Proceedings Paper:
Sargeant, A, Townend, P, Xu, J orcid.org/0000-0002-4598-167X et al. (1 more author) (2013) An Evaluation Framework for Assessing the Dependability of Dynamic Binding in Service-Oriented Computing. In: 2013 IEEE 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. ISORC, 19-21 Jun 2013, Paderborn, Germany. IEEE . ISBN 978-1-4799-2111-9

https://doi.org/10.1109/ISORC.2013.6913206

An Evaluation Framework for Assessing the Dependability of Dynamic Binding in Service-Oriented Computing

Anthony Sargeant, Paul Townend, Jie Xu, and Karim Djemame
School of Computing
University of Leeds
Leeds, LS2 9JT, United Kingdom
{scs5ajs@leeds.ac.uk, p.m.townend@leeds.ac.uk, j.xu@leeds.ac.uk, k.djemame@leeds.ac.uk}

***Abstract*—Service-Oriented Computing (SOC) provides a flexible framework in which applications may be built up from services, often distributed across a network. One of the promises of SOC is that of Dynamic Binding where abstract consumer requests are bound to concrete service instances at runtime, thereby offering a high level of flexibility and adaptability. Existing research has so far focused mostly on the design and implementation of dynamic binding operations and there is little research into a comprehensive evaluation of dynamic binding systems, especially in terms of system failure and dependability. In this paper, we present a novel, extensible evaluation framework that allows for the testing and assessment of a Dynamic Binding System (DBS). Based on a fault model specially built for DBS's, we are able to insert selectively the types of fault that would affect a DBS and observe its behavior. By treating the DBS as a black box and distributing the components of the evaluation framework we are not restricted to the implementing technologies of the DBS, nor do we need to be co-located in the same environment as the DBS under test. We present the results of a series of experiments, with a focus on the interactions between a real-life DBS and the services it employs. The results on the NECTISE Software Demonstrator (NSD) system show that our proposed method and testing framework is able to trigger abnormal behavior of the NSD due to interaction faults and generate important information for improving both dependability and performance of the system under test.**

Keywords—Dependability evaluation, dynamic binding, Service-Oriented Architectures, testing, Web services

I. INTRODUCTION

Service-Oriented Computing (SOC) provides a framework that allows for the flexible integration of distributed, networked services [1]. Through loose coupling of service instances, and by utilizing standardized intercommunication methods and well-defined interfaces, one or more of these services can be aggregated into more complex applications. Service-Oriented Architectures (SOA) provides a logical architecture for constructing these service-based applications [2]. Furthermore, SOC and SOA facilitate the agile choice of services according to the needs of the service consumer, such that services can be swapped out for those that are functionally equivalent [3].

One of the promises of SOC is that of Dynamic Binding of services where abstract consumer requests are bound to concrete service instances at runtime. Existing work involving dynamic binding investigates certain aspects of the design and implementation challenges, such as dynamic service composition [4, 5], how to best match requests to services [6, 7], dynamic service discovery [8], and dynamic reconfiguring of services [9]. Other existing research proposes frameworks for the implementation of context-aware dynamic binding such as in ubiquitous computing environments [1]. What is clear from the current work is that there exist several components that help to provide the necessary behavior for dynamic binding.

With existing work, the focus has been on the evaluation of the methodology employed to enable dynamic binding, such as in the case of [4]. However, a dependability evaluation of their respective Dynamic Binding Systems (DBS) is not considered. Indeed, this represents a gap in the literature as the binding system itself could be subject to faults, and the behavior of the dynamic binding system could be non-deterministic [10]. Because dynamic binding is one of the key enabling mechanisms to realize the promises of SOC, there is an urgent need for a good understanding of the dynamic behavior of a DBS and how different types of fault would affect its operations. Unfortunately, to the best of our knowledge, the known fault models developed for SOC, such as those proposed by [11, 12], do not cover the kinds of failure modes specific to a DBS.

The contributions of this paper are two-fold. First, we design and implement a Dynamic Binding System Evaluation Framework (DBS-EF) tool that employs the system and fault models to generate the types of fault that may affect a DBS at runtime. These faults are then injected at real time as the system is running in order to evaluate the behavior of the DBS. Secondly, we apply the DBS-EF framework and tool to a real-life system in order to demonstrate the effectiveness of the proposed framework.

The rest of this paper is organized as follows: Section II describes dynamic binding and discusses the related challenges and issues with a DBS. Section III examines the dependability concerns for dynamic binding in SOC and introduces a fault model specially developed for service-oriented dynamic binding. Section IV describes the DBS-EF framework and its implementation. Section V presents a case study; by introducing the NECTISE Software Demonstrator (NSD) which is used to illustrate the effectiveness of the DBS-EF. We

include in this section results and analysis of experimentation on the NSD. Section VI concludes the paper and points to future work.

II. DYNAMIC BINDING IN SOC

Dynamic Binding in Service-Oriented Computing (SOC) is the binding of a client request, to a concrete service instance at runtime. This is typically done with a brokering service who will, on behalf of the client, select the 'best' service to serve the client's request. For our work, the term 'best service' is used to refer to a service that will meet with the highest rank, both the functional and non-functional requirements of the client's request [4, 5]. (Ultra-late) binding is possible and may take place at the point in time when the service is needed so as to maximize the flexibility and the opportunity of acquiring the best service. Furthermore, functional requirements refer to the parts of the request that is concerned with the interface of a service, e.g. the method name, number of parameters, parameter types and expected response. Non-functional requirements refer to the Quality of Service (QoS) [12]. Typical examples of QoS attributes used in SOC include reliability, availability, response time and cost [4, 5, 13].

Dynamic Binding operations in a DBS normally give rise to the following abstract workflow:

1. The client sends an abstract request to the DBS, which acts as a broker between the consumer and provider(s).
2. The request is received and processed to ascertain the aims of the consumer's request.
3. Once processed, the request is passed to the service discovery mechanism to discover functionally-equivalent candidate service or services that will meet the client's request.
4. Once the candidate services have been obtained, the service selection mechanism will use the consumer's nonfunctional requirements to rank the services in order to find the best service. Either the top ranked candidate service is chosen to be the concrete service instance or a service is chosen from a pool of services that meet or exceed the request's minimum nonfunctional requirements [4, 14].
5. The integration mechanism ensures that the consumer's abstract request is interoperable with the concrete service interface.
6. Following the integration phase, the request is then passed to the concrete service instance.
7. The response from the service instance is passed back to the integration mechanism so that the response can be formatted in such a way that is understood by the consumer.
8. A context monitor is also included so that the current context of the consumer's request is maintained. If there is a change, then the monitor can ensure that the system adapts as necessary.

The workflow indicates several components that would be required in order to provide the necessary functionality. Consequently, we have developed the following system model for Dynamic Binding in SOC, as shown in Figure 1.

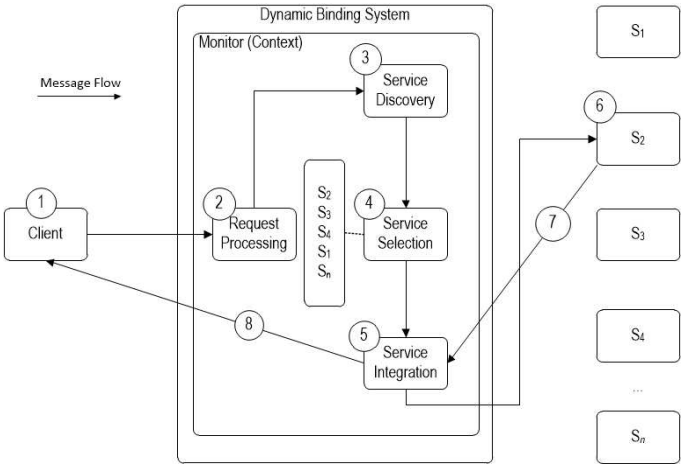

Figure 1. System Model of Dynamic Binding in SOC.

The system components are described below:

- **Request Processor** - Here is where a DBS will analyze a client's request for functional and non-functional requirements. The information out of the analysis process is used to determine the best service to meet a client request [8].
- **Service Discovery** - The functional requirements of the request (i.e. the method signature) are used to find candidate services. Discovering a service then is a simple case of searching through a repository of registered services.
- **Service Selection** - The non-functional requirements of the request (i.e. the QoS attributes) are used to determine which of the candidate services meet, or exceed the minimum nonfunctional requirements. The 'best' service is then chosen from those services using a predetermined algorithm. (Note that when there are two or more top-ranked services the selection may be made randomly or using additional information.)
- **Service Integration** - If the interface of the chosen service differs from that of the request and/or the response differs from what the client expects, then the integrator should mediate those differences [7].
- **Context Monitoring** - If the context in which the original request was sent changes, then this might affect the decision of which service to bind to. Therefore, it is necessary to monitor for any changes in context and to allow the selection of the most appropriate service to meet the request, given the context change [14].

III. DEPENDABILITY OF DYNAMIC BINDING

From the system model above, we are able to derive a fault model to aid the assessment of the dependability of a DBS.

A. Fault Model

In order to establish an appropriate fault model for the evaluation framework to employ, we must first consider the previous work in the area by Avizienis et al. [15], Jhumka [12], Brüning [11] and Chan [16]. Avizienis et al. provided basic concepts and taxonomy for dependability in software systems. Their work provides the foundation from which our fault model is derived. Jhumka and Brüning were able to apply these concepts to the area of Service-Oriented Computing. Their research gave a more specific model which Chen was able to then apply to Web Services. These models are hierarchical in nature as each subsequent model seeks to encapsulate the types of fault that can affect specific areas of SOC.

To the best of our knowledge, existing work in the area of dependability does not consider the types of fault that can affect a DBS, something that we have addressed in our previous work [17]. We present our fault model in the table below, by giving the fault categories and specific examples of the representative types of fault affecting a DBS and also how a DBS is expected to behave in the presence of these faults.

TABLE I. FAULT CATEGORIES

Fault Category	***Fault Description***	***DBS Behavior***
Communication Fault	• Crash of a service instance • Crash of hosting environment • Service hang • Duplication of messages • Omission of messages • Delay of messages	DBS should detect fault using time-out values, or DBS should rebind to another service, or DBS should return an exception/fault message and not attempt a rebinding.
Interaction Fault at the service side (Interface Fault)	• Too few parameters • Too many parameters • Invalid method name • Invalid parameter types • Parameter value out-of-range	DBS should detect via fault message from service, or rebind to a service that provides the correct interface, or mediate differences between interfaces, or DBS should return an exception/fault message and not attempt a rebinding.
Interaction Fault at the client side (Invalid Request)	• Missing token(s) • Invalid parameter types • Parameter values out-of-range • Invalid method name • Empty method call • Zero parameters • Empty parameters	DBS should return a meaningful error message/exception notifying the client that the request cannot be processed or no service(s) are available to service the request.

We note that whilst value faults are still something that would need to be handled by the system as a whole, it is out-of-scope for this work as the DBS is only responsible for forwarding responses from the chosen service to the client, and not for the correct content of that message. Any other value fault in the context of the working of the DBS, i.e. invalid SOAP message can be considered a value fault as it is the contents of the payload itself, would be picked up by the error handling mechanism of the middleware such as a `SOAPException` in Java [18]. As a result, we assume that the values sent by the services are fault-free.

Interaction faults fall into two sub-categories: *Invalid Request* i.e. any outgoing interaction from the client to the DBS, and *Interface Fault* i.e. any fault that arises from a mismatch between the concrete request, and the interface of the chosen service.

We acknowledge that it is impossible to ascertain every possible fault that could manifest in a system, however, it is our intention that this model is extensible, and consequently we have developed our evaluation framework to also be extensible as new test cases arise.

IV. AN EVALUATION FRAMEWORK FOR DYNAMIC BINDING SYSTEMS

In order to evaluate the dependability of a DBS, it is necessary to develop a simple but effective testing framework. In this section, we introduce our Dynamic Binding System Evaluation Framework (DBS-EF) for such a task.

A. Dynamic Binding System Evaluation Framework

The DBS-EF consists of a wrapper system, which encloses the DBS itself. Due to the distributed nature of SOC, it is desirable that the DBS under test does not need to be co-located in the same operating environment as the DBS-EF.

To combat the challenge, we provide a system that utilizes handlers to intercept messages to and from the DBS, and we treat the DBS itself as a *black box* system. Where this approach has its strength, is that it does not require any knowledge of the system under test – the system behavior is observed as a function of its input and output [19]. This approach, however, is not without limitations. Without knowledge of the underlying code, it is not possible to locate where a failure may manifest itself, or indeed which parts of the system are being affected by the fault [20]. In the case that the system's internal structure is available and therefore white-box testing is applicable, the structure information can be used to guide the development of test cases within the DBS-EF.

To achieve our aim of 'wrapping' the DBS within the test environment, in the DBS-EF the client and candidate services are under the control of the framework in order to provide assessment at both client-side and service-side execution. Under this scheme, Interaction Faults are introduced as client-side Invalid Requests, whereas Interface Faults occur at the service-side. Communication Faults can take place during either client-side or service-side execution, or at both sides in the case of network loss at the DBS. For the purposes of this paper, we will concentrate on Interaction Faults occurring between client and DBS, and between DBS and the chosen candidate service.

The DBS-EF contains a series of test cases that are based on the domain context of the DBS under test, and test campaigns are administered by the Dynamic Binding Fault Coordinator Service (DBFCS). This service is contacted by the

candidate services in order to ascertain the type of fault that is to be injected into the messages returned to the DBS.

Our evaluation framework, as illustrated in Figure 2, works as follows:

- A client is created that will be used to send test requests to the DBS under test.
- Test cases are created that will be used to assess the dependability based on the domain context of the DBS under test.
- The client alerts the DBFCS as to which test campaign to use (e.g., Invalid Request, Communication Fault or Interface Fault).
- The client sends the request to the DBS.
- The DBS finds and selects the 'best' service from a list of candidate services and forwards the request to the chosen service.
- The service processes the request and passes the response to the service's handler.
- The handler then contacts the DBFCS and requests a fault to inject.
- The instrumented message is then returned to the DBS to be forwarded back to the client.

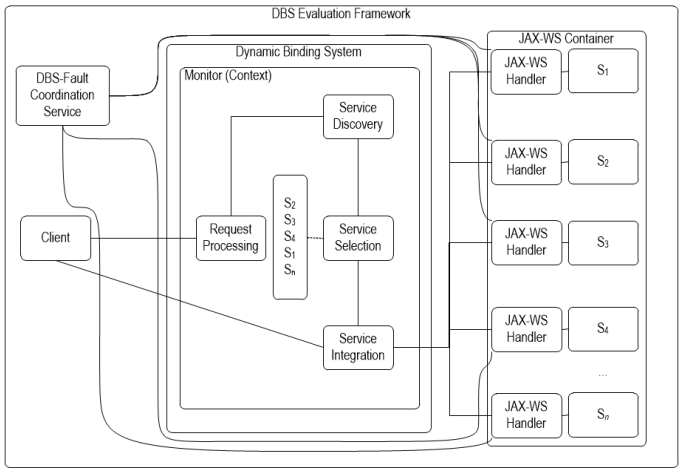

Figure 2. Dynamic Binding System Evaluation Framework

B. DBS-EF Implementation

As with the previous implementations by Looker et al [21] (method 1), and by Farj et al [22] (method 2), our implementation uses Web services as it is a commonly used example of SOC. In developing our DBS-EF, we have adopted similar approaches to that of their work that SOAP messages are intercepted and instrumented prior to being returned to the DBS, but after being processed by the service.

Methods 1 and 2 both concerned themselves with the testing of services themselves. Looker et al first developed a technique that allowed the interception of SOAP messages and instrumented them based on a test campaign for a single Web service. The main drawback to this approach is that their system relies on deploying the services under test on an instrumented Axis web container.

Method 2 extended this work, by making the fault injection mechanism distributed. This method employed proxies that sat in front of each Web service. This afforded the authors a distributed fault injection framework which no longer required the service under test to be co-located on an instrumented server.

Our framework consists of elements of both methods in order to test a DBS. However, unlike methods 1 and 2, the aim of the DBS-EF is not to test a service but to test the dependability of the DBS itself. Due to the nature of those test frameworks, they would not be suitable for the testing of a DBS as they focus only on interactions between a client and a service. However, it is important to be able to test both client-DBS and DBS-service interactions. As a result, the DBS-EF has been designed to assess such interactions as part of the evaluation process.

We have developed our framework using Java 5.0 and JAX-WS from the Glassfish 2.1.1 platform. The fault injection mechanism has been developed using the inbuilt JAX-WS handler framework, with the aforementioned distributed DBFCS coordinating the test campaign. Similar to method 2, our system is fully distributed and consequently does not require the DBS to be co-located in the same operating environment as the evaluation framework.

V. CASE STUDY: DBS-EF APPLICATION

To demonstrate the effectiveness of the DBS-EF, we have taken an existing DBS – the NECTISE Software Demonstrator developed for the UK's EPSRC NECTISE project – and we have used it as a case study for the application of the evaluation framework.

A. NECTISE Software Demonstrator (NSD)

The NECTISE Software Demonstrator (NSD) was created as part of the Network Enabled Capability Through Innovative Systems Engineering (NECTISE) program which investigated how loosely-coupled services can be used to describe the functions and Quality of Service for heterogeneous systems and networks [23]. The NECTISE program studied issues related to a larger UK Ministry of Defence (MoD) initiative to enhance military effect through the networking of existing and future military capabilities, under the banner of Network Enabled Capability (NEC).

The scenario aim of the NSD was to model a real-time Region Surveillance capability using dynamic service integration of sensor networks in the NEC battlefield - this allowed a comprehensive 'picture' to be formed of the geographical region based on data communicated to a controller from deployed mobile sensors [13].

The basic application workflow is shown in Figure 3, and can be described as follows:

- A client may submit (real time) requests to the system for information of Point of Interests (POI) for a specified region of interest. In this case study, the QoS parameters employed are sensor availability and sensor response time

in milliseconds. These parameters are specified alongside the region of interest in the request.

- The system will return the related information about the POIs within that region, e.g. current locations of those POIs. This is done by requesting POI information from only those sensors that can 'see' the region of interest.

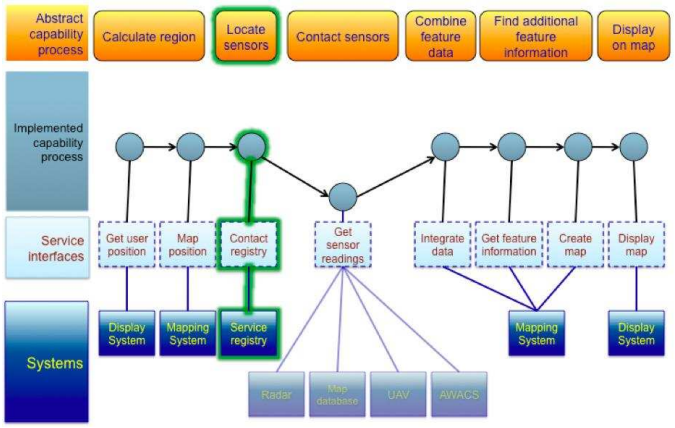

Figure 3. NECTISE workflow from [23]

- The system then pools the information together and filters out any duplicate POIs (as multiple sensors may see the same POI), then encapsulates the information into a single service response which is returned to the client.

The system model for the NSD is illustrated in Figure 4 and shows how the components of the NSD relate to the system model for dynamic binding in SOC. A screenshot of a mapping client is shown in Figure 5 and illustrates the region of interest (the blue box) and shows sensors and POI within the region of interest.

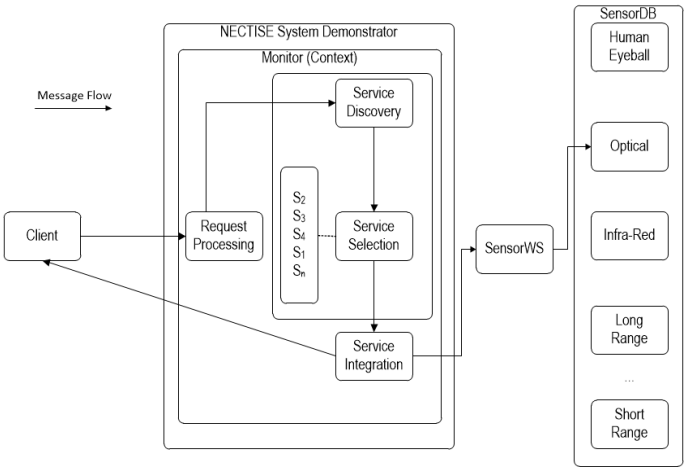

Figure 4. NECTISE System Model in context of Dynamic Binding

In this model, the Service Discovery and Selection components are combined into a single component. Additionally, there is a single gateway service (`SensorWS`) that accesses all sensors within the region of interest. Sensor information is modeled in an SQL database, and the integration mechanism is responsible for the data fusion by removing duplicate sensor information.

The interface of the NSD, consists of a single method: `GetFeatures(MonitorLocation2 ml2)`. This object consists of four floats representing the region of interest ((x1,y1), x2,y2)) and a `ServiceAvailability` object that representing the minimum requested QoS for each sensor. Once processed by the NSD, the method returns a MonitorLocationInformation2 object to the client which contains the details of the points of interest contained in the region of interest.

Figure 5. Screenshot of the NECTISE Software Demonstrator

The advantage of using the NSD is that we have knowledge of how the system has been implemented. This information is valuable as it allows us to verify that the evaluation framework is affecting the behavior of the NSD. It is worth noting however, that from the perspective of the DBS-EF, the NSD is still considered a black box as it is only concerned with the inputs and outputs from the NSD.

B. NECTISE Test Cases

To demonstrate the DBS-EF in action, we have focused on Interaction Faults in the form of Invalid Requests between the client and the NSD, and also Interface Faults between the chosen concrete service and the DBS. Using these faults, we are able to see a clear difference in performance of the NSD in the presence of these types of fault.

To simulate invalid requests, we consider the makeup of the request as being two separate parts; the region of interest, and the `ServiceAvailability` object. As such, we have split the test cases into two different sub-cases – Invalid Request – i.e. the region of interest (ROI), and QoS faults – the `ServiceAvailability` object.

For the ROI test cases we partitioned the request parameters into the following instances: minimum value, maximum value, NaN (Not a Number), +/- Infinity. These instances refer to the values attached to the float parameters the NSD expects. Similarly, for the QoS fault, we applied a similar approach. This gave us a total of 41 different test cases with which to apply to the NSD.

To apply these faults, requests were formed and sent to the NSD to ascertain what behavior the NSD would exhibit in the presence of requests that were not valid.

To inject Interface Faults, we simulate a change of interface at the `SensorWS` to demonstrate the effectiveness of the DBS-to modify the behavior of a DBS under test. We do this by introducing a simulated change of method name for the SensorWS by simply replacing one character in the method name. We used SoapUI 4.5.0 to verify that if a request sent to a service does not conform to the interface, we receive a `SOAPFault` back from the service. In this instance, we found that, the request would not be recognized by the service. The response returned from the NSD when supplied an invalid method name is given in Figure 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
 <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-
envelope">
  <faultcode>S:Client</faultcode>
  <faultstring>
    Cannot find dispatch method for
{http://sensor.nectise.comp.leeds.ac.uk/}#etFeatures
  </faultstring>
 </S:Fault>
</S:Body>
</S:Envelope>
```

Figure 6. Example SOAP Fault (Interface Fault) response from the NECTISE Software Demonstrator with the key change in bold text.

The next stage was to develop a handler that would simulate this type of fault by replacing the regular response from the service, with a `SOAPFault` stating that the dispatch method was not recognized.

C. *Results and Analysis*

To illustrate the effectiveness of our approach, we made 1000 requests to the NSD without any simulated faults in order to get a baseline for our experiments.

Table II shows values that are representative of those within the domain of valid values for the context of the NSD. The aim of these baseline runs was to see the typical output of the system without the presence of any faults and to demonstrate the dynamic nature of the NSD. Our experiments were run on an Apple MacBook Air, 1.8 GHz Core i7 processor with 4GB RAM. Each run of 1000 experiments was performed five times in order to demonstrate the repeatability of the experiments, and also to illustrate the dynamic nature of the NSD.

TABLE II. REQUEST PARAMETERS

MonitorLocation2		**ServiceAvailability**	
x1	0.0	maxResponseTime	5.0
y1	50.0		
x2	50.0	availability	99.0
y2	0.0		

The results of these runs are shown in Figures 7, 8(i) and 9. What these results show is that for the specified region of interest, the NSD returned between one and five Points of Interest (POI) for each request, per run. In particular, for the five runs, the NSD returns the number of POI with a variability of 18.2724% between each run. What was particularly insteresting was that when we considered the distribution of POI returned, we noticed that on average, the majority (51.36%) of requests returned only one POI.

Whilst this data demonstrates that the NSD provides dynamic behavior between each batch of requests, we noticed that there was a definite trend that is mirrored in each test run. This trend can be seen in Figures 7 and 9. The explanation for this trend is that the NSD uses fixed data with respect to each of the sensors and what POI they can see. As a result, repeated runs show a similar trend in the results produced as this data is replayed with each successive run. Despite this, there is enough variability to demonstrate dynamic behavior to give us a useful baseline with which to compare the output of the NSD when the DBS-EF is applied.

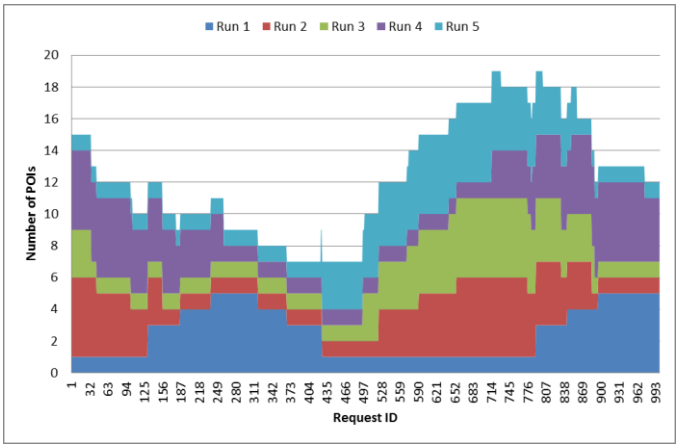

Figure 7. Number of POI returned per request

Our first set of tests concerned Invalid Requests. For these tests, each request was sent to the NSD 1000 times and the results recorded.

We observed that the output of the NSD fell into three separate categories. The first category included results from the NSD which included data on the POI within the ROI despite being given erroneous QoS parameters. In this category, the NSD returned differing numbers of POIs from the baseline. These results are shown in Figure 8(ii). Subsequent Invalid Request tests returned zero POI.

These results indicate that the NSD is not checking the QoS values for errors as it was possible to supply invalid data and still get a response from the NSD.

The second category returned no data from the NSD when the ROI parameters were set to +/- infinity or NaN. Following on from the previous category, given the NSDs insensitivity to QoS faults, we observed that the NSD is sensitive to invalid ROI values, but fails to return any valid error messages to the client. The only error we were able to observe was an `EJBException` which gave no indication as to the cause of the exception.

The final category returned POI data that contained zero POI. This only occurred when the ROI data was set to min/max float values (i.e. the ROI was way beyond the boundaries expected by the NSD). In this instance the NSD exhibits behavior that is erroneous as it still attempts to return POI information for an ROI that does not exist in terms of the NSD's specification.

We also analyzed the run times for the three categories. In category 1, the runtimes were between 136.502 seconds and 157.018 seconds. Category 2 exhibited shorter runtimes - between 96.404 seconds and 104.975 seconds - as the sensor service was not called. Category 3 exhibited the longest runtimes – between 68.36485 minutes and 68.6256 minutes – as the NSD was searching an extremely large ROI.

For the second batch of test runs, we introduced the simulated interface faults. For these test runs, we ran each test in five runs of 1000 requests due to a smaller set of tests. In each run, we noticed a significant reduction in performance of the NSD similar to the performance degradation observed in the Invalid Request test cases. In this instance, whereas the baseline runs took approximately two minutes to complete 1000 requests, when we introduced the Interface Faults, we found that each test run took approximately 69 minutes to complete.

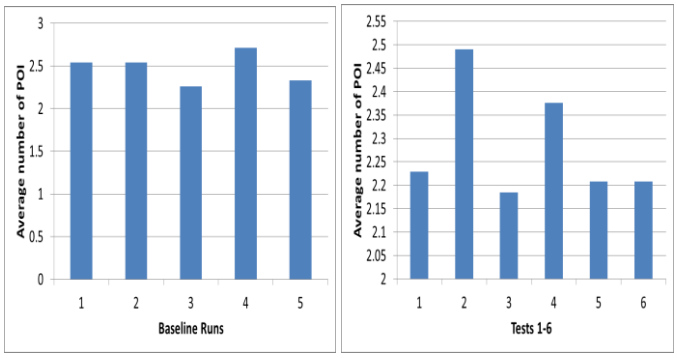

Figure 8. (i) Baseline average number of POI per test run.
(ii) Average number of POI per test (Test 1-6)

The output from the NSD showed that in the presence of Interface Faults, the NSD still returned a `MonitorLocationInformation2` object, but each object contained zero POI. This output from our experiments demonstrates that our DBS-EF triggered successfully the abnormal behavior of the NSD as expected such that we are able to observe how severely the system was affected, e.g. how severely the response time of the system was impeded, and the output from the system, e.g. with or without any POI information to the client and any exception information.

In the context of the dependability of the NSD, the system relies on a single gateway Web service which represents a single point of failure. Hence, by introducing the Interface Fault at the service, we are able to prevent the system from functioning as per the specification. It is interesting to note that while this represents a deviation from normal behavior, the NSD returns no information to the client that indicates that a problem was encountered whilst attempting to access the `SensorWS`.

Additionally, the performance penalties exhibited in the presence of Invalid Requests, could be avoided were the NSD to error check the values of the parameters prior to submission to the DBS.

It is this kind information that provides vital and valuable information for the system designer as in addition to suggesting a fault-tolerant and robust strategy for service instances, to prevent a single point of failure, it also suggests that they might wish to include appropriate error handling that will tolerate a change in interface, or develop a strategy for mediating between interface versions. Tolerating this type of fault is desirable in the context of the dynamic nature of the environment, as it may not be possible to know the interface of services *a priori* [7].

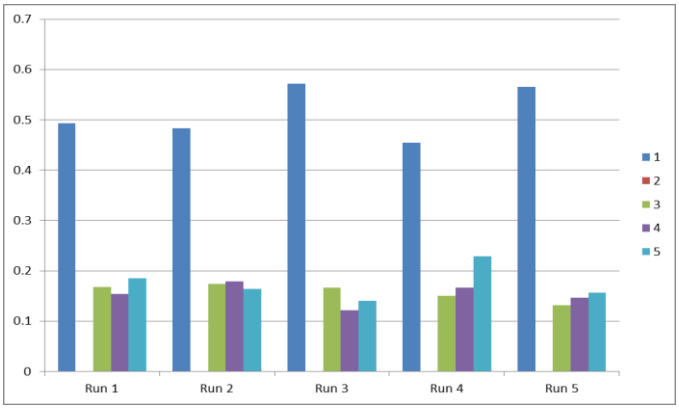

Figure 9. Distribution of number of POI returned per baseline run

VI. Conclusions and Future Work

In this paper we have discussed dynamic binding in SOC and the need to be able to assess the dependability of mechanisms that enable the binding of services at runtime. Previous work in this area does not consider the dependability of dynamic binding, rather choosing to focus on the correctness of the implementation. We have developed a novel evaluation framework that enables the evaluation of a DBS under test by considering the types of fault that can affect a DBS. These faults are taken from fault and system models derived from existing literature and presented in this paper.

The Dynamic Binding System Evaluation Framework (DBS-EF) has the ability to trigger abnormal behavior of a DBS as expected, due to the certain type of fault injected. More importantly, the behavior triggered by the DBS-EF provides vital information about the levels of severity of different types of fault, and how the system reacts to certain types of fault. Such information can offer hints on how the system under test could be improved further. For example, when an Interface Fault is considered, the returned response from the system can be enhanced with a properly detailed message for exception handling.

We have demonstrated the effectiveness of our approach by applying the DBS-EF to an existing DBS implementation in the form of the NECTISE Software Demonstrator (NSD). We have shown that by using the DBS-EF, we are able to simulate Interaction faults by injecting Invalid Requests and Interface Faults into the system. In turn, we then affect the behavior of

the NSD. Our results show that the response time of the NSD is greatly affected by those types of fault and that in the case of both types of Interaction Fault, each response returned erroneous results to the client. Furthermore, in some cases the NSD exhibited a dramatic degradation in performance.

In addition to this, there is no information at all in the response to suggest that either fault is the cause of behavior. Our analysis concludes that because the NSD uses a single gateway service to access sensor information, it suffers from a single point of failure. Additionally, we conclude that the NSD is not sensitive to invalid data with respect to the QoS parameters, or extreme values for ROI parameters.

Our work is not without its limitations – because we are using a 'black box' approach to testing a DBS, we are unable to exactly pinpoint which parts of the system are affected by the introduction of faults. The experiments presented in this paper focus on demonstrating the working of the DBS-EF by injecting Interaction Faults and do not give a full picture of the dependability of the NSD. Our ongoing work considers the injection of Communication Faults into the NSD.

The use of the NSD, whilst providing a real-life DBS to use to evaluate our framework also has limitations. The authors of NSD used fixed sensor data to model the sensors abilities and POI that they can see. This means that repeated runs showed similar trends with a variability of 18.2724% between each run. We have also used our knowledge of the NSD to verify that the DBS-EF is having an effect on the performance of the NSD in the presence of Interaction Faults. However, we note that the DBS-EF operates on the NSD in a strictly black box fashion. We also note that there is nothing to prevent the tester from combining the output of the DBS-EF with further information on the internal structure of a DBS if visible and additional White-Box testing if applicable.

Our future work will focus on the injection of faults arising from the monitoring of QoS attributes at runtime. We also aim to put all of our experiments into a larger suite of test campaigns in order to provide a larger test suite that can be applied to any DBS such that improvements to the dependability to a DBS under test can be made.

ACKNOWLEDGMENTS

The work in this paper has been supported in part by the National Basic Research Program of China (973) (No. 2011CB302602), the UK EPSRC WRG platform project (No. EP/F057644/1), and the UK EPSRC NECTISE program (No. EP/D505461/1).

REFERENCES

[1] Vuković, M., E. Kotsovinos, and P. Robinson, *An architecture for rapid, on-demand service composition.* Service Oriented Computing and Applications, 2007. **1**(4): p. 197-212.

[2] Papazoglou, M.P. and W.J. van den Heuvel, *Service oriented architectures: approaches, technologies and research issues.* The VLDB Journal The International Journal on Very Large Data Bases, 2007. **16**(3): p. 389-415.

[3] Callaway, R.D., et al., *An Autonomic Service Delivery Platform for Service-Oriented Network Environments.* Services Computing, IEEE Transactions on, 2010. **3**(2): p. 104 -115.

[4] Mabrouk, N.B., et al., *QoS-aware service composition in dynamic service oriented environments*, in *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. 2009, Springer-Verlag New York, Inc.: Urbanna, Illinois. p. 1-20.

[5] Erradi, A. and P. Maheshwari, *Dynamic binding framework for adaptive web services*, in *Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*. 2008. p. 162-167.

[6] Di Penta, M., et al., *WS Binder: a framework to enable dynamic binding of composite web services*, in *SOSE '06: Proceedings of the 2006 international workshop on Service-oriented software engineering*. 2006, ACM: Shanghai, China. p. 74-80.

[7] Cavallaro, L. and E. Di Nitto, *An approach to adapt service requests to actual service interfaces*, in *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. 2008, ACM: Leipzig, Germany. p. 129-136.

[8] Maximilien, E.M. and M.P. Singh, *A framework and ontology for dynamic web services selection.* IEEE Internet Computing, 2004. **8**(5): p. 84-93.

[9] Zheng, Z. and M.R. Lyu, *Collaborative Reliability Prediction for Service-Oriented Systems*, in *Proc. IEEE/ACM 32nd Int'l Conf. Software Engineering (ICSE'10)*. 2010. p. 35-44.

[10] Looker, N., *Dependability Analysis of Web Services*. 2006, Durham University.

[11] Brüning, S., S. Weissleder, and M. Malek, *A Fault Taxonomy for Service-Oriented Architecture*, in *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*. 2007. p. 367-368.

[12] Jhumka, A., *Dependability in Service-Oriented Computing*, in *Agent-Based Service-Oriented Computing*. 2010, Springer London. p. 141-160.

[13] Webster, D., et al., *Migrating Legacy Assets through SOA to Realize Network Enabled Capability*, in *New Directions in Web Data Management 1*, A. Vakali and L. Jain, Editors. 2011, Springer Berlin Heidelberg. p. 311-346.

[14] Châtel, P., J. Malenfant, and I. Truck, *QoS-based Late-Binding of Service Invocations in Adaptive Business Processes*, in *Web Services (ICWS), 2010 IEEE International Conference on*. 2010. p. 227 -234.

[15] Avizienis, A., et al., *Basic concepts and taxonomy of dependable and secure computing.* Dependable and Secure Computing, IEEE Transactions on, 2004. **1**(1): p. 11-33.

[16] Chan, K., et al., *A Fault Taxonomy for Web Service Composition*, in *Service-Oriented Computing - ICSOC 2007 Workshops*. 2009, Springer Berlin / Heidelberg. p. 363-375.

[17] Sargeant, A, P. Townend, J. Xu and K. Djemame, *Evaluating the Dependability of Dynamic Binding in Web Services*, in *14th IEEE International Symposium on High Assurance Systems Engineering (HASE)*. 2012, IEEE: Omaha, Nebraska. p. 139-146.

[18] Oracle. *SOAPException (Java EE 5 SDK)*. 2012 [cited 2012 18/11/2012]; Available from: http://glassfish.java.net/nonav/javaee5/api/javax/xml/soap/SOAPException.html.

[19] Myers, G.J., *The Art of Software Testing*. 1979, New York: John Wiley & Sons.

[20] Roper, M., *Software Testing*. 1994, Maidenhead: McGraw-Hill.

[21] Looker, N., M. Munro, and J. Xu, *WS-FIT: A tool for dependability analysis of web services*, in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. 2004.

[22] Farj, K., C. Yuhui, and N.A. Speirs. *A Fault Injection Method for Testing Dependable Web Service Systems*. in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*. 2012.

[23] Russell, D., et al. *Service-Oriented Integration of Systems for Military Capability*. in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. 2008.