2012 19th Working Conference on Reverse Engineering

Understanding the Runtime Topology of Service-Oriented Systems

Tiago Espinha, Andy Zaidman, Hans-Gerhard Gross
Delft University of Technology, The Netherlands
{t.a.espinha, a.e.zaidman, h.g.gross}@tudelft.nl

***Abstract*—Through their dynamic and loosely coupled nature, service-oriented systems are ideal for realizing runtime evolvable systems. However, runtime evolution demands proper understanding of the configuration of service-oriented software systems over time. In order to keep system disturbance as low as possible while replacing existing services with their newer versions, engineers require an adequate illustration of how the connections and dependencies of services change in the running system. That way, they can identify the most appropriate point in the service-oriented systems' operation time for maintenance.**

In this paper, we make use of the runtime topology (i.e. the configuration of a distributed, dynamically composable software system which describes available services and how they depend and interact with each other) to help software engineers understand service-oriented systems, and we describe a tool, *Serviz*, that visualizes how services are activated, and how much they interact over time. A user study demonstrates to which extent the runtime topology can support the analysis and understanding of service-oriented software systems.

I. Introduction

The fact that software systems need to evolve in order to remain successful has been long recognized [1]. With the advent of Service Oriented Architectures, the maintenance problem was said to become easier [2], as a Service Oriented Architecture (SOA) would be composed of several loosely coupled (and smaller) services. These smaller services are said to be easier to evolve and more flexible to compose, thus easing overall evolution. However, in 2004 already, Gold et al. warned that even though the actual evolution was possibly easier, the understanding of service-oriented software systems would still be a challenge [2]. This is mainly due to the shift from understanding a monolithic application to understanding a distributed system composed of many entities (services).

Thus, program comprehension remains an absolute and important prerequisite to evolving systems [3], also for systems built around the concept of service-orientation. In this context Corbi even reported that up to 60% of the time effort of a maintenance task lies in understanding the system at hand [4]. Strangely enough then, a large-scale survey of scientific literature in the area of program comprehension using dynamic analysis from 2009 did not reveal any major advances in the area of understanding service-oriented software systems using dynamic analysis. As the authors note, this seems strange as dynamically orchestrated compositions of services would seem to benefit from dynamic analysis for understanding them [5].

In particular, when understanding a service-oriented software system, one of the challenges lies in the uncertainty of the software construction [2]. This uncertainty is instigated by the fact that service-oriented software systems are composed of loosely coupled components or services. As the composition of the entire service-oriented system only happens at runtime, the services composing the system may only be known at execution time [6]. Furthermore, the service-composition can be changed at runtime, a situation that can become even more complicated when services can be automatically discovered [6], [7]. It comes as no surprise then that Kajko-Mattsson et al. identified the understanding of the service-oriented infrastructure as one of the most important challenges when evolving service-oriented software systems [8]. In order to support the understanding task, we propose to reverse engineer [9] the *runtime topology* of the service-based system. The runtime topology is the configuration of a distributed, dynamically composable software system and describes which services are available and how they depend and interact with each other at any point in time. Knowing exactly *which* services depend on each other, and *when* they are interacting with each other, enables a better understanding of how the service-oriented system is composed *and* a more efficient maintenance strategy as insight is obtained in when an upgrade of a service should ideally occur. However, this knowledge is only possible by continuously analyzing the runtime topology. That is, not only looking at the current runtime topology, but analyzing how it evolved over time. As such, we consider the time-dimension of this runtime topology to be very important.

In this paper, we demonstrate how the runtime topology of a service-oriented software system can be reverse engineered from observing its operations, and investigate how the topology, augmented with a time-dimension, enables a better understanding. More specifically, our research is steered by the following research questions:

- **RQ1** Does the runtime topology help in identifying services involved in a specific use case?
- **RQ2** Does the runtime topology help in debugging a service-oriented system?
- **RQ3** Does the runtime topology augmented with the time-dimension help to identify services that are used more often (e.g., to optimize them)?
- **RQ4** Does the runtime topology help to identify periods in time when the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?

Our proposed runtime topology reverse engineering approach is implemented in a tool called *Serviz*. In order to evaluate the approach, we set up a user study that involves Spicy Stonehenge [10], a stock-market simulator, as subject system. The user study is conducted as a one-group pre-test post-test experiment and involves 8 participants.

The remainder of this paper is structured as follows: Section II describes our approach (the data requirements, how we collect the data and how it is presented). Section III outlines the experiment we performed and Section IV presents the results of our experiment. In Section V-B we reason about possible threats to validity and we finish with related work and conclusion in Sections VI and VII.

II. Approach

Central to our understanding approach stands the runtime topology of a service-oriented software system, representing the configuration of a set of services that are deployed within an environment. Before developing our approach, we compiled a number of maintenance scenarios representative for service-oriented systems. They are described in Table I. Further down in Subsection II-A, we present the data requirements inherent to creating a runtime topology of a service-oriented system that supports the maintenance tasks described.

A. Required data

In order to create a runtime topology, a significant amount of data must be collected from the service-oriented system. We must first look at exactly which data we require to be able to build a complete picture of the service-oriented system, and

Scenario 1: Which services are available?
Motivation: Since services are loosely coupled, they are ideal to be reused in different service compositions realizing various business goals. While working on new business functionality, a maintainer requires to know which services are already deployed, and which of them can be reused in the service composite for realizing a new business case.
Scenario 2: Which services are actually used?
Motivation: An important maintenance requirement is the knowledge about the used and disused services in a service-oriented system. When replacing services or adding new versions it is important to know whether these services are still being used. Otherwise, engineers could be attempting to treat the wrong (disused) versions of system components. Knowledge about used/disused services also represents the first step in identifying dead services that should be removed.
Scenario 3: When are certain services least used?
Motivation: On top of knowing which services there are and which are used, it is also important to know when they are used, and to which extent. If services must be updated or their new versions installed, the knowledge of their usage patterns, based on historical data, helps to determine when system updates may be conducted with the least impact on the users of the service-oriented system.
Scenario 4: How do services interact for realizing a particular use case?
Motivation 1: In a large system, composed of many services, an important maintenance task is the deployment of new versions of services. An essential requirement, therefore, is the knowlegde about which services are normally required in a particular use case, so that dependent services can be configured to use the new version. **Motivation 2**: When a particular feature offered by the service-oriented system needs to be changed, it is important to understand which other services are involved in the provision of this feature, and thus which individual services, possibly, must be updated or reconfigured.

TABLE I
Motivating scenarios

then look at how it must be extracted and used. The runtime topology can contain different amounts of data, depending on the goal at hand. We focus, however, on the data required to satisfy our maintenance scenarios.

First, we present the different data requirements for the different scenarios outlined in Table I. We then proceed to describe how the data can be retrieved and presented.

a) Static data: Data collected at runtime provides great insight into which services are dependent on each other. This dynamic technique, however, is unable to provide a full picture of the services deployed in the system, therefore making it impossible to satisfy Scenario 1 based on this data alone. Also, in order to be able to detect unused services and satisfy Scenario 2, we can not rely on a technique which requires the code to be executed. For this reason, we also require static snapshots of what the system looks like in terms of which services are deployed in the platform. These snapshots are collected whenever a change occurs in the system. By storing the latest static configuration and verifying which services present in the static configuration have not recently stored dynamic data, we can infer which services have become disused.

b) Historical usage: The historical usage data is required to determine the periods of time when services have been used the least in the past (Scenario 3). This requirement is easily satisfied by adding a timestamp to each invocation pair, thus adding information about when each invocation occurred.

c) Causal relationship: In order to enable Scenario 4 we require the causal relationships between services to be stored. To satisfy this scenario, different levels of granularity can be provided. For example, if a web service A invokes a web service B, the event describing this invocation relationship can be stored as a pair relating web service A to web service B. However, this data is too coarse. Narrowing the scope down to the method level provides system maintainers with a more focused aid as to which method is involved in a specific invocation pair thus making it easy to identify “use case flows”.

These invocation pairs on their own still fall short of fully fulfilling Scenario 4. This data tells us that web service A invokes web service B and that web service B invokes web service C. It does not, however, tell us whether these two invocations are related or if they were triggered independently. This makes it impossible to trace a single use case to all the pairs involved in providing its functionality.

In sum, to satisfy the requirements of inferring the causal relationship between web services and to tie it to use cases in a way which is useful to system maintainers, we are required to store the invocation pairs at method level and a common request identifier which binds all the pairs to a single request trace.

B. Data extraction

The data extraction step required to enable the runtime topology is highly dependant on the platform being used. Some service platforms may already provide parts of the required data whereas others may require deeper changes in order to obtain such data. Ultimately, however, this is a completely automated step after it has been enabled for a specific platform. This means the data is automatically pulled from the running system rather than requiring constant manual intervention.

Next we present the dynamic and static data we use, as well as the steps involved concerning its extraction.

Dynamic data. Different web services platforms provide different means on what concerns data extraction. For our approach we chose the Turmeric SOA[1] platform, the open source version of eBay's services platform, as it already provides some of the information we described in the previous subsection. This data is not provided by default in Turmeric SOA and in order to collect it, we make use of one of the framework's features. This framework allows us to intercept each incoming request by means of a Java class which can then access and manipulate information regarding the request. Since this data is only available during the web service's invocation lifetime, the data must be stored persistently. For this reason, the request handler stores the data in an Apache Derby database in order to keep a historical track of the system's usage.

In practice, we are able to collect all the required data by handling the incoming requests from the point of view of each web service. This event provides us with all the information mentioned in Subsection II-A. On the event of an incoming request, the following data is stored:

- **Request ID** - A unique identifier per invocation trace. This ID allows us to link pairs of services as belonging to the same trace.
- **Timestamp** - The current timestamp at the time of the request, as seen by the database server. This timestamp is a crucial piece of data as it allows us to analyze exactly when the web services were busiest.
- **Consumer name and method** - The name of the caller service (i.e. client) and the respective method which caused this request pair.
- **Service name and method** - The name of the callee service in this request pair and the respective web method being called.

This data is readily available in string format through Turmeric SOA's handlers, except for data about the methods involved in the invocation pairs. This must be added manually by appending it to the existing request string.

Another challenge comes from how different instances of the handler are presented with different data. When a web service A is invoked, the request string available on this service does not yet contain the data identifying a possible AB invocation pair. We must then collect the data provided by all the instances of the request handler (i.e. also collect the request string from web service B, C, etc.) and then decide, based on comparing the request ID, which data is the most complete. This is stored in the database and then queried by our visualization tool, Serviz.

Static data. In order to store the static information we created a small script which periodically inspects the application server's folder to check whether services have been added or removed; this is possible by comparing the current set of services with the latest stored information in the database. If the deployed services change (a service is removed or added), this event is stored, along with the new configuration.

C. Data presentation

The data described in the previous steps is meaningless unless transformed into information which can be consumed by system maintainers. In order to satisfy the motivating scenarios presented before and given the size of modern software systems, we also developed a visualization tool to present this data to maintainers. In particular, we create a dynamically updated time-based dependency graph, thus creating a runtime topology of the system.

D. Serviz

Serviz is our main contribution putting into practice the implementation of the two aforementioned steps. It encompasses both a data collection component and the visualization component. Data collection relies on the Turmeric SOA platform, while the visualization is web-based and makes use of open-source Javascript visualization and graphing libraries. Serviz is an open-source tool and its user-interface allows system maintainers to visualize and inspect the runtime data collected from web-service based systems. All the code, detailed instructions on how to use Serviz and quick-start binaries are available on GitHub[2].

By using Serviz, maintainers are presented with a topology of a running service-oriented system[3], and are therefore able to analyze the system's usage over user-defined periods of time. This way it becomes easier to identify possibly affected services whenever maintenance is performed in a specific deployment of a certain web service. Similarly, by introducing the time dimension to the runtime topology and allowing maintainers to scan the web service usage over past periods of time, we are able to estimate which periods of time are less busy and thus more suitable to perform maintenance.

III. Experimental Setup

In the experiment we performed, we assessed to which extent Serviz can help the maintenance tasks of distributed software systems. In particular, we gauged how useful, adequate and effective Serviz is throughout a series of maintenance tasks performed on a web-service-based system. The experiment is organized as a one-group pretest-posttest pre-experimental user study design [11]. The subject system that we used for doing the maintenance tasks on is *Spicy Stonehenge*, a simulation of the stock market [10].

[1]Turmeric SOA — https://www.ebayopensource.org/index.php/Turmeric/

[2]Serviz — http://git.io/serviz

[3]Screenshot — http://goo.gl/SPzH1

This type of experiment is called pre-experimental, to indicate that it does not meet the scientific standards of experimental design [12], yet it allows us to report on facts of real user-behaviour, even those observed in under-controlled, limited-sample experiences. In particular, the pretest-posttest design does not allow to identify an event related to the dependent variable that intervenes between the pretest and the posttest where the effects could be confused with those of the independent variable [12].

A. One-group pretest-posttest

In this type of experiment, there is no control group. Instead, every participant is required to fill out a questionnaire before the assignment takes place. This questionnaire helps understanding the participant's opinions and expectations before the tool is used. After the participant uses the tool, a posttest questionnaire brings back some of the same questions from the pretest in order to compare whether the tool has in fact been useful.

For both questionnaires, close-end matrix questions are used where the participants can rate each question on a 1 to 5 scale, based on whether they strongly disagree up to strongly agree with the question (Likert scale).

Pretest design. The pretest questionnaire (Table II) is divided into five main sections.[4] Each of these sections helps us filter out conditioning factors that might influence the user's experience with Serviz. These factors are divided into:

1) Experience with software development: has the participant used the underlying technologies supporting the case study system? (Java, Eclipse, Maven, web services, etc)
2) Opinion on software maintenance: did the participant perform maintenance tasks before? Is he/she familiar with such tasks on software written by third parties?
3) Opinion on distributed software: is the participant familiar with distributed software? Does he/she think distributed software is more difficult to maintain than its monolithic counterpart?
4) Views on dynamic analysis: has the participant used dynamic analysis techniques to perform maintenance before? Does he/she understand the added value of such techniques?
5) Expectation regarding Serviz: after reading a short description of Serviz, what are the participant's expectations from Serviz?

i-a	I am an experienced Java developer
i-b	I often use Eclipse to write Java code
i-c	I am familiar with the Maven build process
i-d	I am familiar with the concepts behind web services
i-e	I have developed web services before
i-f	I am familiar with service-oriented architectures
ii-a	I have performed software maintenance before
ii-b	I am often involved in software maintenance tasks
ii-c	I mostly maintain software developed by other people
ii-d	I have used software visualization tools before
ii-e	I have maintained a distributed system before
ii-f	For this task I used static analysis
ii-g	For this task I used dynamic analysis
iii-a	Software maintenance in general is a difficult task
iii-b	Maintaining distributed software is more difficult than monolithic software
iii-c	I mostly maintain software developed by other people
iv-a	During software maintenance, I often use static analysis
iv-b	During software maintenance, I often use dynamic analysis
iv-c	I have performed dynamic analysis on a software system before
iv-d	Dynamic analysis provides an added value to static techniques
iv-e	I have used dynamic analysis tools on distributed systems before
Tool description: Serviz is a tool that provides you with a topology of the system created based on both dynamic and static information. With this tool, you can see which services are being and have been used throughout time. You can also see whenever services have not been used and which services are used the most.	
v-a	By displaying runtime artifact relationships, Serviz will help me identify artifacts involved in a specific use case
v-b	By displaying runtime artifact relationships, Serviz will help me debug a distributed system
v-c	With the dimension of time, Serviz can help me identify services that are used more often
v-d	Serviz will stand in the way of maintenance of distributed systems instead of aiding it

TABLE II
PRETEST QUESTIONNAIRE

Posttest design. After the completion of the pretest and the assignment, participants are asked to fill in a posttest questionnaire (also composed of close-end matrix questions based on the Likert scale). Its goal is to understand how useful the participants found Serviz and whether it met their initial expectations.

The posttest (Table IV) is divided into five categories. We ask the participants to identify the following aspects of their experience with Serviz:

1) Overall experience: participants are asked to report on their overall experience with the assignment.
2) Usability: was Serviz clear and intuitive for the participants?
3) Usefulness: how did Serviz help in identifying different runtime aspects of the running system? In other words, how useful was it in aiding the task of system maintenance?
4) Favorite feature: in this question, participants are asked to rank their favorite Serviz features.
5) Further remarks: the last question is an open-end question where we allow participants to provide more detailed feedback about their experience and possible suggestions to improve Serviz.

B. Assignment

In order to evaluate Serviz we designed an assignment which aims at simulating common maintenance tasks. As an attempt to align our research with realistic maintenance tasks, we refer to the nine software comprehension activities as described by Pacione et al. [13].

[4]Both the pretest and posttest questionnaires are available at http://goo.gl/0Rs2r

	Question	Pacione's Task
Q1.1	Which services are involved in the buy functionality?	**A1**
Q1.3	Based on what Serviz shows you, can you identify which services depend on the OrderProcessorService and which services BusinessService depends on?	**A4**
Q2.2	Did your changes repair the system's functionality to provide correct behavior?	**A2**
Q3	Which period did you choose [to perform maintenance] and why?	**A5**
Q4.1	How are services used over a long period of time?	**A6**
Q4.2	Which services and which periods are those [when services never get used]?	**A7**

TABLE III
ASSIGNMENT QUESTIONS MATCHED TO PACIONE'S TASKS

vi-a	The assignments were too hard for me
vi-b	I felt a lot of time pressure
vi-c	The assignments were very interesting to do
vi-d	I felt enthusiastic about the proposed assignments
vi-e	I got enough guidance for completing the assignments
vii-a	It was clear that the thickness of the arrows was proportional to the number of calls
vii-b	It was clear I was supposed to input dates before any data was shown to me
vii-c	It was clear I could scroll the timeline to access different periods of time
vii-d	It was clear that the arrows represent dynamic dependencies
viii-a	Serviz helped me identify which services were involved in a specific use case
viii-b	Serviz helped me identify where the fault was in the system
viii-c	Using Serviz I could easily infer the dependencies between the different services
viii-a	Serviz also helped me investigate such dependencies during runtime and for different periods of time
viii-a	It was clear, using Serviz, which services are used more often
viii-a	Serviz helped me identify dominating usage patterns of the system (e.g. when is it most often used)
viii-a	By identifying dominating usage patterns, Serviz also allows me to determine the best periods for performing maintenance
ix - Favorite feature	
x - Further remarks	

TABLE IV
POSTTEST QUESTIONNAIRE

These activities are:

A1. Investigating the functionality of (part of) the system.
A2. Adding to or changing the system's functionality.
A3. Investigating the internal structure of an artefact.
A4. Investigating dependencies between artifacts.
A5. Investigating runtime interactions in the system.
A6. Investigating how much an artefact is used.
A7. Investigating patterns in the system's execution.
A8. Assessing the quality of the system's design.
A9. Understanding the domain of the system.

The scope of our tool forces us to discard three out of the nine activities, i.e., activities A3, A8 and A9. They refer to internal artefact structure, system design quality and system domain, respectively. All other activities have been specifically targeted by our assignment.

The assignment starts with an anecdotal story of a recently hired developer (the participant) who receives a call from a customer currently unable to purchase stock using Stonehenge. The participant is then asked to attempt to purchase stock and is faced with an error.

Afterwards, the participant is asked to load Serviz and analyze the runtime information collected last year (when the system was working normally) and compare it with the information collected during the previous few minutes. By comparing the two, the participant should be able to figure out that the `QuoteServiceV1` is never invoked.

In order to have more truthful results, the assignment is divided in two parts and the participants are only handed the second part after they completed the first part. This is done because the second part of the assignment, where the fault is meant to be fixed, reveals the nature of the fault in the system. The fault is then revealed in order to let the participant continue with the assignment, in case the participant was not able to pinpoint it in the first part.

The second part of the assignment contains two questions: one of which asks the participant to pick a suitable time to deploy the repaired service into the production system, and the other tests whether the user is able to identify the periods of time when the system is not used.

The assignment was designed to be performed in two hours and it has been carefully tailored to target Pacione's maintenance activities as described in Section III. How the assignments match Pacione's six activities that we consider is shown in Table III.

C. Pilot

In order to find and eradicate possible problems with the tool and the experimental set-up, we also performed a pilot run with an additional participant. This pilot run allowed us to minimize unexpected problems with our tool. For instance, it allowed us to amend a problem with the experimental set-up where the data was not being logged correctly by the services framework. It also allowed us to polish the assignment questions to make them clearer and easier to follow.

IV. RESULTS

Following is the data obtained from the pretest (Fig. 1)[5] and posttest (Figs. 2 and 4) questionnaires, along with a description of the collected data.

[5]The radar charts should be interpreted as follows: each axis of the chart represents one question, the minimum and maximum values of the respondents answers are marked by the colored area, the median score is indicated by the (blue) line.

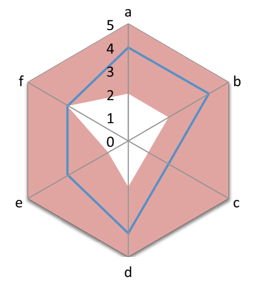
A. *Pretest Data*

In this section we analyze the data collected from the pretest questionnaires. We look at the participants' background, focusing on questions that might have had a direct impact on their experience with the assignment.

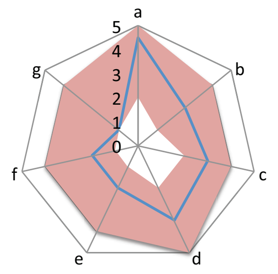
Subject Profile. For our user study involving Serviz, we recruited eight volunteers in the computer science faculty of the Delft University of Technology. The participants have various backgrounds and the group is composed of four MSc students, two PhD students (one of which with five years of experience as a software engineer), one post-doctoral researcher and a software engineer. All participants are male with ages ranging between 23 and 32 years of age.

None of the participants received training on Serviz and all participants were using it for the first time. The premise is that the tool is intuitive enough for someone with a generic software engineering background to grasp.

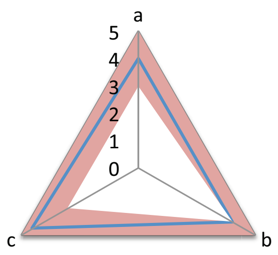
Experience with Software Development. We refer now to the participants' experience with software development, shown in Fig. 1i. Based on this data we can infer the participants have a fair amount of experience with Java development (Fig. 1i-a, median 4) and while most participants are familiar with web services and SOA (Fig. 1i-d, median 4 and 1i-f, median 3) not all of them had participated in web service development before (Fig. 1i-e, median 3, but some reported to have no experience). From these results we can also gather that participants are fairly acquainted with Eclipse for Java development purposes (Fig. 1i-b, median 4). The participants reported to be less experienced with Maven (Fig. 1i-c, median 2). This particular observation is unlikely to have an impact on the outcome of the experiment as step-by-step instructions on how to use Maven were provided (Maven was required to recompile any changed web services). Furthermore, the participants were encouraged to ask questions during the experiment in case of uncertainty.



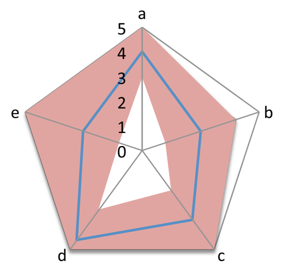
(i) Experience with Software Development Practices



(ii) Opinion on Soft. Maintenance



(iii) Opinion on Dist. Software



(iv) Views on Dynamic Analysis

Fig. 1. Pretest Data

Opinion on Software Maintenance. As part of the pretest questionnaire, we also gathered the participants' opinion on software maintenance. From our data we observed that most participants had performed software maintenance before (Fig. 1ii-a, median 4.5) even though this is not something they do often (Fig. 1ii-b, median 2.5). Based on the participants' input, this is also a task mostly performed on their own software (Fig. 1ii-c, median 3). When asked in particular about maintaining distributed systems, the results reveal this is also something not often performed by the participants (Fig. 1ii-e, median 2). This is further made clear by the responses to whether they had used dynamic or static analysis for this task (Fig. 1ii-f, median 2 and Fig. 1ii-g, median 1). Also in the context of software maintenance, we asked the participants whether they had used software visualization tools before (Fig. 1ii-d, median 3.5) and based on the responses we can assume the participants are, at least, familiar with the concept of such tools.

Opinion on Distributed Software. Looking at the participants' opinion regarding distributed software, the results show that the participants see software maintenance as a difficult task (Fig. 1iii-a, median 4) made even more difficult by the distributed dimension (Fig. 1iii-b, median 4). The general consensus was also that making use of runtime information can help maintain distributed software (Fig. 1iii-c, median 4.5).

Opinion on Dynamic Analysis. Lastly, we gather the participants' opinions on dynamic analysis. When asked in general whether the participants used static or dynamic analysis, the tendency is vastly in favor of static analysis (Fig. 1iv-a, median 4 vs Fig. 1iv-b, median 2.5). This makes it clear that the participants are not extremely familiar with dynamic analysis techniques. Despite this fact, the participants in general have performed dynamic analysis at least once (Fig. 1iv-c, median 3.5) and a smaller percentage has used dynamic analysis tools on distributed systems (Fig. 1iv-e, median 2.5). It is also worth noting that participants are well aware of the added value dynamic analysis provides over static analysis (Fig. 1iv-d, median 4.5).

B. *Posttest Data*

In this section we discuss the data collected after the participants performed the assignment. For some questions we also cross-check their perceived values with those expected in the pretest questionnaire in order to measure how well Serviz meets the participants' expectations.

Overall Experience. Our posttest results regarding the overall experience with Serviz have been mostly positive with small exceptions. When asked whether the assignments were too hard to do or if there was time pressure, the participants responded with a median of 2 and 1 respectively (questions vi-a and vi-b). This excludes difficulty and lack of time as potential factors which affect the final outcome of the experiment. This is by all accounts expected, as all the participants

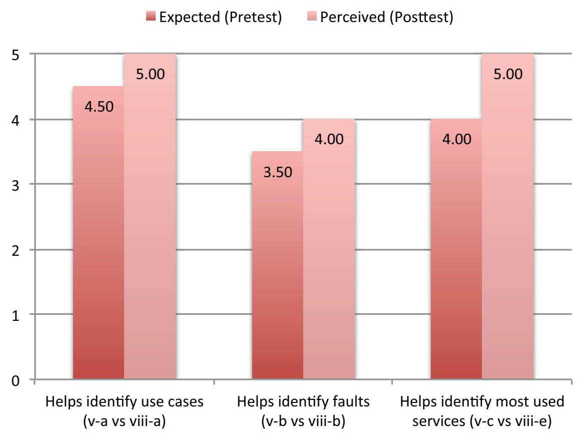


Fig. 2. Expectation vs Perception (Median)

managed to finish the assignment within the allotted two hours. The participants were also quite interested in the experiment and were enthusiastic about it (questions vi-c and vi-d, with medians of 4 in both accounts) which is further supported by the fact that all participants but one left further remarks at the end of the assignment. The consensus was also that they got enough guidance for completing the assignments (question vi-e, median 4).

Identifying use cases. One of the purposes of making use of the runtime topology of a service-oriented system is to identify web services that together participate in providing a specific functionality. We measured how well Serviz achieves this by asking the following question before and after the assignment "Serviz helped me identify which services were involved in a specific use case" (Fig. 2, v-a vs viii-a).

We can now compare what the participants expected of Serviz before using the tool and how they perceived Serviz after using it. In particular, for question v-a, we see that although the participants already had quite high expectations (median 4.5), the perceived usefulness of Serviz after having used the tool rose to a median of 5. This rise in the expected vs. perceived score means that not only were the participants confident that such a feature would likely help, they were even more convinced after having used Serviz.

Manual fault identification. Before each participant started the assignment, a fault was intentionally added to the software system under maintenance. This fault consisted of disabling the invocation from the `OrderProcessorService` to the `QuoteService`, resulting in the *purchase stock* action of the system to fail with an error displayed to the participant.

One of the tasks the participants were asked to solve during the assignment was to fix this fault. One of the goals of adding this fault was to infer whether the runtime topology of a service-oriented system aids in manually identifying such faults in an effective manner.

This facet is assessed through our posttest questionnaire where the participants are asked whether "Serviz helped [them] identify where the fault was in the system" (Fig. 2, v-b vs viii-b). This question was met with a median of 3.5 in the pretest which rose to a median of 4 after the participants performed the assignment. From this we understand that, perhaps based on the wording (Table II, Tool description) which does not explicitly mention fault detection, the participants were not extremely confident Serviz would help in this task. However, after using Serviz they were convinced that Serviz does indeed help in identifying where the fault was located.

Identifying most used services. Directly linked to RQ3 is the need to identify the most used services in a service-oriented system. This aspect of our approach is measured in both the pretest and posttest with the question "It was clear, using Serviz, which services are used more often".

The results show that the participants were already expecting Serviz would help them identifying the most used services and this was confirmed after the assignment with a rise of the median from 4 to 5 points (Fig. 2, v-c vs viii-e).

Identifying dependencies between different services. In the posttest we also asked the participants whether "using Serviz [they] could easily infer the dependencies between the different services" (Fig. 4, viii-c). This question, somewhat related to question viii-a, is aimed at making a more direct assumption on whether Serviz can at least provide an overview of which services depend on which other services. In this specific case, having already obtained the maximum score for question viii-a with a median of 5, the participants' opinion remains consistent also with a median of 5 for question viii-c. This makes it clear that Serviz does indeed help identifying dependencies as well as identifying the use cases related to these dependencies.

We elaborated on this question by asking whether the above is also possible whilst including the time dimension (Fig. 4, viii-d). While the median for this particular question lowered to 4 points, it is still clear that Serviz helps when the time dimension is added to the equation. The reason for the median being lower compared to not including the time dimension might be related to the participants' frustration with the date picking controls which were, at times, buggy. This is discussed further down, in the additional remarks from the participants.

Identifying periods of time for maintenance. Another goal of using the system's runtime topology is that of finding the most suitable time for performing maintenance. By following our assignment, the participants were also expected to find such periods of time. The results related to this task were asked only in the posttest due to the more complex nature of the questions (viii-f and viii-g).

The results for these particular questions (Fig. 4) have been somewhat disappointing with the median for both cases lying at 2.5 points. This result might be related to either the data shown which is not enough to provide this information, or due to the tool which does not do a good enough job at displaying such data. Based on the participants' comments (further discussed in the additional remarks), we are inclined towards a shortcoming of the tool itself.

Usability. Regarding the usability of Serviz, a very disappointing factor comes to light. Our data shows that the participants did not notice that the thickness of the arrows connecting the web services was proportional to the number of

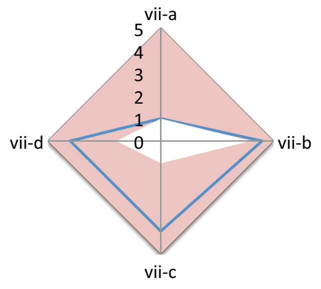


Fig. 3. Usability Questions (Median)

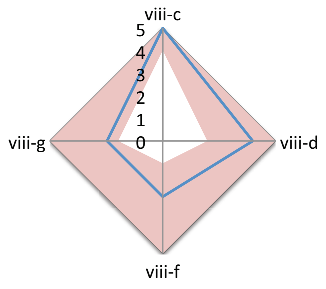


Fig. 4. Additional Posttest Questions (Median)

calls (Fig. 3, vii-a, median 1). This might have had a negative impact on how the participants used the tool, as understanding how many requests are made per web service is a crucial means to an end in order to find which web services are used less often and when. In hindsight the relationship between the arrow thickness and number of invocations is not always obvious; in order to overcome this issue, the next version of Serviz will contain a legend which explicitly describes line thickness as a metric.

Despite this disappointing outlier in our data, the remaining features of Serviz seem to have been obvious and intuitive enough. The participants claimed it was clear they were supposed to input dates before any data was shown (Fig. 3, vii-b, median 4.5), it was clear they could scroll the timeline to access different periods of time (Fig. 3, vii-c, median 4) and lastly that it was clear the arrows represent dynamic dependencies between services (Fig. 3, vii-d, median 4).

Additional remarks. From the open question at the end of the posttest we obtained more detailed feedback from the participants on what can be improved for future work. Several participants were perplexed by the purpose of the timeline, claiming it had no practical usefulness. Some participants suggested the timeline should include a histogram so it becomes easier to visualize at a glance when is the system not being used.

This timeline was initially envisioned to visualize all the static configuration changes and it allows maintainers to load each of these changes. This feature was, however, not targeted by the assignment as the "load latest static data" button is more intuitive for our maintenance scenarios.

Some participants also mentioned that the date picking process is sometimes buggy (e.g., when the participants input a date manually rather than using the date pickers) and moving the services around in the graph can also be quite slow. This is mostly due to the prototype nature of the tool and is something we will improve in further versions. We also believe that while these minor bugs might have somewhat frustrated the participants, they have clearly not had a direct impact on the results which remained generally positive.

Another common remark was the fact that Serviz can only be used when the browser is maximized. Any other window configuration of the browser forces the date pickers to hide which in turn makes it impossible to use the tool.

Finally, the last remark has to do with the positioning of the web services in the graph. By default, and whenever the dates are changed, the web services revert to their initial position, where they overlap on top of each other. This was reported by the participants as being very cumbersome to manage. Whenever the dates are changed, the participants were forced to rearrange the services to visualize the runtime topology of the system in that particular interval. Looking for particular periods of time when specific services do not show is certainly a task that requires copious amounts of patience and time. This is something we also want to fix in future versions by providing a usage histogram per service.

V. Discussion

A. Revisiting the Research Questions

Our first research question (**RQ1**) is aimed at determining whether "*the runtime topology helps in identifying services involved in a specific use case?*" Based on our experiment results and referring to Fig. 2 (v-a vs viii-a), the participants replied with a median of 5 points when asked this question directly after using Serviz. This allows us to infer that indeed, the runtime topology of a system (more specifically, the one provided by Serviz) helps in addressing RQ1.

As for the second research question (**RQ2**), we wanted to find out if "*the runtime topology helps in debugging a SOA-based system?*" While the term *debugging* can describe a whole range of techniques and approaches aimed at finding and removing faults from a software system, we were interested in investigating whether our approach would help with this task on a service-oriented system. Our approach has indeed helped the participants, as demonstrated by the median of 4 points in Fig. 2 (v-b vs viii-b). In fact, all the participants managed to successfully find and fix the intentional fault of the system.

The third research question we presented (**RQ3**) asks whether "*the runtime topology augmented with the time-dimension is suited to identify services that are used more often (e.g., to optimize them)?*" The answer to this question is also positive, supported by the results shown in Fig. 2 (v-c vs viii-e). The participants have given Serviz a median score of 5 for this particular feature, which we can interpret as a positive outcome of Serviz in addressing this challenge.

Lastly, the fourth reseach question (**RQ4**) tries to identify if *the runtime topology helps to identify periods in time where the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?* This question was, unfortunately, met with disappointing results. The participants have, for these two posttest questions, replied with a median of 2.5 points as shown in Fig. 4 (viii-f and viii-g). These results are, in our opinion, not related to the approach but to how we implemented the visualization of this particular data. Judging by the further remarks left by the participants, we understood that using our implementation of this particular feature is not trivial to use and requires a significant amount of time in order to obtain useful information about the services' usage. This is something we will mitigate in further versions of Serviz by providing a histogram to better visualize usage peaks.

B. Threats to validity

In this section we present a discussion of how the results of our experiment might be challenged. We discuss internal validity, namely whether the participants might have been directly affected by factors we did not consider, and external validity, meaning whether our results are generalizable.

1) Internal Validity: The participants in this experiment were being told to use Serviz. It remains to be known whether such a tool fares better than other means of system analysis. While the tool seems to have, indeed, helped the participants find and fix the problem, it is unknown whether they could have achieved the same without the tool. It should also be said that Serviz simply aims at making it easier to achieve the solution and not to simply provide the solution.

Another threat to the internal validity of our study concerns the participant group chosen. The participants have been told to be impartial when evaluating the tool but, despite this, they might have felt emotionally biased to give favorable results.

2) External Validity: Our main concern with external validity has to do with the performance impact of our approach. We are particularly concerned with the performance overhead added by the data collection step. However, the data collection is supported by a robust framework used by a company with a large web services infrastructure (eBay).

The storage requirement of our approach is also quite high. In a system with a large traffic of web service requests, a large amount of storage space is required in order to maintain the system's state over time. This threat can be mitigated by using compression techniques, e.g., as applied in the Compact Trace Format (CTF) [14].

The applicability of our results to larger systems is also something we tried to mitigate by using Spicy Stonehenge. Spicy Stonehenge, despite its small size, contains all the ingredients of an industrial system, including complex interactions between services and a well-specified domain.

Another issue is that the group of participants was mostly composed of students. Despite this, two of the students have working experience in industry. Another factor is that all participants in the experiment should be considered *software immigrants* [15] and as such, the findings that we report upon are based on developers trying to find their way in a previously unknown software system and do not reflect situations were developers are already familiar with the system. We acknowledge that a follow-up study should also investigate the usefulness of Serviz in situations where the developers are already familiar with the domain.

Lastly, the tasks chosen for the assignment might not be realistic. We tried to mitigate this by looking at the nine principle maintenance activities identified by Pacione et al. [13] and tailoring the assignment tasks to cover six of these activities.

VI. Related work

In general, the increased maintenance complexity of SOA-based systems has been acknowledged and emphasized by Lewis and Smith [16], requiring for instance, impact analysis for an unknown set of users, or increased number of externally accessed services to be considered in maintenance. These are asking for a readjustment of current maintenance practice for SOA-based systems at large.

For this article, we started by analyzing the survey of Cornelissen et al. [5] on program comprehension through dynamic analysis, which, among others, lists the work of De Pauw et al. [17]. They describe a web services navigator for generating service topologies, and focus on detecting incorrect implementations of business rules and "excessively chatty" communications. Our approach is different w.r.t. two significant improvements to the service topology: it allows to identify the method of an invoked service plus its invoking client, and it includes the time dimension providing a historic view on the topology.

White et al. [7] present a dynamic analysis approach to aid the maintenance of SOA-based composite applications where they propose using a feature sequence viewer to recover sequence diagrams from such systems. This approach, however, does not seem to clearly provide a good basis for understanding the topology of a running service-oriented system and rather focuses on mapping features to software artifacts.

In other work, White et al. [18] investigated the information of developers during the maintenance of SOA-based systems. They established that the first question a maintainer must ask is "how does the software work now?" The authors also motivated that maintainers require immediate additional assistance in understanding an application's data types.

Finally, we investigated citations to the aforementioned papers, in particular the systematic survey by Cornelissen et al. [5] in order to find more recent additions to the body of knowledge. Unfortunately, this search has yielded no extra relevant related work.

VII. Conclusion

In this article, we investigate how the topology of a running service-oriented system can help in its maintenance. More specifically, our contributions are:

- The runtime topology augmented with the time dimension.
- Serviz, an open-source implementation of this approach.
- A user-study with 8 participants evaluating the effectiveness of such an approach.

We now summarize how our approach and the tool address our original research questions formulated in Section I:

RQ1 *Does the runtime topology help in identifying services involved in a specific use case?* Our experiment participants agreed that by providing them with a runtime view of what the system looks like at any point in time, they can easily discover which services are involved in providing a specific functionality.

RQ2 *Does the runtime topology help in debugging a service-oriented system?* Similarly, participants have also indicated that they strongly agree that by comparing normal circumstances of a system to periods when a fault is occurring, the runtime topology aids them in debugging.

RQ3 *Does the runtime topology augmented with the time-dimension help to identify services that are used more often (e.g., to optimize them)?* Our user study indicates that participants also found that the runtime topology augmented with an invocation count helps them to quickly identify the most used services in a specific time interval.

RQ4 *Does the runtime topology help to identify periods in time when the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?* Our experiment participants were not convinced Serviz helps them with this task, which we mainly attribute to our own implementation of the runtime topology falling short. In particular, this task involves manually scrolling the data hour by hour, while the participants would like to see this task automated.

A. Future work

As future work we propose to bring the dimension of users to the runtime topology. This way we can determine which users are using which services the most. This added information makes it even easier to filter the data provided by the runtime topology and it makes it also easier to find periods of time which minimize perceived downtime.

We also want to explore how the runtime topology can help us understand service versioning. In that line of research we aim to make statistical assertions (based on historical usage data) about whether a certain version of a service has become dead code and should be undeployed.

Another goal is to keep on developing Serviz, make it more user-friendly and improve its visualization. Subsequently, we aim to perform a full-fledged controlled experiment [19] aimed at measuring the actual time-gain developers have when using the runtime topology during maintenance.

ACKNOWLEDGMENTS

The authors would like to acknowledge NWO for sponsoring this research through the Jacquard ScaleItUp project. Also many thanks to the participants in our user study and our industrial partners Adyen and Exact.

REFERENCES

[1] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*, ser. Apic Studies In Data Processing. Academic Press, 1985.

[2] N. Gold, C. Knight, A. Mohan, and M. Munro, "Understanding service-oriented software," *IEEE Software*, vol. 21, no. 2, pp. 71–77, 2004.

[3] A. Zaidman, M. Pinzger, and A. van Deursen, "Software evolution," in *Encyclopedia of Software Engineering*, P. A. Laplante, Ed. Taylor & Francis, 2010, pp. 1127–1137.

[4] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[5] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Software Eng*, vol. 35, no. 5, pp. 684–702, 2009.

[6] G. Canfora and M. Di Penta, "New frontiers of reverse engineering," in *Future of Software Engineering (FOSE).* IEEE CS, 2007, pp. 326–341.

[7] L. J. White, T. Reichherzer, J. Coffey, N. Wilde, and S. Simmons, "Maintenance of service oriented architecture composite applications: static and dynamic support," *J. Softw. Maint. Evol.: Res. Pract.*, *To appear*.

[8] M. Kajko-Mattsson, G. A. Lewis, and D. B. Smith, "Evolution and maintenance of soa-based systems at sas," in *Proc. Hawaii Int'l Conf. on Systems Science (HICSS).* IEEE CS, 2008, p. 119.

[9] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

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

[11] D. Campbell, J. Stanley, and N. Gage, *Experimental and quasi-experimental designs for research*. Rand McNally, 1963.

[12] E. Babbie, *The practice of social research, 11th edn.* Wadsworth Belmont, 2007.

[13] M. J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *Proc. of the Working Conf. on Reverse Engineering (WCRE).* IEEE CS, 2004, pp. 70–79.

[14] A. Hamou-Lhadj and T. C. Lethbridge, "A metamodel for the compact but lossless exchange of execution traces," *Software and System Modeling*, vol. 11, no. 1, pp. 77–98, 2012.

[15] S. Elliott Sim and R. C. Holt, "The ramp-up problem in software projects: a case study of how software immigrants naturalize," in *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE CS, 1998, pp. 361–370.

[16] G. Lewis and D. Smith, "Service-oriented architecture and its implications for software maintenance and evolution," in *Proceedings Frontiers of Software Maintenance*. IEEE CS, 2008, pp. 1–10.

[17] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar, "Web services navigator: Visualizing the execution of web services," *IBM Systems Journal*, vol. 44, no. 4, pp. 821–846, 2005.

[18] L. White, N. Wilde, T. Reichherzer, E. El-Sheikh, G. Goehring, A. Baskin, B. Hartmann, and M. Manea, "Understanding interoperable systems: Challenges for the maintenance of soa applications." IEEE CS, 2012, pp. 2199–2206.

[19] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 341–355, 2011.