Pedagogic Data as a Basis for Web Service Fault Models

N. Looker, L. Burd, S. Drummond, J. Xu*, M. Munro
Department of Computer Science,
Durham University, UK
*School of Computing,
University of Leeds,
Leeds, UK

{n.e.looker,liz.burd,sarah.drummond,malcolm.munro}
@durham.ac.uk
jxu@comp.leeds.ac.uk

Abstract

This paper outlines our method for deriving Fault Models for use with our WS-FIT tool that can be used to assess the dependability of SOA. Since one of the major issues with extracting these heuristic rules and Fault Models is the availability of software systems we examine the use of systems constructed through pedagogic activities to provide one source of information.

1. Introduction

Software-Oriented Architectures (SOA) based upon Web Services are becoming the de-facto standard for both web based commercial applications as well as distributed scientific projects. Given the prominence of this technology, test methods and tools are required to ensure that robust systems are deployed. Testing is required not only to uncover existing problems with systems but to also provide potential users with metrics to compare similar serviced based solutions.

Standards like SOAP, UDDI, and WSDL [1] are being rapidly adopted by all major Web Service providers. All Web Services need to establish and adhere to standards so Quality of Service, and particularly reliability, will become an important differentiating point for services. This differentiation may become even more important in the future with technologies such as dynamic service composition being developed, with consumers having a choice of similar services.

The WS-FIT fault injection method [2] is a modified version of Network Level Fault Injection. WS-FIT differs from standard Network Level Fault Injection techniques in that the fault injector decodes the SOAP messages and can inject faults into individual RPC parameters, rather than randomly corrupting a message, for instance bit-flipping. This enables API-level parameter value modification to be performed in a non-invasive way as well as standard Network Level Fault Injection. This novel technique can be coupled with an ontology that is derived from a standard Fault Model. This ontology allows the automatic generation of test cases for any of the messages or RPC parameters in a SOA. A similar ontology is applied to failure modes to implement failure detection.

Currently the WS-FIT method is capable of automatically generating unit tests for individual Web Services. This paper outlines our plans for expanding the WS-FIT method, using heuristic rules, to allow automatic test campaign generation for a complete SOA. One of the major problems encountered in determining these rules is the requirement for independently implemented test cases, and we show how pedagogic data can be used as one of the sources of these systems.

2. Reliability Definitions

This section reviews and gives a consistent definition of field of Reliability since, particularly in the area of Quality of Service, there is variation amongst the meaning of some of the terms used. This work therefore attempts to give a standard definition of all relevant terms and they are taken to be definitive throughout the rest of this paper.

Quality of Service (QoS) [3] covers a whole range of factors that are combined to define the QoS offered by a system. QoS is commonly used in networking where it is defined by Steinmetz et al [4] as *"a concept for specifying how 'good' the offered networking services are. QoS can be characterized by a number of specific parameters."* But when applied to Web Services there is no universally agreed definition but the following factors are commonly used [5] and are taken as definitive when used in this work:

Availability: the quality aspect of whether a Web Service is present and ready for use. This is represented as the probability that a Web Service will be available at a specific time. This may be affected by such things as time to complete a pervious operation, loading on a particular service, etc.

Accessibility: the quality aspect that represents the degree the Web Service is capable of serving a request and a specific point in time. This is different from Availability since a service may be available but not

accessible. For instance the initial request can be accepted but it cannot be processed due to some other dependency, for instance it may depend on another unavailable service, so that request would be queued awaiting a response from the unavailable service. Accessibility can be improved by improving the scalability of a system.

Integrity: the quality of the Web Service maintaining the correctness of any interaction. If a transaction fails data should remain in a consistent state. This can be achieved through mechanisms such as distributed commit [6], rollback mechanisms [7], etc.

Performance: the quality aspect that is defined in terms of the throughput of a Web Service and the latency. Throughput is the number of requests serviced in a given period and the latency is the time taken to service a request. The aim is to produce a high throughput but low latency system. Throughput and latency can be affected by such factors as processor speed, code efficiency, network transfer time, etc.

Reliability: the quality aspect that represents the capability of maintaining the service and service quality. One measurement of reliability is the number of failures during a given period [8]. Another aspect of reliability refers to the probability that defines the dependability that a request is correctly sent and serviced.

Regulatory: the quality aspect that the service corresponds to rules, laws, standards and specifications. This can have an effect on areas such as availability, performance, and reliability through Service Level Agreements (SLA). SLA can define minimum levels of performance expected by a service that set levels for its dependability.

Security: the quality aspect that defines confidentiality for parties using a service. This can be influenced by regulatory factors. It can also affect performance due to the extra overhead incurred in implementing security mechanisms.

Certain factors can be quantitatively measured and others remain harder to quantify, for example Reliability can be measured by failures over time but the effectives of Regulatory can't be measured by simple means.

2.1. Dependability

The IFIP Working Group on Dependable Computing and Fault Tolerance [9] defines dependability as "*The notion of dependability, defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers, enables these various concerns to be subsumed within a single conceptual framework.*" A number of factors affect the dependability of a system [10] but for the purpose of our research we are interested in factors that can be used to assess dependability via fault injection [11]. A summary of the key factors follows.

Fault: A fault (which is usually referred to as a bug for historic reasons) is a defect in a system. The presence of a fault in a system may or may not lead to a failure, for instance although a system may contain a fault its input and state conditions may never cause this fault to exhibit as a failure. An analogy from engineering would be part of a metal bridge being susceptible to rust (this is the fault). The component may or may not rust depending on whether it comes into contact with water and air (its input conditions). If it is painted correctly and maintained it will never rust.

Error: An error is a discrepancy between the observed behavior of a system and its specified behavior. Errors occur at runtime when some part of the system enters an unexpected state. Since errors are generated from invalid states they are hard to observe without special mechanisms, such as debuggers or debug output to logs.

If the fault can be located it can be removed. In some cases a fault cannot be adequately verified (i.e. intermittent faults, timing faults, etc). It is important to note the difference between fault location and error detection. Error detection indicates the presence of an error somewhere within a system but it may only become apparent through interaction with another system thus moving the apparent location of the error to another system [12]. Fault location is the process of actually locating the physical fault. To go back to our bridge analogy, it would be analogous to a suspension cable becoming unfixed (this would be the error) due to a securing bolt not being of sufficient strength (this would be the fault).

Failure: A failure is an instance in time when a system displays behavior that is contrary to its specification. An error may not necessarily cause a failure, for instance an exception may be thrown by a system but may be caught and handled using fault tolerance techniques so the overall operation of the system will conform to the specification.

Mean Time Between Failure (MTBF): The average time expected to elapse between one failure occurring and the next failure. This time includes the time taken to rectify the fault. The time taken to rectify the problem could be relatively short, for instance the fault could be rectified by restarting the software or the failure could be handled by a fault tolerance system. The MTBF can also be relatively long, for instance a fault may have to be fixed in the system before execution can continue such as a database being restored from backup.

Mean Time To Recovery (MTTR): The time taken to repair and damage done by a failure and restore the software to a state where it can provide it's specified function.

Mean Time To Failure (MTTF): The average time expected to elapse before a failure occurs.

3. Fault Injection

Some dimensions are quantifiable by direct measurements whilst others are more subjective, for instance *Safety* cannot be measured directly via metrics but is a subjective assessment that gives a level of confidence where by *Reliability* can be quantified by physical measurements.

System dependability can be assessed using either model or measurement based techniques [13]. Both techniques have their merits. Modeling can be used in the design stage to predict potential errors and faults in algorithms. Measurement can be applied to existing systems to provide metrics on dependability.

Modeling can only make predictions of the dependability of a system since it is derived from system design, specification and code documents. Once a system has been implemented actual measurement techniques can be applied to obtain specific metrics and allow data on dependability to be derived from them.

Measurement techniques are useful because they can be applied to existing systems without requiring access to source code or design documentation. There are two main measurement techniques:

Observation [14]: measurements can be performed by the observation of errors and failures in a large set of deployed systems. This technique uses existing logs, either logs maintained by the system administrator or logs generated automatically by the system. Analysis of the data can obtain information on the frequency of faults and the activity that was in progress when they occurred. Since failures and errors may occur infrequently data must be collected over a long period of time and from a large number of systems. Even with this it is unlikely that this technique will catch rarely seen errors [15] since they may take a very long time to occur (in the order of years).

Fault Injection [11]: since the MTBF for a given failure may be very long since the statements containing the error may be executed infrequently, fault injection attempts to speed up this process by injecting faults into a running system in an attempt to cause the execution of seldom used control pathways within a system. By doing this either a failure will occur or the systems fault tolerance mechanism will handle the fault. Fault Injection can be implemented as Hardware Implemented Fault Injection (HWIFI) or Software Implemented Fault Injection (SWIFI).

Fault injection can be used to simulate unusual input conditions and exercise the boundaries between software components that would otherwise rely on being exercised by calls generated by other components in response to user input. Since this input will go through a number of intermediate steps it is extremely unlikely that this testing would be able to exercise all conditions of the component using traditional testing techniques [16].

Fault injection should be used as a supplement to traditional testing techniques. Since traditional testing is intended to check that a system meets the requirements of the specification using expected input and conditions, whilst fault injection techniques are intended to test the operation of a system under exceptional conditions and invalid input. Because fault injection tests exceptional conditions and accelerates the occurrence of errors it is not suitable for the measurement of reliability or MTBF since these statistics are related to time. These must be measured by observational techniques [17].

The technique of fault injection dates back to the 1970s [18] when it was first used to induce faults at a hardware level. This type of fault injection was called Hardware Implemented Fault Injection (HWIFI) and attempted to simulate hardware failures within a system. It was soon found that faults could be induced by software techniques and that aspects of this technique could be useful for testing software systems. Collectively these techniques are known as Software Implemented Fault Injection (SWIFI).

The earliest attempts SWIFI was Mill's *fault seeding* approach [19] which was later refined by *stratified fault-seeding* [20]. These techniques worked by adding a number of known faults to a software system for the purpose of monitoring the rate of detection and removal. Given this, it is possible to estimate the number of remaining faults in a software system still to be detected by a particular test methodology. True SWIFI methods of injecting faults that simulated HWIFI soon followed [11].

3.1. SWIFI Methods

In recent years there has been interest in developing SWIFI based tools. This is partly because a SWIFI tool doesn't require any expensive specialized hardware. SWIFI also allow specific systems running on target hardware to be effectively targeted without injecting faults into other parts of the system. This is difficult to do with HWIFI techniques.

SWIFI also has a number of drawbacks: a) Faults cannot be injected into memory that is protected, b) Instrumentation of the code to be tested may perturb the operation of the system, c) Timing of events may be inaccurate because the timers available to a software system on some hardware platforms may not have a high enough resolution to capture short latency faults.

SWIFI techniques can be categorized into two types. *Compile-Time Injection* and *Runtime Injection*.

Compile-Time Injection (also know as code mutation) is an injection technique where source code (or assembler code) is modified to inject simulated faults into a system. A simple example of this technique could be changing

a = a + 1 to ***a = a – 1***

Although these types of faults can be injected by hand the possibility of coding in an unintended fault is high, so tools exist to parse a program automatically and insert faults.

This technique has the advantage that it can be used to simulate both hardware and software faults and has been shown to induce faults into a system that are very close in nature to those produced by programming faults [21]. It requires no expensive hardware and no additional software during runtime. Because the faults are coded into the running system and require no communication with a fault injector it has a considerably smaller impact on the execution timing of the system under test. Since the faults are hard coded into the system image it is possible to emulate permanent faults as well as transient faults. Finally this system is very simple to implement.

The main drawbacks of this technique are that it requires the modification of the actual source code. This means that the source code must be available to the test team (which will most likely not be the case for COTS systems). Secondly since the code is being modified there is the chance that unintended faults will be introduced, especially if the faults are being injected by hand. Lastly since the source code is being altered this technique can't be used as part of a certification processes since the system under test will be a different system to that which is shipped and a less invasive technique will have to be used.

Runtime Injection techniques use a software trigger to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways.

Robustness testing: Marsden et al states in [22] *"The robustness of a system is a measure of its ability to operate correctly in the presence of invalided inputs and stressful environmental conditions"*. This is achieved by corrupting data at the interface level of a component of observing its behavior. Running the system under a heavy processor load and observing its effects can also achieve robustness testing.

Network Level Fault Injection [23]: injects faults by corrupting, reordering or dropping network packets at the network interface. It is possible to inject faults by instrumenting the protocol stack as in [24] but this runs the risk of being detected and rejected by the receiving systems protocol stack. It is therefore preferable to inject the fault at the application level before this stage [13]. They are then processed normally by the protocol stacks at both ends and can be relayed to the application layer. The faults injected are usually based on the corruption of packet header information and injecting random byte errors into packet payloads.

3.2. Fault Models

A fault model is a model of the types of fault that can occur in a system whilst it is running. A fault model can be categorized into the following types of faults:

Physical Faults: These are faults that are caused by physical hardware failures, such as memory failures, processor failures, power spikes, etc. It has been found that this class of faults can be replicated effectively by using bit flipping techniques.

Software Faults: This class of faults includes both programming faults and design faults.

Resource-management faults: This class of faults includes such faults as memory leakage and exhaustion of resource such as file descriptors.

Communication faults: This class of faults is specific to systems that use some form of communication so they may not be present in all systems. These faults are concerned with simulating faulty network connections by the corruption, duplication, reordering and deletion of network messages. In tradition systems this class of fault is not seen as having a large effect on a system but [25] indicates that this class of fault may have a greater effect on a Web Service based system.

Life-cycle faults: This class of fault is concerned with the mechanisms that maintain objects. This class of faults includes such faults as premature object destruction, delayed asynchronous operations (outside specified timing constraints).

This list is not exhaustive but covers commonly found classes of faults found within most systems. A specific fault model must be defined for a system before its failure modes can be defined.

3.3. Failure Modes

Once a fault model has been defined the ways in which a system can fail must be defined. These are know as the *Failure Modes* and these will vary from system to system.

More detailed failure modes are defined on a per-application basis. For instance The Apache Software Foundation defines a set of failure modes for Axis [26] that are show in Table 1. The failure modes given can be arranged into groupings mainly concerned with communication, with only one group "Internal Error" being concerned with programming failures and exceptions.

This is a reasonable set of failure modes from an end user perspective of Web Service middleware such

as Apache Axis since it's failure model is concerned with the correct function of the middleware rather than users Web Services. The failure modes given in Table 1assume that the middleware is fault free and these are the failure modes that will be encountered by a normal user. To test the actual middleware a far more extensive set of failure modes is required since other failures may be encountered during testing. These failure modes will be more akin to the failure modes found in any large application.

Table 1: Axis Failure Modes

- Connection refused
- Unknown host
- Not Found
- Moved
- Wrong content type/MIME type
- XML parser error
- Internal Error
- Connection Timed out/ No Route To Host
- GUI hangs/ long pauses

Web Services are a rapidly evolving technology and as such specialized testing has concentrated on protocol issues and general middleware validation. It is a sad fact that most general texts on the subject neglect testing all together [27, 28] the assumption being that normal test methodologies can be used to test systems. Whilst this may be the case during development of an in-house SOA larger SOA using external third party components would not have access to source code so validation techniques could not be applied.

As far as we are aware dependability analysis of Web Services and middleware has not be extensively applied [13] although some work is now underway to apply Interface Propagation Analysis (IPA) to Web Services [29] although this work is only concerned with applying IPA to Web Services and not with providing a non-invasive framework to undertake dependability analysis.

A more detailed set of failure modes is given by Gorbenko et al [30]. This work concentrates on providing information on constructing fault tolerant Web Services from unreliable sets of Web Services. Consequently the failure modes given are aimed at providing a basis for constructing fault tolerant solutions rather than testing for correct implementation of Web Services, for instance the taxonomy given in this report groups all suspected corrupted results under one failure mode to allow these to be dealt with by Multi-Version Diversity mechanisms. Conversely for testing requirements we may require a higher level of granularity to help assess at which point a Web Service is failing.

4. WS-FIT

WS-FIT (Web Service – Fault Injection Technology) has been developed specifically to perform Network Level Fault Injection and parameter perturbation on SOAP based SOA and a detailed description of its design is given Looker et al [31]. The technology can cause state perturbation through the targeted modification of RPC parameters within SOAP messages and also inject communications faults. This is particularly useful in assessing fault tolerance mechanisms.

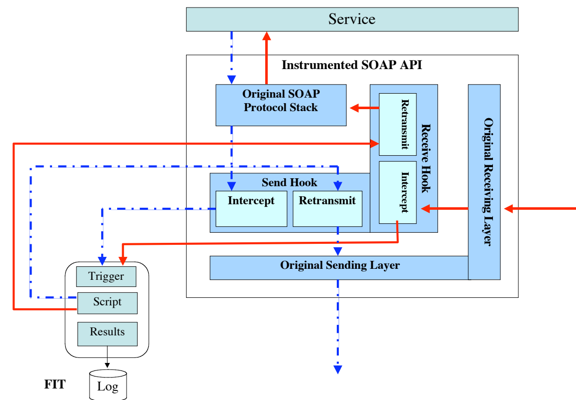

Figure 1: System Instrumentation

WS-FIT implements a variation of Network Level Fault Injection as a means of determining system dependability. Although we intend to inject faults into network packets we cannot do this directly because of the problems of altering encrypted/signed packets after they have been constructed. We therefore inject the faults at the API boundary between the application and the top of the protocol stack, this being the lowest, easily accessible point to inject faults before any encryption and signing has taken place. It overcomes these problems and allows us the level of control that we require for parameter perturbation.

WS-FIT uses an instrumented SOAP API that includes two small pieces of hook code. One hook intercepts outgoing messages, transmits them via a socket to the fault injector engine and receives a modified message back from the fault injector. This message is then transmitted normally to the original destination via the original protocol stack. There is a similar hook for incoming messages (See Figure 1).

WS-FIT creates an ontology from the WSDL used to describe the SOA under test. The WSDL defines all interfaces and messages that flow between SOA elements. Specification data can be added manually through the WS-FIT tool, for instance upper and lower bounds on parameters, but eventually our aim is to remove this manual step and import the information directly from electronically stored specifications.

The ontology can be used to decode and intercepted RPCs and allow the user to construct triggers to fire on certain events. Test scripts can be constructed to run on the triggers and inject perturbations into RPC messages in much the same way that state perturbation functions operate using code insertion techniques.

4.1. Enhanced Fault Model

Our enhanced fault model takes the standard concept of a fault model and extends it to include more detail. This is done by apply the technique of functional decomposition to the top-level fault model, for instance we can define a normal fault model (See Figure 2) which shows a fairly standard fault model for a SOA. Then each item in the fault model is decomposed into more detailed sub-categories after which further decomposition of each item in the model can continue until a level is reached where we have a detailed enough description to implement a specific test case (See Figure 3).

The detailed test case should be generic enough so that it can be applied to any message/parameter of an appropriate type, for instance a test case could be written to perturb any integer input parameter so that it contains a random value within a specified range. The range is entered for a particular parameter and this information is obtained from the specification of that RPC.

The enhanced fault model is intended to be extendable so that a user of WS-FIT can customize the enhanced fault model to cater for the system under test. The model described here and implemented in the tools is therefore not extensive but is intended to provide a basic level of functionality that can be enhanced for a specific system.

It is possible, by careful perturbation of input parameters, to perturb the internal state of a service but this requires detailed knowledge of the design of the service and so is considered outside the scope of WS-FIT's automatic test case generation, but it can be accomplished via manual test case construction and manual inspection of the service code.

- Physical Faults
- Software Faults
- Resource Management Faults
- Communication Faults
- Life-cycle Faults

Figure 2: High Level Fault Model

- **Software Faults**
  - ***Perturbation of Data into a Service***
    - **Values in Specified Range**
      - Upper Bound
        - *Replace specified parameter with the upper bound value specified for this parameter.*
      - Upper Bound – 1
        - *Replace the specified parameter with the upper bound specified for this parameter with one subtracted from it.*
      - Lower Bound
        - *Replace specified parameter with the lower bound value specified for this parameter.*
      - Lower Bound + 1
        - *Replace specified parameter with the lower bound value specified for this parameter with one added to it.*
      - Random Values between Upper and Lower Bounds
        - *On test generation, generate a static sequence of randomly distributed values that lie between the upper and lower bounds inclusively. Cyclically substitute the next value from the statically generated sequence for the specified parameter.*
    - **Values out of Specified Range**
      - Upper Bound + 1
        - *Replace specified parameter with the upper bound value specified for this parameter with one added to it.*
      - Lower Bound – 1
        - *Replace specified parameter with the lower bound value specified for this parameter with one subtracted from it.*

Figure 3: Detailed Fault Model

4.2. Enhanced Failure Model

To detect potential failures in a system, WS-FIT includes an enhanced failure model. This is similar in concept to the enhanced fault model described in Section 4.1. We have defined a high level set of failure modes that are suitable for use with an SOA.

WS-FIT applies decomposition to the failure modes in a similar way to the enhanced fault model, to give an enhanced failure model. Decomposition is applied iteratively to the failure modes to compose a number of partial models. These partial models sub-divide each failure mode into a number of sub-models

until a simple enough level is reached, at which point a script can be written to detect the detailed failure mode (See Figure 4).

- **Corruption of data out of service**
  - ***Data out of range***
    - Data above upper bound
      - *Check specified parameter against upper bound in specification and if it is greater than this value flag an error condition.*
    - Data below upper bound
      - *Check specified parameter against lower bound in specification and if it is less that this value flag an error condition.*

Figure 4: Detailed Failure Model

Once a detailed enhanced failure model is available, it is applied to the whole (or part) of the SOA, so unlike the enhanced fault model it is active for all message exchanges so it can detect normal failures as well as failures caused by fault injection.

5. Fault Models for SOA

Currently our WS-FIT tool and Extended Fault/Failure Models are capable of automatically generating unit tests for single Web Services. This is done by applying Fault Models to specific RPC messages and parameters contained within them. This can be done with a minimum of knowledge of the internal working of a Web Service and be based on the WSDL for that service as shown in Looker et al [2]. Our tools can be used to coordinate testing between multiple Web Services in an SOA but this requires detailed knowledge of the operation of the SOA under test.

For instance, a stock trading SOA is shown in Figure 5. It is made up of a Bank service that handles simple deposits, withdrawals and balance requests; A Quote service that provides real-time stock quotes; and a Trading service which sets buy and sell limits and handles the purchase of shares.

WS-FIT can be used to individually robustness test each individual component but cannot currently automatically assess the operation of the system as a whole.

For this scenario a skilled tester is required who possess heuristic rules that they apply to generate specific test scripts that coordinate tests between the different components, such as triggering a bank deposit transaction to be lost/corrupted after a stock quote of a certain limit is detected.

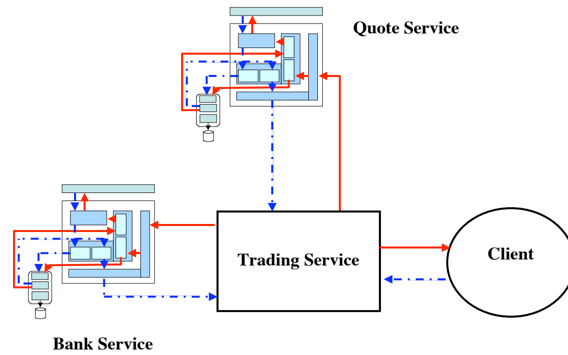

Figure 5: Stock Trading SOA

We envisage that these heuristic rules will differ between SOAs constructed to solve different types of problems, for instance Finance, Data Intensive, Scientific (GRID) applications, etc. would each have their own differing sets of heuristic rules to construct test cases.

These rules would be derived from studying test case systems for each classification of system. Ideally there should be multiple examples of each classification of SOA so that the derived models could be extensively evaluated.

6. Pedagogic Data

A major problem in evaluating any assessment technique or fault model is the availability of systems to evaluate it against. Ideally, to test the effectiveness of a particular technique or fault model, there should be a number of independently implemented systems derived from the same specification. This would not only allow a statistical evaluation of the technique but it would also provide a range of different faults in each system.

Unfortunately this is rarely the case since most software systems are rarely developed to the same specification. Specifications are usually developed 'in-house' to specify a system, even COTS systems will tend to differ slightly in specification since they will attempt to differentiate themselves from competitors in the market place.

An exception to this rule is the generation of undergraduate software projects. These are often constructed to an identical specification to simplify marking. We propose to set a series of group projects centering around SOA to provide a number of similar but independently implemented systems, from which we can extract our heuristic rules for assessing SOAs.

To a certain extent we would expect common-mode failure to be a factor in the implementation of these systems. Since software engineers are taught to construct systems in certain ways it is reasonable to

assume that certain common faults would be introduced into all the systems.

We further assume that systems generated pedagogically would include more faults since undergraduate students would have less experience constructing large systems. By comparing the generated systems with a professionally implemented control system it may be possible to isolate commonly introduced undergraduate faults. This data could then be fed back into the undergraduate teaching program to improve teaching practices.

7. Summary

This paper provides a review of dependability and fault injection techniques. We have provided a description of our current WS-FIT method and have indicated our intended directions of research.

Since multiple test case systems are not normally available we have proposed that we use pedagogically generated systems as a possible source of test cases. This would have the added benefit that common faults could be fed back into teaching methods to improve teaching quality.

8. Acknowledgments

This work was partly funded by the HEFCE funded project Active Learning in Computing (ALiC) and also by the EPSRC through a DTA studentship.

9. References

[1] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," in *IEEE Internet Computing*, vol. 6, 2002, pp. 86-93.

[2] N. Looker, B. Gwynne, J.Xu, and M. Munro, "An Ontology-Based Approach for Determining the Dependability of Service-Oriented Architectures," presented at 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems, Sedona, USA, 2005.

[3] I. Sommerville, *Software Engineering*, 7 ed: Addison-Wesley, 2004.

[4] R. Steinmetz and K. Nahrstedt, *Multimedia Computing, Communications & Applications*: Prentice Hall, 1995.

[5] A. Mani and A. Nagarajan, "Understanding Quality of Service for Web Services," vol. 2005: IBM, http://www-106.ibm.com/developerworks/library/ws-quality.html?n-ws-1172, 2002.

[6] A. S. Tanenbaum and M. v. Steen, "Distributed Systems: Principles and Paradigms," Prentice Hall, 2002, pp. 393-400.

[7] A. S. Tanenbaum, *Distributed Operating Systems*. London: Prentice-Hall International, 1995.

[8] R. S. Pressman, "Software Engineering: A Practitioner's Approach," 3rd ed: McGraw-Hill, 1992, pp. 549-594.

[9] IFIP, "IFIP WG10.4 on Dependable Computing and Fault Tolerance," International Federation For Information Processing, http://www.ifip.or.at/bulletin/bulltcs/tc10_aim.htm#wg104, 1988.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.

[11] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*: John Wiley & Sons, 1998.

[12] B. Kitchenham, "Measuring Software Development," in *Software Reliability Handbook*, P. Rook, Ed., 1st ed: Elsevier Applied Science, 1990, pp. 303-331.

[13] E. Marsden, J. Fabre, and J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection," presented at Symposium on Reliable Distributed Systems, Osaka, Japan, pp. 276-285, 2002.

[14] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure Data Analysis of a LAN of Windows NT based Computers," presented at Reliable distributed systems, Lausanne, Switzerland, pp. 178-189, 1999.

[15] H. Hecht and M. Hecht, "Qualitative Interpretation of Software Test Data," presented at Computer-aided design, test, and evaluation for dependability, Beijing, pp. 175-181, 1996.

[16] J. A. Whittaker, "Software's Invisible Users," *IEEE Software*, vol. 18, pp. 84-88, 2001.

[17] M. C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, pp. 75-82, 1997.

[18] J. V. Carreira, D. Costa, and S. J. G, "Fault Injection Spot-Checks Computer System Dependability," *IEEE Spectrum*, pp. 50-55, 1999.

[19] H. D. Mills, "On the Statistical Validation of Computer Programs," IBM Federal Systems Division, Gaitherburg, MD, Red. 72-6015, 1972.

[20] L. J. Morell and J. M. Voas, "Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability," College of William and Mary in Virginia, Department of Computer Science WM-82-2, September 1998 1988.

[21] M. Daran and P. Thevenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *Software Engineering Notes*, vol. 21, pp. 158-171, 1996.

[22] E. Marsden, J. Fabre, and J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection," presented at Symposium on reliable distributed systems, pp 279, Osaka, Japan, pp. 279, 2002.

[23] J. A. Whittaker and H. H. Thompson, *How to Break Software Security*: Addison-Wesley, 2003.

[24] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool," *Software Practice and Experience*, vol. 27, pp. 1385-1410, 1997.

[25] N. Looker and J. Xu, "Assessing the Dependability of OGSA Middleware by Fault Injection," presented at Symposium on Reliable Distributed Systems, pp. 293-302, 2003.

[26] Apache, "Client-Side Axis," vol. 2005: Apache Software Foundation, http://ws.apache.org/axis/java/client-side-axis.html, 2005.

[27] S. Potts and M. Kopack, *Teach Yourself Web Services in 24 Hours*, 1st ed: SAMS, 2003.

[28] T. Erl, *Service-Oriented Architecuture: A Field Guide to Integrating XML and Web Services*, 1st ed: Prentice Hall, 2004.

[29] J. Zhang, "An Approach to Facilitate Reliability Testing of Web Service Components," presented at International Symposium on Software Reliability Engineering and Technology, Saint-Malo, Bretagne, France, pp. 210-218, 2004.

[30] A. Gorbenko, V. Kharchenko, P. Popov, A. Romanovsky, and A. Boyarchuk, "Development of Dependable Web Services out of Undependable Web Components," School of Computing Science, University of Newcastle upon Tyne CS-TR-863, 2004.

[31] N. Looker, M. Munro, and J. Xu, "Simulating Errors in Web Services," *International Journal of Simulation Systems, Science & Technology*, vol. 5, 2004.