an operational service system with minimal performance impact demands a fault localization technique with ultra-low computational overhead such as spectrum-based fault localization (SFL) [8], and inbuilt monitoring approaches for detecting failures.

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

RQ1 How can a failure be detected in an operational service-oriented system? This is concerned with extracting relevant information from a running service-oriented system for initiating diagnosis.

RQ2 How can SFL be applied in a service-oriented system in order to trace a failure back to its respective root cause? This focuses on identifying and providing the right input for SFL in a service-oriented system.

RQ3 How well does SFL perform in a service-oriented system in terms of correctness of the diagnosis?

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

The article is organized as follows: Sect. II presents the research field and techniques related to our approach. Sect. III focuses on the concepts and implementation of SFL for service-oriented systems. Sect. IV describes the case system and the setup of the experiment. Sect. V discusses the experimental results and the limitations of the approach. Related work is presented in Sect. VI. Finally, Sect. VII concludes this paper.

II. Background

A. Quality Assurance for Services

Quality assurance for services is difficult, specifically, for checking their interactions and integration [9][10]. This is mainly attributable to their loose coupling, late (runtime) binding, and deployment in many application contexts. Runtime integration testing [1][11] can alleviate the detection of failures in dynamic systems. However, it requires specific architectures that support it, and eliminate side effects, for example, interference with normal system operation. Runtime integration testing also requires test suites that must be built and maintained. In addition, it only detects a problem, but does not identify it.

B. Online Monitoring vs. Online Testing

Instead of performing proactive online testing, we favor passive online monitoring plus online diagnosis for identifying residual defects. Monitoring is less intrusive, requires fewer assumptions about the system, and is easier to incorporate into a service-oriented system. Many modern service platforms such as Apache's Axis2[1], Redhat's JBoss[2], or Ebay's Turmeric[3] come equipped with extensive monitoring/profiling frameworks that can be adapted to diagnosis. Our case system makes heavy use of Turmeric's monitoring functionality. The disadvantage of monitoring: errors hiding in seldom used parts of a service-oriented system cannot be triggered on purpose, and are unlikely to be identified. However, this is not an urgent issue, since only those services or parts thereof which are actually used in a particular application, will be exercised and monitored.

C. Spectrum-based Fault Localization

SFL is a statistics-based technique that automatically infers a diagnosis from symptoms. The *diagnosis* is a ranking of potentially faulty components (block, source code line, etc.) in a system, with the most likely faulty one ranked top. The *symptoms* are observations about component involvement in a system execution, plus pass/fail information about that execution [12]. Component involvement is expressed in terms of *block-hit-spectra* (hence its name), producing for each execution a binary coverage value per component [13][14]. Further, each system execution (test), is associated with a binary verdict (pass=0, fail=1) from an oracle. Several tests lead to an activity matrix, representing coverage of each component over time. The binary verdicts lead to an *output vector*. The diagnosis is calculated through applying a similarity coefficient (SC) to each component activation vector and the outcome vector. The similarity denotes the likelihood of a component being the faulty one,

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	

and, therefore, determines its position in the ranking. Any SC may be used; however, the Ochiai SC has been found to work best [15]. This technique mimics how a human diagnostician would infer a diagnosis from observing which parts of a system were involved in producing a failure.

SFL is illustrated in Table I by means of a Ruby program. This example is comprised of components $C_0 - C_{10}$ with a source code line as component granularity. It is exercised with six tests/transactions, leading to the component activation for each transaction $t_1 - t_6$ noted down in the activity matrix. Four transactions have failing test outcomes (1); two have passing test outcomes (0), noted in the output vector. The Ochiai SC is calculated for the output vector and each component activation vector. Then, the similarity values are brought in a descending order. This results in component 3 being ranked top with 100% likelihood, which represents the location of the fault in this example system.

III. SFL for service-oriented systems

A. Concepts of SFL for service-oriented system

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

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

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

In service-oriented systems, this is not the case. Because a service instance serves many applications, it will not be activated exclusively from within one application context,

[1]http://axis.apache.org
[2]http://www.redhat.com/products/jbossenterprisemiddleware/
[3]https://www.ebayopensource.org/index.php/Turmeric

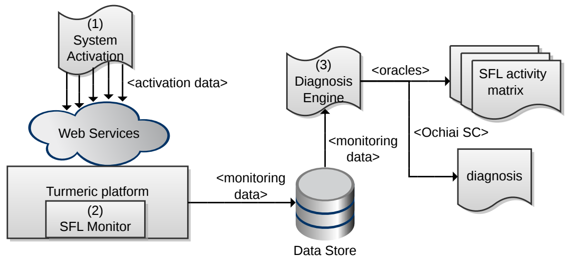

Figure 1. Monitoring and diagnosis architecture

but from a potentially arbitrary number of other applications in other contexts. To apply SFL in a service-oriented system, a system execution needs to be made explicit through introduction of a unique transaction ID. This allows a clear separation of system executions in the activity matrix of SFL.

3) Component involvement: In the basic SFL approach, component involvement is measured through coverage tools. However, in service-oriented systems, coverage is a delicate issue. Because of its inherent distributed nature, there is no single controlling authority that is able to produce service coverage information, by overseeing all service invocations and associating them with the different application contexts in which a service is used. This can only be done by the services themselves, or an underlying service framework. Applying SFL in service-oriented system requires dedicated monitors that observe the service communication and associate the services/operations with their corresponding transaction IDs.

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

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

B. Implementation of SFL for service-oriented systems

The first step in applying SFL to a service-oriented system requires online monitoring to obtain information about each user transaction with the system. The second step involves the construction of a diagnosis engine that maintains the SFL activity matrix, and calculates the diagnosis. Third, component granularity is set to the service operation, because it permits a more fine-grained diagnosis. The SFL implementation for our case study is summarized in Fig. 1 and explained in the following sub-sections.

1) System Activation: Typically, a system is invoked through its user interface. However, in our case, user interaction is automated in order to evaluate our approach. We use SoapUI[4] to create XML templates of SOAP messages which are required for calling the services. Then, the templates are passed to JMeter[5] in order to generate multiple user requests that are exercised automatically.

2) Online Monitoring: Online monitoring follows a framework-based approach [16], realized in Turmeric[6], eBay's open source service framework. Turmeric offers many inbuilt features supporting the implementation of online monitoring required in our approach, and it confines the necessary amendments for online SFL to the absolute minimum, yielding a slender implementation.

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

Another monitoring requirement is the observation of exceptional behavior in the service-oriented system. This is used as oracle by the diagnosis engine (explained later in Sect. III-B3), but it is also realized in the four handlers already introduced. All services in our case system are designed to log their occurring exceptions in a data store. The handlers constantly monitor the data store for new exceptions. Once an exception is detected, it will be associated with the correct transaction through the transaction ID in the data store.

In summary, we use the following monitoring data:

- *Transaction ID*: Turmeric generates a unique ID to associate messages involved in the same transaction.
- *Service and operation name:* the name of a component in the diagnosis is made up of the service name plus the operation name.

[4]http://www.soapui.org
[5]http://jmeter.apache.org
[6]https://www.ebayopensource.org/index.php/Turmeric/

- *Message body:* the content of the message can be checked for failures.
- *Exception:* indicates that a transaction threw an exception.
- *Pipeline:* information about which pipeline handled the message; this distinguishes between requests and responses in provided and required interfaces.

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

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

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

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

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

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

IV. Experimental Setup

A. Case System

We devised a case study based on Spicy Stonehenge[7] [17] to demonstrate how online SFL can be applied in service oriented architectures, and validate to which extent SFL helps to pinpoint problematic service operations. Spicy Stonehenge is a service-based system simulating the stock market. It was deployed on top of the Turmeric platform.

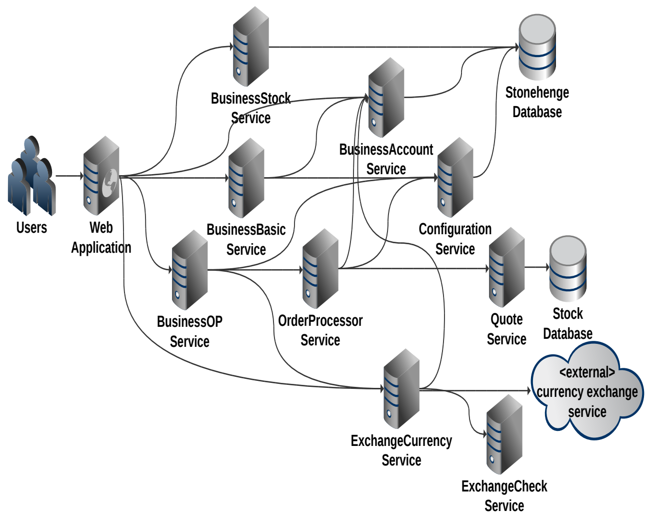

Figure 2. Architecture of the extended Spicy Stonehenge

We extended the original Spicy Stonehenge system[8], because we required more complex service interactions, so that an error seeded in one service can propagate to other services and make them fail. That way, we could assess to what extent our diagnosis approach is able to pinpoint the real erroneous service, instead of the failing, but healthy one.

The extension of Spicy Stonehenge concerned adding new functionality, so that users could buy and sell stocks in different currencies. When a user subscribes to shares in a foreign currency but intends to buy in domestic currency, the system will automatically calculate and apply the correct exchange rate. This extension introduces more complicated transactions in the system and makes it more realistic. In addition, some service operations are modified in order to impose more and more interesting service interactions. Moreover, we split some services that contain large sets of operations into smaller ones, which makes the interactions more complex, and fault injection easier (see Section V).

Figure 2 illustrates the architecture for the extended Spicy Stonehenge which is comprised of 10 web services including one *external currency exchange service*, plus a *web application* for user interaction. *BusinessBasicService* and *BusinessAccountService* provide the functions for user authentication, login, and the user account. *BusinessOPService* and *BusinessStockService* are used for buying and selling stock, and checking orders and market summaries. *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/spicy-stonehenge

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

In the following, we show typical system transactions that can be performed with our extended version of Spicy Stonehenge.

```
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
```

B. Conducting the experiment

We created 160 faulty versions of our case system outlined in Fig. 2, by applying the PIT mutation tool[9]. For each faulty version, we applied JMeter to execute 48 web service requests consecutively to cover all service operations. Upon completion of all transactions for one faulty system version, the diagnosis engine was invoked to parse the monitoring data, identify the failures in the system, and create an activity matrix with an output vector. Then, it was assessed whether the resulting diagnosis correctly pinpoints the faulty service operation. The whole experiment was designed for the single fault case, i.e. we ensured that each version of the system contains only one fault.

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

There are many mutation tools available such as μJava, Jumble, Javalanche. However, we chose PIT, because it mutates the byte-code, rather than the source code. This represents a quick way to mutate code, and it is also safe, i.e., generation of invalid programs is avoided. Moreover, PIT provides extensive documentation, a wide range of useful mutators (i.e., mutation operations)[10], and a comprehensive reporting function. Its only drawback comes from the fact that it maintains its mutated classes in memory, so that we have to extend it with the ability to save the mutants as files.

[9]http://pitest.org/
[10]http://pitest.org/quickstart/mutators/

Table II
ACTIVE MUTATORS IN THE EXPERIMENT

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

Only the service implementation classes are mutated, neither platform code, nor library code. The mutation operations applied to a subject depend on what PIT finds in the service's implementation logic. Several mutators may be applied per implementation class, of which we choose one for generating one fault in the system. This is due to the single fault scenario, and it explains the high number of faulty system versions. For every version of the system, we replace the original class with its respective mutated one in the service's *.war-file*, and execute the system. All nine internal services shown in Fig. 2 are mutated that way.

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

V. RESULTS AND DISCUSSION

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

A. Experimental Results

A diagnosis refers to a component ranking according to the similarity coefficients. If the diagnosis is not accurate, it might well rank healthy services before the faulty one. Accuracy of a diagnosis can be measured through residual diagnosis cost [18], i.e., the cost of unnecessarily treating healthy services before arriving at the real faulty one.

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

Table III summarizes the diagnosis results for our extended version of Spicy Stonehenge. For each service, the table indicates the type of the mutation operations used (IDs displayed in Table II), the total number of mutations performed, the correctly and incorrectly performed diagnoses, and the percentage of correct diagnoses.

In total, 115 out of the 160 faulty system versions are diagnosed correctly, yielding a 72% success rate for our experiment. Faulty versions of five out of the nine services used in our system can always correctly be diagnosed by SFL. However, the mutants of four services cannot be diagnosed so successfully. A careful analysis of these cases presents a number of issues to be discussed in the following.

Reasons for Incorrect Diagnoses: We can identify two significant reasons for why diagnoses are incorrect (summarized in Table IV):

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

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

Table III
EXPERIMENTAL RESULTS

Services	Applied Mutators	# of Mut.	Diagnosis Correct	Diagnosis Incor.	Correct Diagn.
BusinessAccountService	2,4	7	7	0	100%
BusinessBasicService	1,2,4	27	23	4	85%
BusinessOPService	1-4	19	14	5	75%
BusinessStockService	2	8	8	0	100%
ConfigurationService	2	9	9	0	100%
ExchangeCheckService	1-3	8	8	0	100%
ExchangeCurrencyService	1,2,4	24	3	21	13%
OrderProcessorService	1-5	41	26	15	63%
QuoteService	1-4,6	17	17	0	100%

Table IV
REASONS FOR INCORRECT DIAGNOSES

Services	Incorrect Diagnoses	No Activation	Tight Interaction on Failure
BusinessBasicService	4	2	2
BusinessOPService	5	1	4
ExchangeCurrencyService	21	2	19
OrderProcessorService	15	0	15

Multiple Faults: Initially, we stated that we are only interested in the single fault case, and the Ochiai similarity coefficient represents a single-fault approach. However, in our example system, we observe that one mutation in a service implementation may affect more than one service operations. Since the granularity is at the service operation-level, rather than at the service-level, we actually introduce multiple faults into our system. This problem is attributable to a mismatch between the granularity of the fault injection and the granularity of the diagnosis. SFL always ranks one of the faulty service operations at the top (but not all of them), meaning it finds the fault. Which of the several faulty service operations will be ranked top, depends on its number of activation. According to our definition of correctness, we treat this result as a correct diagnosis.

B. Discussion and Lessons Learned

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

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

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

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

Although, in this work, we specifically target the single fault case for demonstration purposes, we acknowledge the fact that this is not realistic. The Ochiai coefficient is limited to the single fault. Even though Ochiai identifies the root cause of the failure, it cannot pinpoint all operations involved in exhibiting it. In this case Ochiai fails to convict all faulty operations, which is to be expected. Heavy-weight, bayesian- and model-based diagnosis approaches [19] do work in the multiple fault case. However, it remains to be evaluated in future work how such techniques can be applied online and in the context of service-oriented architectures.

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

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

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

VI. Related work

Chen et al. present *Pinpoint* [20], a tool based on similarity coefficients. However, they do not address the problems of inter-service diagnosis (that services are used in different contexts), and use a weaker similarity coefficient. Zhang et al. [21] propose a hybrid approach, combining a matrix- [22] and a Bayesian-based probabilistic diagnosis method for SOA systems. Since the dependency matrix is generated before operation, the diagnosis cannot adapt well to the dynamic nature of SOA. Even though, the authors considered various ways to reduce the computational complexity, bayesian approaches are still heavyweight compared to spectrum-based approaches. Mayer et al. [23] diagnose faults in business processes for SOA systems. Their approach requires partial information of process executions by reasoning about possible activities in system behavior. However, the models for diagnosis are rather complex, and proper evaluation is still pending.

Grosclaude describes a model-based monitoring approach for component-based systems, and suggests to use transactions IDs in order to associate messages sent between components [24]. This is also proposed by [20], and we see it as a standard approach to determine which service takes part in which system transaction. Although slightly less related, Zhang et al. [22] present a framework for diagnosing QoS problems in SOA through monitoring service states. Another interesting approach is introduced by Heward et al. [25], in which they propose an algorithm for optimization of monitoring configurations for web services. They use an optimization algorithm in order to reduce the monitoring overhead in a service-based system, something that would also benefit our proposed techniques.

VII. Conclusion and Future work

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

RQ1: How a failure can be detected in an operational service-oriented system: We enabled framework-based monitoring, reusing the tools of an existing service platform, i.e. Turmeric, for transaction tracing. In addition, we devised a three-phased oracle using the monitoring in order to associate failure information with the transaction traces. Both monitor and oracle generate component involvement and pass/fail information required in fault diagnosis.

RQ2: How SFL can be applied in a service-oriented system: The fault localization technique is implemented in a dedicated (external) diagnosis engine for efficiency. This accesses the information generated by the monitor and the oracle and turns that into an activity matrix and an output vector, and then, calculates the diagnosis.

RQ3: How well SFL can identify faults seeded into service implementations: The results confirm the feasibility of the approach, and indicate a high success rate of the diagnoses, i.e., 72% correctness. The fraction of incorrect diagnoses can be explained after careful analysis, which results in a number of feasible directions for future work.

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

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

[2] 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.

[3] 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.

[4] M. Papazoglou, "The challenges of service evolution," in *Advanced Information Systems Engineering*, ser. LNCS. Springer, 2008, vol. 5074, pp. 1–15.

[5] 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.

[6] 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.

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

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

[9] A. Bertolino and A. Polini, "Soa test governance: Enabling service integration testing across organization and technology borders," in *Proc. Int'l Conf. on Software Testing, Verification, and Validation Workshops (ICST)*. IEEE, 2009, pp. 277–286.

[10] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," in *Software Engineering*, ser. LNCS. Springer, 2009, vol. 5413, pp. 78–105.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-based multiple fault localization," in *Proc. Int'l Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[20] 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.

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

[22] 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.

[23] W. Mayer, G. Friedrich, and M. Stumptner, "Diagnosis of service failures by trace analysis with partial knowledge," in *ICSOC*, 2010, pp. 334–349.

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

[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.

TUD-SERG-2012-023
ISSN 1872-5392