
Proceedings Paper:
Sargeant, AJ, Townend, PM, Xu, J et al. (2012) Evaluating the Dependability of Dynamic Binding in Web Services. In: High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on. 14th IEEE International Symposium on High Assurance Systems Engineering (HASE), 25-27 Oct 2012, Omaha, NE, USA. IEEE, 139 - 146 . ISBN: 978-1-4673-4742-6. ISSN: 1530-2059.

https://doi.org/10.1109/HASE.2012.28

Evaluating the Dependability of Dynamic Binding in Web Services

Anthony Sargeant, Paul Townend, Jie Xu, 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 are 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. What is clear from existing research is that there exist several components that help to provide the necessary behavior for dynamic binding. However, the focus of these works is on the evaluation of the implementation of dynamic binding and does not consider an evaluation of dynamic binding systems themselves. To remedy this, we propose new system and fault models for Dynamic Binding in SOC that incorporate the types of components required for a Dynamic Binding System (DBS) and the types of fault that can affect these components. In addition to these models, we introduce a novel evaluation framework for the testing of a DBS. This distributed and extensible framework treats the DBS as a black box and hence is not restricted to the implementing technologies of the DBS. Finally we present the results of a series of experiments, which focus on the interactions between a client and the DBS. We discuss what these results reveal about the DBS under test and how they illustrate the value of our evaluation framework.**

Keywords-Service-oriented computing, Service-oriented architectures, Web Services, testing, dependability, evaluation tool.

I. INTRODUCTION

Service-Oriented Computing (SOC) provides a flexible framework in which applications are built up from services, often distributed across a network [1]. By loosely coupling service consumers to well-defined interfaces, and by using standardized intercommunication methods, we are able to create complex applications through the aggregation of one or more services. Service-Oriented Architectures (SOA) provides a logical architecture for the creation of these service-based applications [2]. Furthermore, SOC and SOA provides a framework in which the choice of service is agile to the needs of the service consumer, such that services can be swapped out for functionally equivalent services [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 consider certain aspects such as dynamic service composition [4, 5], how to best match requests to services [6, 7], dynamic service discovery [8], or dynamic reconfiguring of services [9]. Other works propose frameworks for the implementation of context-aware dynamic binding such as in ubiquitous computing environments [1]. What is clear from these works is that there exist several components that help to provide the necessary behavior for dynamic binding.

With existing work, the focus is 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. This represents a gap in the literature as the binding algorithm itself and/or its subsequent implementation could be subject to faults, and the behavior of the dynamic binding algorithms could be non-deterministic [10]. In addition to this, and to the best of our knowledge, the existing fault model for SOC as proposed by [11, 12] does not cover the kinds of failure modes that can affect a dynamic binding system.

The contributions of this paper are as follows: we collate the contributions of existing work to identify and unify the components that enable dynamic binding, into a single reference model for dynamic binding for SOC. Next, we consider the impact of dynamic binding on existing fault models for SOC and extend them so that they incorporate dynamic binding behavior. Using these models, we generate a series of test cases that is combined with a modular DBS testing framework to create a test suite to test the interaction between a client and a DBS. Finally, through a large number of experiments, we demonstrate the effectiveness of our evaluation framework with a large number of empirical results.

The rest of this paper is organized as follows: Section 2 discusses Dynamic Binding in SOC and frames it within the context of existing research in dependability and SOC. Section 3 discusses the system and fault models of dynamic binding in SOC, including all the necessary components and the process by which messages are bound. Section 4 describes the Evaluation Framework and its implementation. We also detail the test cases employed to evaluate a DBS under test. Section 5 introduces a proof of concept Dynamic Binding System that is then put under test using an implementation of the Evaluation Framework. Subsequent analysis and discussion of the results is also included. Section 6 evaluates this work through comparison with existing works in dynamic binding, service-oriented testing and dependability of SOC. Finally; Section 7 summarizes the paper and gives conclusions and a pointer to future work.

II. DYNAMIC BINDING IN SOC

SOA and SOC offers loose coupling of services, and a flexible infrastructure in which services need not be bound to service instances at design time [6]. In fact, it is desirable to leverage these characteristics by searching out services that are functionally equivalent and selecting from them, the 'best' service at runtime. Services that are found to match the consumer's request are considered as candidate services [4, 5, 7].

In order to ascertain what the 'best' service is, we must look to nonfunctional requirements [12]. This is often done using a separate framework for describing nonfunctional requirements – often referred to as Quality of Service (QoS) requirements [4, 5] – as interface definitions, such as the Web Services Description Language (WSDL), do not describe nonfunctional properties [12].

The final aspect to ascertaining the 'best' service is that of context. In [1], the authors discuss dynamic behavior in ubiquitous computing environments. They note that one additional factor is context-awareness. To illustrate this, they consider a person who wants directions to a local restaurant. If they are planning to walk, then their request to a navigation service can displayed on a smartphone. However, if they decide to drive to the restaurant, then an additional text-to-speech service is required. This change in context requires a change in service composition and represents a change from the original request.

As we can see, dynamic binding gives us flexibility in the way clients can consume services. Provided that a consumer's request is correctly formatted to a given interface, then it is not necessary to bind to a specific concrete instance at design time. However, this is not without its challenges.

When we make the comparison with static, or design-time binding, we note that there is a one-to-one relationship between a consumer and provider. The consumer will generate a client based on the provider's service interface at design time that will ensure that any compatibility issues can be dealt with in advance. However, should the provider's service be unavailable at runtime, then it is necessary for the consumer to find a new service, generate a new client for the service interface and then invoke that service. It is clear that what we gain in terms ease of integration, we lose in terms of flexibility.

By contrast, with dynamic binding we can select from a number of services that are functionally equivalent. In the event of a service being unavailable, then it is possible to invoke a replacement service that has similar functionality. Here we have improved the dependability of our system as the probability of getting a response is increased through redundancy [13].

If we assume that all functionally equivalent services employ a standard interface [13], then such a system will be easy to implement. However, Cavallaro et al in [7] state that is not a realistic assumption as service providers will develop services independently. Consequently, whilst two services may offer the same functionality, their interfaces may differ in some way. What we can see is that dynamic binding gives us added flexibility, but at the expense of added complexity of dealing with interface mismatches.

It is clear that there are several steps that need to be addressed in order to realize dynamic binding and it is important that the DBS must deal with faults at each of these steps. As such, we will now identify the components needed that help to realize the system model for Dynamic Binding in SOA.

III. SYSTEM AND FAULT MODELS

Existing literature discuses dynamic binding in SOC by suggesting different components that would be needed in order to achieve the required dynamic behavior. To the best our knowledge, a common theme is the algorithm by which dynamic binding takes place. This algorithm is described alongside the system model below. In order to compile a reference system model, we bring together these components from existing research into a unified system model for dynamic binding in SOC.

A. System Model for Dynamic Binding in SOC

The system model for Dynamic Binding in SOC consists of the following components and is illustrated in figure 1:

- **Request Processing**: This component receives abstract requests from a consumer and processes it to ascertain the functional and nonfunctional requirements of the consumer. This information is used to determine the best service to meet a consumer s request [8].
- **Service Discovery**: The aim of this component is to find candidate services that will meet the consumer's functional requirements. As discussed, services will publish their interfaces online and might be stored in a registry. Discovering a service then is a simple case of searching the registry.
- **Service Selection**: In order to select the best service, it is necessary to ascertain which of the candidate services meets, or exceeds the minimum nonfunctional requirements [14]. This might be selecting the cheapest service, or the quickest service by ranking them in order of price or response time.
- **Service Integration**: Where static binding of services is employed, ensuring the client conforms to the service interface is a trivial exercise as this can be done at design time. However, when the exact service interface is not know a priori, then it is important that some form of mediation is employed to ensure that the client request meets that what is expected by the service [7]. Similarly, it is necessary to employ interface mediation if the format of the service response does not match the response as expected by the client.
- **Context Monitoring**: When a request is initially sent, it will be sent with a given context. However, 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 to allow the selection of the most

appropriate service to meet the request, given the context change [14].

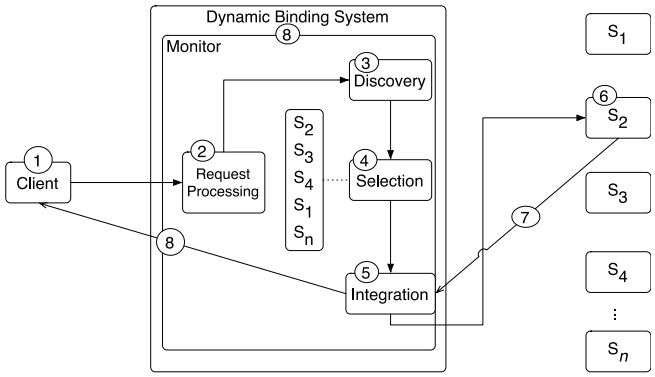

Figure 1. System Model of Dynamic Binding in SOC.

Figure 1 depicts the system model for a Dynamic Binding System (DBS) where the following algorithm is employed:

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 requests minimum nonfunctional requirements [4, 14].
5. The integration mechanism ensures that the consumer's abstract request is interoperable with concrete service interface.
6. After integration phase, the request is then passed to the concrete service instance.
7. The response from the provider is then 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 included so that the current context of the consumer's request is maintained. If there is a change, then the monitor can ensure the system adapts as necessary.

B. Fault Model for Dynamic Binding in SOC

Now that we have described the system model, in order to evaluate a DBS from a functional perspective the next step is describing the types of fault that can affect the DBS.

The fault model is based on the work of Avizienis et al. in [15] and Chan et al. in [16]. In their seminal work, Avizienis et al. present a comprehensive taxonomy that seeks to capture the classes and categories of faults that can affect software systems. Similarly, Chan et al. take this approach and apply it to Web Services to present a fault taxonomy for web services. In this paper, we further extend these works to include the dynamic binding of services. The fault model is listed below:

- **Request Processing Faults**: Processing a client's request is important as it ascertains the needs of the consumer. However, if the request has been poorly formatted, or if the request falls outside of the system scope, then this component may fail. In this case, the request would be considered to be not complete [17]. Alternatively a Network Time-out could occur between the client and the DBS.
- **Discovery Faults**: Although how service discovery is implemented is out of scope for this paper, it is still important to understand the faults that could occur. Correct service discovery is key to the successful use of dynamic binding as if no service exists, or the wrong search parameters are used, then the system will fail [12]. Typical faults include 'Service does not exist' and 'Network Time-out'.
- **Selection Faults**: Selection faults occur when the dynamic binding system selects the wrong service, or there is no suitable candidate service available. Typical faults include 'Invalid Selection Criteria' [8], and 'No Suitable Service' where there is no service that meets the minimum nonfunctional criteria [4].
- **Integration Faults**: Integration faults occur as a result of one or more services not being interoperable and where the mediation between the interfaces and/or protocols is not possible. This might include 'Interface Mismatch', 'Incorrect Response' and 'Dependency' faults [7].
- **Context Faults**: Context faults occur when either the current context is not reported correctly, or that the timing of selecting the 'best' service is incorrect. Faults that occur here would include 'Timing' [4] and 'Monitoring' faults [14]

IV. EVALUATION FRAMEWORK (DBS-EF)

In order to evaluate a dynamic binding system, we have developed a framework for the testing of a DBS implementation. Our framework allows a DBS to be placed within a controlled environment that is able to manage all messages into, and out from the DBS. We then use the system and fault models to guide the selection and placement of faults. By inducing failures at one or more of the components in the system model, we can observe the behavior of the DBS by monitoring the outputs of the system. This black-box approach means that we are not dependent on the implementation details of the DBS, in order to introduce faults and we are able to concentrate more on the behavior of the DBS in the presence of faults.

A. DBS-EF Implementation

The first component of this framework is the instrumented service. As we are focused on the testing of a DBS, and in order to provide the DBS with input that is faulty, we utilize handlers in each service to inject appropriate faults based on the test cases. These handlers connect to a Fault Coordination Service (FCS) that controls the time and location of faults into the DBS. This Fault Coordination Service uses a test campaign that is supplied via the user prior to the running of the Evaluation Framework. This allows for a flexible framework that can be tailored by the user to include user-defined failure modes as the application demands.

The client is the entry point for the system and can also supply faults to the system by sending invalid requests to the DBS. The client is also responsible for collating the output from the FIS, and the handlers to supply a summary of the results to the user.

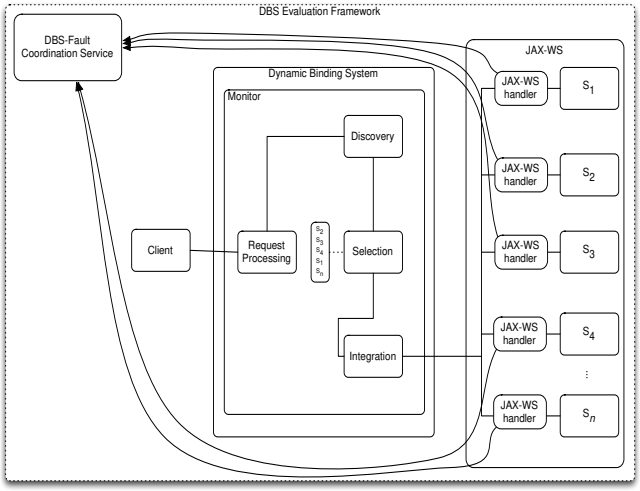

Figure 2. Evaluation Framework for SOC that utilizes Dynamic Binding

We have developed the implementation of our framework using Web Services and SOAP. The implementing technologies used are Java and Glassfish v3.1, which uses JAX-WS for sending and receiving SOAP messages. JAX-WS also allows the incorporation of handlers that enable access to the SOAP messages being passed between the DBS and service/client of the DBS.

The key benefit of our approach is that we are not dependent on how the DBS is implemented. SOA relies on standardized communication methods via defined interfaces to allow a distributed computing environment. By taking these key tenets of SOA, and by modularizing the evaluation framework, we have ensured that the system can be applied to any DBS system irrespective of its implementation. Furthermore, our approach is a 'black-box' such that it is not necessary to know the exact implementation of the DBS in order to evaluate it.

B. Test Cases

One of the key features of the Evaluation Framework is to design and implement user-specified test cases. Here the user can state the type of test, the expected output and the number of times the test is to be run. This allows for an extensible framework whereby the user is not limited to predetermined outcomes, but can state what the DBS should return to the user in the presence of a particular fault.

In our work, test cases for the evaluation framework was based on the fault model, and targeted the individual components of the system model. The purpose of these test cases was to see how faults in each of these components affected the behavior of a DBS under test. By selecting the types of test and through comparison of the expected output with the actual output, we would be able to ascertain whether or not a DBS can tolerate faults.

For this paper, as an illustration of how this technique will aid the evaluation of a DBS we have concentrated on the interactions between the client and the DBS only. Here, we focus on faults relating to invalid requests. A request to the DBS consists of two parts; the request itself and a QoS attribute and value. As such, we have split the test cases into two different sub-cases – Invalid Request, and QoS faults.

C. Invalid Request Fault

An invalid request fault has the following test cases: *Missing Token(s)*, *Incorrect Parameter Types*, *Parameter Mismatch*, *Empty Request*, *Zero Parameters*, and *Empty Parameters*. Relating to this are the expected outputs from the DBS when faced with the above faults. Those outputs are: *Invalid Request Exception* (specific error handler response), and *No Service Available*. In addition to this, there are also outputs relating to unexpected outputs, i.e. indicators that the DBS cannot tolerate the fault being introduced. In this case, the outputs are:

- **Service Response**: the DBS binds to a service and returns a response when an exception should have been raised.
- **Binding Fault**: as the name suggests refers to a service has been bound to the request erroneously.
- **Unexpected Exception**: an exception is raised that does not correspond to the expected output.
- **No Response**: the DBS does not return anything indicating either a failure in the network, or a crash of the hardware, the hosting environment or the DBS itself.

Table I gives the range of test cases that were employed.

D. QoS Faults

QoS faults has the following test cases: Unrecognized QoS Attribute, Missing Attribute, Missing Value, Negative Value, Max Boundary Value + 1, Min Boundary Value - 1, Invalid Value Type. The expected outputs from these faults are QoS Exception (specific error handler response), Binding Exception, Unexpected Error, and No Response.

In table I, we specify that test cases for an Invalid Request should return the following:

- **No Service Available**: here the DBS cannot find a service that can meet the client request
- **Invalid Request Exception**: here the DBS returns a meaningful error back to the client that states that the

request cannot be fulfilled due to an invalid request being passed to the DBS.

- **Service Response**: here the DBS returns a response from the concrete service chosen by the DBS.

The remaining test cases relate to the behavior of services to the DBS. As we are focused on the interaction between the client and the DBS, these test cases are out of scope.

TABLE I. INVALID REQUEST TEST CASES

Test Cases	Input	Expected Output
Missing Tokens	add(,3)	Invalid Request Exception
Missing Tokens	add(1,)	Invalid Request Exception
Missing Tokens	add(1,3	Invalid Request Exception
Missing Tokens	dd(1,3)	No Service Available
Incorrect Parameter Types	add("one",3)	Invalid Request Exception
Incorrect Parameter Types	add(1,"three")	Invalid Request Exception
Parameter Mismatch	add(1)	No Service Available
Parameter Mismatch	add(1,3,3)	No Service Available
Parameter Mismatch	add(3,1)	Service Response
Empty Method Call	""	Invalid Request Exception
Zero Parameters	add()	No Service Available
Empty Parameters	add(null,null)	Invalid Request Exception
Misbehaving Service(s)	-	-
Slow Service	-	-
Unresponsive Service	-	-
Incorrect Results	-	-
Binding Failure	-	-
Unavailable Service	-	-
No Service Available	-	-

E. Output from the DBS-EF

As mentioned previously, the DBS-EF will output a summary of the results of the test cases. In order to marry up the test case results, we record details in log files of the input, output and any faults that have been introduced. The log files also indicate which of the DBS-EF services were called, and the response they supplied back to the DBS. The client also records the response as received from the DBS.

A sample of the output is supplied below:

```
08-02092: QoS Exception; 5;
2092,2012-07-18 19:58:20.088, No Fault,
http://addws.scs5ajs.leeds.ac.uk/
```

The above entry shows the following: Test Case ID and test run ID, expected output from the DBS, actual output from the DBS and finally an entry from the DBS-FCS log. The DBS-FCS log states the following: test run ID, timestamp, type of fault to be injected, and the URI of the web service being invoked.

V. DBS-EF EXPERIMENTS AND RESULTS

We have developed a specimen Dynamic Binding System based around an online distributed calculator as shown in Figure 3 in order to test the effectiveness of our framework. Here the DBS would field requests and then choose from six web services. Two web services (CalculatorWS and CalcWS) offered a full range of methods: add, subtract, multiply and divide. One web service offered multiplication only (MultiplicationWS) and three offered addition only (AdditionWS, AddWS1, AddWS2). Each of these services were provided with an advertised QoS attribute of *Availability* which was set and fixed for the duration of the experiments. As we were testing the interaction between the client and the DBS, none of the services were instructed to behave erroneously and consequently assumed to be fault free.

In order to obtain meaningful results, each test case was run 100 times on an Apple MacBook Air, with a 1.8GHz Core i7 processor with 4GB RAM. Furthermore, we used soapUI 4.5.0 to verify requests to confirm our observations in certain cases where we felt the results needed further investigation.

Of the two types of test, there were a total of 12 different test cases for invalid requests, and 9 different test cases for invalid QoS. It is worth noting that these are not exhaustive test cases, and only represent some of the major types of fault to be encountered. Our work is not to provide an exhaustive suite of test cases, but to provide a modular, extensible framework such that new test cases can be appended to the suite of existing test cases.

As each individual test case was run 100 times, this gave us a total of [illegible] test case runs.

1) Invalid Request Test Results

From the results of the experiments involving invalid requests, we observed that the tests were repeatable and would return the same results for each set of test cases.

We also observed that only 16.67% of test cases returned a result that was expected, 83.33% returned a result that was different to the expected output, with no services failing to return a response.

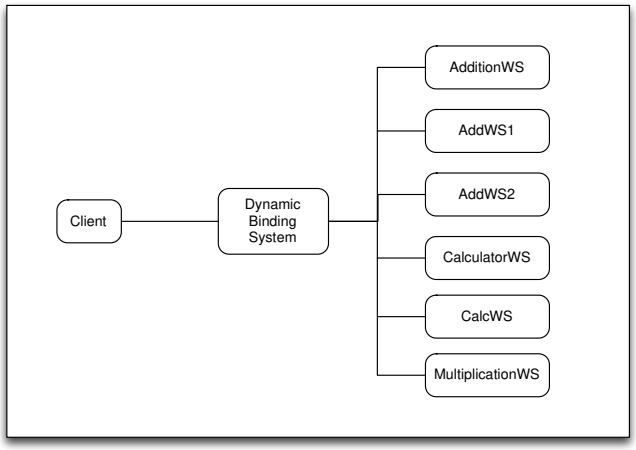

Figure 3. Experimental Dynamic System

Further analysis showed that 25% of tests returned a NullPointerException instead of an Invalid Request Exception. 16.67% of tests returned a NullPointerException instead of matching against services' functional requirements. 8.33% of tests returned a response that was correct from the called service (i.e., 3 + 1 = 4), instead of an Invalid Request Exception. 8.33% of tests returned an IndexOutOfBoundsException due to there being too few parameters instead of not matching against the functional properties of the service, or mediating the request properly.

25% returned a response that was incorrect from the service (i.e., 3 + 1 ≠ 4).

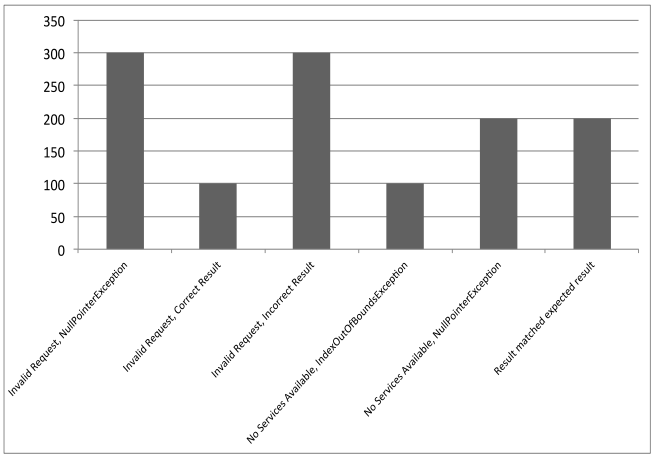

Figure 4. Breakdown of Invalid Request test case results.

What the framework tells us is that the DBS under test does not provide meaningful error messages back to the user in the event of an invalid request. What we note in particular is the tendency for the DBS to process requests where parameters are not of a valid type. For example, if the request was: `add(1, "three")`, the DBS returned the value `1`. This appeared to be an issue with JAX-WS, as supplying the above request via the soapUI tool showed that JAX-WS replaces all non-integer values with zero, such that the request becomes `add(1, 0)`. This was the same for the case where the request was `add("null", "null")`. However, it is still the responsibility of the DBS to mediate this kind of error such that either the string is converted to an integer where possible, or it returns an error message to the user.

2) Invalid QoS Test Results

From running the tests for invalid QoS requests, we observed that as before, the tests returned the same results on repeated runs. In 33.33% of cases, the returned result from the DBS matched the expected result in the test case. We subsequently considered these as passed tests. In 33.33% of cases the DBS returned a correct result from the service, when we expected an exception due to a poorly formatted QoS requirement. In 33.33% of test cases the DBS returned a response from the service which was correct as previous, when we expected there to be No Services Available to field our request. Lastly, no test case failed to receive a response from the DBS indicating that all test cases were serviced by the DBS.

As before, we used the soapUI framework to verify some of our observations. The one interesting observation was that if we supplied a QoS attribute that we knew to be unknown to the DBS (for example, a random string of text), then the DBS would still return a result if the value of the bogus attribute was less than the advertised values of the registered services with the DBS.

What we concluded from this is that the DBS does not properly check for the QoS attribute when considering candidate services.

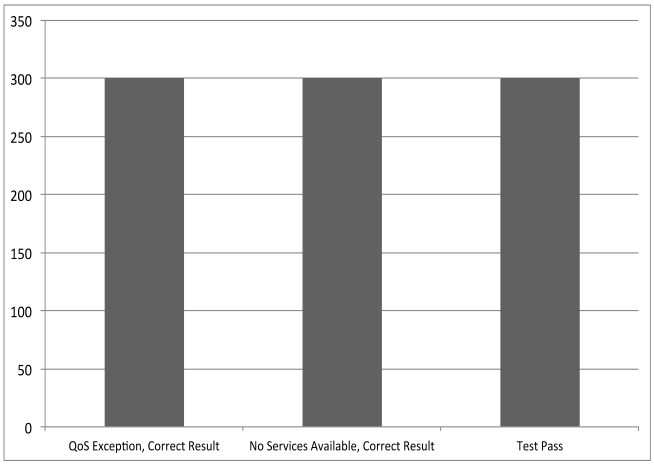

Figure 5. Breakdown of Invalid QoS test case results.

3) Observations

As mentioned, a total of 2100 tests were run, and when we combine both data sets, we see that 14.29% of tests returned the expected result with 85.71% of tests failing to return a result that matched the expected outcome.

From this we were able to conclude that the Request Processing (RP) component was unable to process the request appropriately but failed to return a meaningful error message to the user. Consequently, it would require the developer of the DBS to redesign the RP component in order to handle invalid input.

VI. Related Works

A. Comparison with Farj and Looker

Our work is similar to that of Nik Looker in [10] and Khaled Farj, et al. [18] in that we are using fault injection in a Service Oriented Environment. In Looker's work, the author is testing the dependability of web services by placing those services in an instrumented Apache Axis implementation. Whilst this approach afforded the tester a non-invasive approach for fault injection, it focuses solely on the testing of services. It also requires that the service is deployed on the instrumented Axis container, which limits the portability of the approach.

Farj in [18] extended Looker's approach by providing a framework based on web service proxies and employing network emulation in order to provide a real-world environment to test services. For each service instance, a proxy is generated and the fault injection campaign is employed by intercepting messages into, and out of each service under test. The network emulation then afforded a realistic distributed environment for the services. One advantage of this approach is that it was no longer required to have the service residing on an instrumented service container. However, this approach also focuses on the

testing of services only and does not consider dynamic binding.

Our work differs from both of these works as we are focused on testing the dependability of a Dynamic Binding System that brokers requests to concrete service instances and not the correctness (or absence thereof) of the concrete service instances. Similar to Looker and Farj, we adopt a fault injection approach as it does not require access to the source code of the DBS under test, and we have employed a distributed test environment so as to avoid being restricted in terms of the DBS implementation technologies.

Existing work in dynamic binding focuses on the development of techniques to enable dynamic behavior in SOC. For example, [6] utilizes dynamic service discovery at runtime for an abstract workflow. Additionally, [7] looks at techniques for matching abstract services to concrete service instances. Finally [14] looks at incorporating QoS attributes into dynamic service selection and invocation.

All these works follow a common algorithm as mentioned in section III. Where our work differs from the above works is that we are focused on the dependability of dynamic binding in SOC as opposed to identifying a new technique that enables dynamic behavior.

B. *Dependability in SOC*

Jhumka in [12] presents a high-level model for the dependability of SOC, however, this model is too abstract to be applied as a testing framework for Dynamic Binding in SOC.

Similar to Jhumka, Chan in [16] created a fault taxonomy for web services. This taxonomy whilst being more suited to our research, still failed to consider how dynamic binding affects the dependability of service-oriented architectures such as web services.

In order to be applicable to our work, we have sought to extend these works to incorporate dynamic binding in the system and fault models. These models have been used to provide a series of test cases that can be used to evaluate a DBS. Our results have shown that we are able to exercise the fault tolerance mechanisms (or absence thereof) of a DBS under test.

VII. SUMMARY

In this paper, we have shown that to the best of our knowledge, there exists a gap in present literature, as the dependability of dynamic binding is not considered. To remedy this, we have presented new system and fault models for dynamic binding in SOC.

Furthermore, we have presented a novel method for the testing and evaluation of a Dynamic Binding System (DBS). This DBS Evaluation Framework (DBS-EF) is based on research into the types of component needed to realize a DBS and the types of fault that can affect a DBS that would need to be tolerated.

The DBS-EF utilizes a series of test cases designed to exercise the components to determine the presence (or lack thereof) of fault tolerance mechanisms by injecting faults based on the fault model into messages sent into and from the DBS. This is achieved via the use of instrumented services, and a Fault Injection Service that coordinates the placement and timing of faults.

In this paper, we have tested the interactions between the client and the DBS via a suite of test cases based around the fault model. The results of our experiments have shed light on the implementation of the DBS under test, its fault tolerance mechanisms (or lack thereof) and also given insight into the workings of JAX-WS as a web services container.

A. *Future Work*

Presently our work concentrates on only on the interactions between the client and the DBS and so only one type of fault – Invalid Request. However, our future work will include interactions between the DBS and registered services, as well as the full suite of faults. We will also be dealing with multiple QoS attributes. Finally, we intend to plug-in a real DBS implementation in order to further validate our results.

ACKNOWLEDGMENT

The work reported in this paper has been supported in part by the NECTISE program jointly funded by BAE Systems and the U.K. EPSRC Grant EP/D505461/.

REFERENCES

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

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

[3] 3 Callaway, R.D., Devetsikiotis, M., Viniotis, Y., and Rodriguez, A.: 'An Autonomic Service Delivery Platform for Service-Oriented Network Environments', Services Computing, IEEE Transactions on, 2010, 3, (2), pp. 104 -115

[4] 4 Mabrouk, N.B., Beauche, S., Kuznetsova, E., Georgantas, N., and Issarny, V.e.: 'QoS-aware service composition in dynamic service oriented environments', in Editor (Ed.)^(Eds.): 'Book QoS-aware service composition in dynamic service oriented environments' (Springer-Verlag New York, Inc., 2009, edn.), pp. 1-20

[5] 5 Erradi, A., and Maheshwari, P.: 'Dynamic binding framework for adaptive web services', in Editor (Ed.)^(Eds.): 'Book Dynamic binding framework for adaptive web services' (2008, edn.), pp. 162-167

[6] 6 Di Penta, M., Esposito, R., Villani, M.L., Codato, R., Colombo, M., and Di Nitto, E.: 'WS Binder: a framework to enable dynamic binding of composite web services', in Editor (Ed.)^(Eds.): 'Book WS Binder: a framework to enable dynamic binding of composite web services' (ACM, 2006, edn.), pp. 74-80

[7] 7 Cavallaro, L., and Di Nitto, E.: 'An approach to adapt service requests to actual service interfaces', in Editor (Ed.)^(Eds.): 'Book An approach to adapt service requests to actual service interfaces' (ACM, 2008, edn.), pp. 129-136

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

[9] 9 Zheng, Z., and Lyu, M.R.: 'Collaborative Reliability Prediction for Service-Oriented Systems', in Editor (Ed.)^(Eds.): 'Book Collaborative Reliability Prediction for Service-Oriented Systems' (2010, edn.), pp. 35-44

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

[11] 11 Bruning, S., Weißleder, S., and Malek, M.: 'A fault taxonomy for service-oriented architecture', in Editor (Ed.)^(Eds.): 'Book A fault taxonomy for service-oriented architecture' (2007, edn.), pp. 367-368

[12] 12 Jhumka, A.: 'Dependability in Service-Oriented Computing', in Editor (Ed.)^(Eds.): 'Book Dependability in Service-Oriented Computing' (Springer London, 2010, edn.), pp. 141-160

[13] 13 Zheng, Z., and Lyu, M.R.: 'A Distributed Replication Strategy Evaluation and Selection Framework for Fault Tolerant Web Services', in Editor (Ed.)^(Eds.): 'Book A Distributed Replication Strategy Evaluation and Selection Framework for Fault Tolerant Web Services' (2008, edn.), pp. 145 -152

[14] 14 Châtel, P., Malenfant, J., and Truck, I.: 'QoS-based Late-Binding of Service Invocations in Adaptive Business Processes', in Editor (Ed.)^(Eds.): 'Book QoS-based Late-Binding of Service Invocations in Adaptive Business Processes' (2010, edn.), pp. 227 -234

[15] 15 Avizienis, A., Laprie, J.C., Randell, B., and Landwehr, C.: 'Basic concepts and taxonomy of dependable and secure computing', Dependable and Secure Computing, IEEE Transactions on, 2004, 1, (1), pp. 11-33

[16] 16 Chan, K., Bishop, J., Steyn, J., Baresi, L., and Guinea, S.: 'A Fault Taxonomy for Web Service Composition', in Editor (Ed.)^(Eds.): 'Book A Fault Taxonomy for Web Service Composition' (Springer Berlin / Heidelberg, 2009, edn.), pp. 363-375

[17] 17 Küster, U., and König-Ries, B.: 'Dynamic Binding for BPEL Processes - A Lightweight Approach to Integrate Semantics into Web Services', in Editor (Ed.)^(Eds.): 'Book Dynamic Binding for BPEL Processes - A Lightweight Approach to Integrate Semantics into Web Services' (Springer Berlin / Heidelberg, 2007, edn.), pp. 116-127

[18] 18 Farj, K., Yuhui, C., and Speirs, N.A.: 'A Fault Injection Method for Testing Dependable Web Service Systems', in Editor (Ed.)^(Eds.): 'Book A Fault Injection Method for Testing Dependable Web Service Systems' (2012, edn.), pp. 47-55