Advancing Software Services Robustness

Techniques for Assessment and Improvement

Carlos Nuno Bizarro e Silva Laranjeiro

Dissertation submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

January 2012

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work is within the Dependable Systems specialization domain and was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC).

Funding for this work was partially provided by the Portuguese Research Agency *Fundação para a Ciência e Tecnologia* (FCT) through the scholarship SFRH/BD/48333/2008, through the project "*TACID - Timely ACID transactions in DBMS*" – POSC/EIA/61568/2004, and by the Portuguese Government/European Union through R&D Unit 326/94 CISUC.

This work has been supervised by **Professor Marco Paulo Amorim Vieira**, Assistant Professor of the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Abstract

The use of software services in the development and integration of enterprise applications in business-critical scenarios has been steadily increasing. Software services consist of components that can be used by other applications across the web and are supported by standard protocols and specifications. In a service-based environment, providers supply a set of operations for consumers. These operations, supported by a complex software infrastructure (including application servers, middleware stacks, databases, etc.), are based on an immense variety of technologies, and use other services to compose complex business processes (i.e., a collection of services working together towards a goal).

The increasing use of service-based applications in critical business-to-business environments raises the need for new techniques and tools that support the development and deployment of services that fulfill key dependability properties. Currently, services are dynamic, use a vast diversity of systems, are evolving and are frequently deployed over public unreliable networks. Additionally, service developers normally face time-to-market pressure, leading them to focus on functionality development and disregard important testing activities, which may result in the deployment of software with bugs, including robustness problems and security vulnerabilities.

This thesis addresses the problem of robustness assessment and improvement in software services. In particular, we first propose a generic approach for assessing the robustness of distinct classes of services (e.g., web and messaging services). The approach is based on a set of tests, which comprise a combination of valid and invalid input parameters, and are used in services operations. Besides defining the general procedure, the approach describes all the components necessary for testing services for robustness. The proposal is demonstrated with two concrete case studies on web services and messaging middleware.

The thesis also proposes a technique to fix robustness problems in web services. This technique, which extends to the development process itself, is based on the definition and announcement of the service input and output domains in a complete way. Wrappers are automatically built based on the domain definitions to prevent the execution of the service with invalid input parameters. The wrapping technique is taken one step

further to include protection against malicious inputs (although possibly valid in the domain). To achieve this goal, we introduce a learning phase to gather invariant information representing the profile of regular (i.e., non malicious) client requests. The robustness wrapper is then able to use that information, which is complemented with a set of heuristics to handle new cases (i.e., previously unobserved), to prevent the execution of requests that fall out of the regular profile.

Finally, **the thesis proposes techniques to improve relevant fault tolerance aspects of web services**. In particular, we design two concrete mechanisms that endow web services with features that allow increasing correctness, availability and performance, including proper handling of timing requirements. The first mechanism helps developers to deploy fault-tolerant compositions using diverse services, by applying techniques like diversity and voting schemes. The second mechanism provides support for deploying services able to perform runtime detection and prediction of timing failures, based on the collection and analysis of historical data. In this case, when clients' timing requirements are exceeded or are not possible to (predictably) be guaranteed, the service consistently replies with a well known exceptional behavior.

Keywords
Software services; robustness; testing; fault tolerance; dependability assessment; dependability benchmarking; wrapping.

Resumo

O uso de serviços no desenvolvimento e integração de aplicações de larga-escala em ambientes críticos de negócio tem vindo a aumentar progressivamente. Estes serviços são componentes de *software* que podem ser usados por outras aplicações através da *web* e que são suportados por protocolos e especificações padronizados. Num ambiente baseado em serviços, fornecedores disponibilizam um conjunto de operações a consumidores. Estas operações, suportadas por uma infraestrutura complexa de *software* (incluindo servidores aplicacionais, bases de dados, etc.), baseiam-se em uma imensa variedade de tecnologias e usam outros serviços para compor processos de negócio complexos (i.e., um conjunto de serviços que colaboram em função de um objectivo).

A cada vez maior utilização de aplicações baseadas em serviços em ambientes *business-to-business* introduz a necessidade de novas técnicas e ferramentas que suportem o desenvolvimento de serviços com propriedades de confiabilidade. Os serviços são atualmente dinâmicos, usam uma grande diversidade de sistemas, evoluem e são frequentemente disponibilizados através de redes públicas não fiáveis. Para além disto, os programadores são frequentemente sujeitos a pressão de mercado para entregar *software*, o que os leva a concentrar no desenvolvimento de funcionalidades, ignorando frequentemente as atividades de teste e garantia de qualidade. Isto resulta na disponibilização de *software* com defeitos, incluindo problemas de robustez e vulnerabilidades de segurança.

Esta tese foca o problema da avaliação e melhoria de robustez em serviços. Em particular, é proposta uma abordagem genérica para avaliar a robustez de distintas classes de serviços (e.g., serviços *web* e serviços de troca de mensagens). A abordagem é baseada num conjunto de testes, que combinam parâmetros válidos e inválidos, definindo, para além do procedimento genérico, todos os componentes necessários para testar a robustez de serviços. A proposta é ilustrada em dois casos de estudo concretos, nomeadamente serviços *web* e *middleware* de mensagens.

Este trabalho propõe também uma técnica para corrigir problemas de robustez em serviços *web*. Esta técnica, que se integra com o próprio processo de desenvolvimento de *software*, é baseada na definição completa dos domínios dos parâmetros de entrada e de saída dos serviços. Com

base nessa definição, são automaticamente construídos *wrappers* que impedem a execução do serviço na presença de parâmetros de entrada inválidos. Esta técnica de encapsulamento é posteriormente desenvolvida para incluir a proteção contra valores de entrada maliciosos, os quais podem ser válidos no domínio. Para tal, é introduzida uma fase de aprendizagem onde é recolhida informação invariante, a qual representa o perfil de pedidos normais (i.e., não maliciosos). O mecanismo de robustez usa esta informação, complementada com um conjunto de heurísticas que permitem lidar com novos casos (i.e., previamente não observados), para impedir a execução de pedidos que estejam fora do perfil regular.

Finalmente, a tese propõe técnicas para melhorar aspetos de tolerância a falhas relevantes em serviços *web*. Em particular, são desenhados dois mecanismos que permitem aumentar a fiabilidade, disponibilidade e desempenho, incluindo resposta a requisitos temporais. O primeiro mecanismo permite aos programadores desenvolver composições tolerantes a falhas, através da utilização de serviços diversos e de técnicas como diversidade e esquemas de votação. O segundo mecanismo fornece suporte para o desenvolvimento de serviços capazes de detetar e prever falhas temporais durante a execução, com base na recolha e análise de dados históricos. Neste caso, quando os requisitos temporais definidos pelo cliente são excedidos ou (previsivelmente) não podem ser garantidos, o serviço responde com um comportamento excepcional bem conhecido.

Palavras-chave:
Serviços; robustez; testes; tolerância a falhas; avaliação de confiabilidade; testes padronizados de confiabilidade; encapsulamento.

Acknowledgements

I would like to thank Professor Marco Vieira for having me as a PhD student and for all the support during the development of this work. This thesis would not have been possible without his motivation skills, dedication, and outstanding quality. More than a rigorous and determined advisor, his human skills and enthusiasm were crucial in motivating me to aim for more.

I also would like to thank Professor Henrique Madeira for the several research contributions during the last years.

A word of appreciation to the members of the Software and Systems Engineering Group of the Centre of Informatics and Systems of the University of Coimbra for the excellent quality of the work they develop.

To the anonymous reviewers of the papers for their constructive comments that helped producing a higher quality work.

To my family and friends for supporting me during this long path.

Last but not least, to Naghmeh for all the support given whenever was necessary.

List of Publications

This thesis relies on the published scientific research present in the following peer reviewed papers:

P 1. Nuno Laranjeiro and Marco Vieira, "Adapting Test-Driven Development to Build Robust Web Services," *Agile and Lean Service-Oriented Development: Foundations, Theory and Practice*, ed. Xiaofeng Wang, Nour Ali, Isidro Ramos, and Richard Vidgen, IGI Global. *(to appear)* 2012.

P 2. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Building Web Services with Timing Requirements," *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, ed. Valeria Cardellini, Emiliano Casalicchio, Kalinka Castelo Branco, Julio Cezar Estrella, and Francisco José Monaco, IGI Global, June 2011.

P 3. Gabriella Carrozza, Aniello Napolitano, Nuno Laranjeiro, and Marco Vieira, "WSRTesting: Hands-on Solution to Improve Web Services Robustness Testing," *Fifth Latin-American Symposium on Dependable Computing (LADC 2011)*. São José dos Campos, São Paulo Brazil: IEEE Computer Society, 25-29 April 2011.

P 4. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "A Learning-Based Approach to Secure Web Services from SQL/XPath Injection Attacks," *The 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2010)*, Tokyo, Japan: IEEE Computer Society, 3-5 December 2010.

P 5. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Robustness Validation in Service Oriented Architectures," *Architecting Dependable Systems VI*, ed. Rogério Lemos, Jean-Charles Fabre, Cristina Gacek, Fabio Gadducci, and Maurice Beek. LNCS State-of-the-Art Survey Series, Springer-Verlag, 27 October 2009.

P 6. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Protecting Database Centric Web Services against SQL/Xpath Injection Attacks," *20th International Conference on Database and Expert Systems Applications (DEXA 2009)*, 271-278, Linz, Austria: Springer-Verlag, ISBN: 978-3-642-03572-2, doi:10.1007/978-3-642-03573-9_22, 31 September – 5 August 2009.

P 7. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Improving Web Services Robustness," *IEEE International Conference on Web Services (ICWS 2009)*, 397-404, Los Angeles, California, USA: IEEE Computer Society, ISBN: 978-0-7695-3709-2, doi:10.1109/ICWS.2009.27, 6-10 July 2009.

P 8. Nuno Laranjeiro and Marco Vieira, "Extending Test-Driven Development for Robust Web Services," *International Conference on Dependability (DEPEND 2009)*, 122-127, Athens/Vouliagmeni, Greece: IEEE Computer Society, ISBN: 978-0-7695-3666-8, doi:10.1109/DEPEND.2009.25, 18 June 2009.

P 9. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Predicting Timing Failures in Web Services," *International Workshop on Managing Data Quality in Collaborative Information Systems (MCIS 2009) at the International Conference on Database Systems for Advanced Applications (DASFAA 2009)*, 5667/2009:182-196. Vol. 5667, Brisbane, Australia: Springer-Verlag, ISBN: 978-3-642-04204-1, doi:10.1007/978-3-642-04205-8, 21-23 April 2009.

P 10. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Timing Failures Detection in Web Services," *IEEE Asia-Pacific Services Computing Conference (APSCC 2008)*, 554-559, Yilan, Taiwan: IEEE Computer Society, ISBN: 978-0-7695-3473-2, doi:10.1109/APSCC.2008.236, 9-12 December 2008.

P 11. Nuno Laranjeiro and Marco Vieira, "Deploying Fault Tolerant Web Service Compositions," *International Journal of Computational Science and Engineering Special Issue on Engineering Fault Tolerant Systems (EFTS)*, ISSN: 0267-6192, CRL Publishing 23, no. 5: 337-348, 5 September 2008.

P 12. Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Experimental Robustness Evaluation of JMS Middleware," *IEEE International Conference on Services Computing (SCC 2008)*, 119-126, Honolulu, Hawaii: IEEE Computer Society, ISBN: 978-0-7695-3283-7, doi:10.1109/SCC.2008.129, 8-11 July 2008.

P 13. Nuno Laranjeiro, Salvador Canelas, and Marco Vieira, "wsrbench: An On-Line Tool for Robustness Benchmarking," *IEEE International Conference on Services Computing (SCC 2008)*, 187-194, Honolulu, Hawaii, USA: IEEE Computer Society, ISBN: 978-0-7695-3283-7, doi:10.1109/SCC.2008.123, 9-11 July 2008.

P 14. Nuno Laranjeiro and Marco Vieira, "Towards Fault Tolerance in Web Services Compositions," *2nd International Workshop on Engineering Fault Tolerant Systems (EFTS 2007), The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, Dubrovnik, Croatia: ACM, ISBN: 978-1-59593-725-4, 3-7 September 2007.

P 15. Marco Vieira, Nuno Laranjeiro, and Henrique Madeira, "Benchmarking the Robustness of Web Services," *13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007)*, 322-329, Melbourne, Victoria, Australia: IEEE Computer Society, ISBN: 0-7695-3054-0, doi:10.1109/PRDC.2007.56, 17-19 December 2007.

P 16. Marco Vieira and Nuno Laranjeiro, "Comparing Web Services Performance and Recovery in the Presence of Faults," *IEEE International Conference on Web Services (ICWS 2007)*, 623-630, Salt Lake City, USA: IEEE Computer Society, ISBN: 0-7695-2924-0, doi:10.1109/ICWS.2007.63, 9-13 July 2007.

P 17. Marco Vieira, Nuno Laranjeiro, and Henrique Madeira, "Assessing Robustness of Web-Services Infrastructures," *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, 131-136, Edinburgh, United Kingdom: IEEE Computer Society, ISBN: 0-7695-2855-4, doi:10.1109/DSN.2007.16, 25-28 June 2007.

Preliminary versions of the work have been presented in the following short papers:

P 18. Nuno Laranjeiro and Marco Vieira, "Towards Automatic Classification of Web Services Robustness," *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010 – Fast Abstract)*, Chicago, Illinois, USA, 28 June – 1 July 2010.

P 19. Nuno Laranjeiro, "Assessing and Improving the Robustness of Service-based Applications," *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009 – Student Forum)*, Estoril, Portugal: IEEE Computer Society, 29 July – 2 August 2009.

P 20. Nuno Laranjeiro and Marco Vieira, "Testing Web Services for Robustness: A Tool Demo," *European Workshop on Dependable Computing (EWDC 2009 - Demonstration)*, Toulouse, France: IEEE Computer Society, 14-15 May 2009.

The following papers are related to this thesis but were not included:

P 21. Rui Oliveira, Nuno Laranjeiro, and Marco Vieira, "A Composed Approach for Automatic Classification of Web Services Robustness," *The 8th International Conference on Services Computing (SCC 2011)*, Washington D.C., USA: IEEE Computer Society, 4-9 July 2011.

P 22. Nuno Laranjeiro, Rui Oliveira, and Marco Vieira, "Applying Text Classification Algorithms in Web Services Robustness Testing," *29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)*, New Dehli, India: IEEE Computer Society, 31 October – 3 November 2010.

P 23. Nuno Antunes, Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services," *IEEE International Conference on Services Computing (SCC 2009)*, 260-267, Bangalore, India: IEEE Computer Society, ISBN: 978-0-7695-3811-2, doi:10.1109/SCC.2009.23, 21-25 September 2009.

P 24. Mônica Dixit, Antonio Casimiro Costa, Nuno Laranjeiro, and Marco Vieira, "Using Experimental Measurements to Assess Dependable Adaptation Support Mechanisms for Timed Transactions," *Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems at the 27th International Symposium on Reliable Distributed Systems (SRDS 2008)*, Napoles, Italy: IEEE Computer Society, 6-8 October 2008.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Services and Service Oriented Architectures (SOA) are among the most important trends in modern software development (McKinsey & Company and SandHill Group 2008). Software services, consisting of components that may be used by other applications across the web, are supported by standard protocols and specifications, and represent key elements in SOA solutions (Erl 2005). Examples of largely used service technologies are web and messaging services. Web services are self-describing software components that provide developers with interoperability and platform-independency properties, being supported by protocols such as SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language) (Curbera et al. 2002). Messaging Services, such as JMS (Java Messaging Service) and MSMQ (Microsoft Message Queuing), enable programmers to build loosely coupled applications based on messages exchanging.

Software services are increasingly being used to support critical business-to-business interactions, linking suppliers and clients in sectors such as banking and financial services, transportation, automotive manufacturing, and healthcare, just to name a few. Such components provide a well-defined interface between two parties, where the former offers a set of services that are used by the latter. This simple interface enables the aggregation of functionalities in more complex units that work as a whole to achieve an objective, typically not attainable individually. These units

are usually designed as *business processes* and can be seen as a series of steps (e.g., web service calls, message broadcasts) that are executed in predefined sequences according to specific business rules and runtime conditions (Erl 2005). In such context, a failure in one of the components can impair the whole composition (for instance, by degrading or stopping service delivery). As an example, a recent failure in Amazon Web Services (Amazon 2011) affected many Internet websites resulting in total service unavailability in some cases and degraded operation in other sites (C. C. Miller 2011).

Dependability can be defined as the ability to deliver a service that can be justifiably trusted (Algirdas Avizienis et al. 2004). It is an integrative concept that includes the following attributes: availability (readiness for correct service), reliability (continuity of correct service), safety (absence of catastrophic consequences on the user(s) and the environment), confidentiality (absence of unauthorized disclosure of information), integrity (absence of improper system state alterations), and maintainability (ability to undergo repairs and modifications).

There are other aspects that refine the main dependability attributes. Robustness – dependability with respect to external faults (Algirdas Avizienis et al. 2004) – is of particular relevance to the services context, and characterizes essentially the response of a system to external faults, including invalid inputs (from a domain point-of-view) and malicious inputs (although these may be valid in the domain). Such input conditions may ultimately result in interoperability problems and severe security issues. Although previous studies have tested, with great success, the robustness of several software systems in presence of invalid application inputs (P. Koopman et al. 1997; P. Koopman and DeVale 1999; Fabre et al. 1999; Rodríguez et al. 1999; Rodríguez, Albinet, and Arlat 2002), developers still urge the definition of generic techniques and tools that facilitate the deployment of robust software services.

The reality is that the increasing use of service-based applications in business-critical environments is raising the need for new techniques and tools to help developing and deploying services that fulfill relevant dependability properties. Currently, services are dynamic, use an immense diversity of systems, are evolving and are frequently deployed over public unreliable networks. Additionally, service developers normally face time-to-market pressure, leading them to focus on functionality development and disregard important testing activities, which may result in software with residual bugs. As a consequence, there

are key issues regarding robustness and fault tolerance in general that need to be addressed and that map to the goals of this thesis, namely:

1. To propose a robustness assessment methodology for software services;
2. To devise ways for improving the robustness of services, including protection against possibly valid but malicious input conditions;
3. To put forward techniques for increasing fault tolerance characteristics in services environments.

1.1 Testing and improving services robustness

The need for practical means to assess and improve robustness in services is corroborated by several studies that show a clear predominance of software faults (i.e., program defects or bugs) as the root cause of computer failures (Lee and Iyer 1995; Kalyanakrishnam, Kalbarczyk, and Iyer 1999; Sullivan and Chillarege 1991) and, given the huge complexity of today's software, the weight of such faults tends to increase. Services are no exception, as they are frequently intricate software components that implement business critical processes, in some cases using compositions of several services, making them even more complex.

Interface faults, related to problems in the interaction among software components/modules (Weyuker 1998), are particularly relevant in service-based environments. In fact, services must provide a robust interface to the client applications even in the presence of invalid inputs, which may occur due to bugs in the client applications, corruptions caused by silent network failures, or even security attacks. The problem is that client applications are typically developed with the assumption that the services being used are robust, which is not always the case. Robustness failures in such environments are particularly dangerous, as they may originate vulnerabilities that can be exploited with severe consequences.

Robustness testing is an effective approach to characterize the behavior of a system in the presence of erroneous input conditions. It has been used mainly to assess the robustness of operating systems and operating systems kernels (Mukherjee and Siewiorek 1997; P. Koopman and DeVale 1999; Rodríguez et al. 1999), but the concept of robustness testing can be

applied to any kind of interface. Robustness tests stimulate the system under testing through its interfaces by submitting erroneous input conditions (i.e., limit and out-of-domain parameters) that may trigger internal errors or reveal vulnerabilities. Although robustness testing has proven to be an effective way for disclosing programming errors, until this date no practical and generic way has been proposed for the services context.

An application is considered free of robustness problems if it provides complete protection against invalid inputs. However, even when a service is protected against this kind of inputs, valid ones may still be a problem for the service itself or for the supporting infrastructure. For example, web services typically use several components, including data persistence components or external systems (Transaction Processing Performance Council 2008) based either in traditional relational databases or in XML databases. The incorrect use of user inputs and data access interfaces may result in an application being deployed with security issues. In fact, in web applications, the presence of security vulnerabilities allowing command injection attacks (e.g., SQL Injection and XPath injection) is particularly frequent (Christey and Martin 2007). These are input-based attacks that explore the valid domain of operations to take advantage of improperly coded applications to change the queries sent to a database, enabling, for instance, access to critical data. Solutions to secure web services against such attacks are thus extremely important for developers and providers to protect their services from this type of attacks.

Robustness testing is an extremely important tool during development as it enables developers to correct the disclosed issues. However, the generalized presence of robustness problems in software (P. Koopman and DeVale 1999; Shelton, Koopman, and Devale 2000; Mendonça and Neves 2007) indicates that measures that go beyond the identification of such problems are required. Indeed, developers need tools and techniques (e.g., extensions to the development process itself, automatic interface wrapping) that help them avoiding and/or automatically correcting robustness issues.

Improving the robustness of web services should then focus on two points-of-view. On one side, developers need techniques for protecting service applications against invalid inputs. Ideally, there should be a barrier at each application entry point that is able to use the available knowledge about the application domains to block incoming invalid requests (if generated automatically, this can reduce the developer's

coding effort and remove validation code from the service's business logic). On the other hand, there is a need to provide methods to secure services against valid, but possibly malicious inputs. In this case, a simple input barrier may not be sufficient, as a malicious input can take several forms inside the domain of valid inputs, easily rendering a simple barrier scheme useless. A mechanism able to distinguish genuine service requests from malicious ones is thus needed.

1.2 Improving services fault tolerance

Software services are typically deployed over unreliable transport channels (e.g., the Internet), and often suffer from long response delays or temporary unavailability. This becomes quite relevant when services are used in compositions. In fact, although the provider of the composition may try to guarantee that both the infrastructure and the software used are built in such a way that achieve high performance and dependability (by doing extensive tests, code reviews, selecting the adequate infrastructures, etc.), he cannot guarantee any quality attribute in what concerns the component services that are provided by third parties.

The need to deploy services with fault tolerance features is thus evident. Among many other aspects, this includes endowing them with mechanisms that allow increasing correctness, availability and performance, including properly handling timing requirements. A system that can withstand, overcome, or mitigate adverse conditions that may impact these attributes, can also be seen as being robust, in a more general sense.

Redundancy is used in several contexts to tolerate failures and achieve high availability (Marcus and Stern 2003). Design diversity is typically the solution to deal with design faults, where two or more different implementations of a component are used for fault-detection and fault-tolerance (A. Avizienis and Kelly 1984). Diversity can be used to achieve, not only high availability, but also high performance, or tolerance to business logic errors. In fact, a system that uses multiple diverse replicas is able to deliver its end-service even when one of the replicas fails, i.e., it is able to tolerate that specific failure while not compromising the overall availability. Similarly, it may use the fastest working replica to deliver a response to a client, increasing performance by lowering the overall response time.

Diversity allows tolerating programming errors in the diverse replicas. In practice, when replicas reply with different responses to the same request, a closer analysis of each response helps choosing the correct one (i.e., the one to deliver to the client). For example, the most frequently observed response could be selected as it has a higher probability of being correct. In general, diversity is helpful in service-based environments and applications, particularly if we consider the integration issues that arise when a developer tries to create a service that uses multiple components (Weyuker 1998), or even the limitations in controlling the overall quality of the components being used.

When the possibility of using diversity exists (as in many service environments) different invocation schemes can be set up and configured. However, frequently there are costs involved with the invocation of multiple services (e.g., computation power or even financial costs) (Amazon 2011). Even if the invocation costs involved are negligible, the output of the operation may be useless to the client when timing requirements are violated.

In typical web services environments, it is normal to observe high or highly variable execution times. The former is usually due to the nature of the involved protocols (e.g., the SOAP protocol involves a relatively high overhead), while the latter is typically associated with the unreliability of the Internet. In addition, as services are usually involved in business-to-business interactions, time may be very important and may influence aspects such as, client fidelity, income, and even reputation. In this sense, it is vital to deploy web services that are able to deal with clients' timing requirements (i.e., are robust to timing failures), and are able to optimize resources when it is detected or predicted that a given operation cannot conclude on due time (thus being of no use to the client).

Developers urge the definition of mechanisms that tackle the aforementioned aspects (correctness, availability, performance, and timing requirements), which impact the robustness of a system (in a broad sense). Such mechanisms must be simple and easy to use, not over-deviating developers from current development models or typical programming tasks.

1.3 Contributions of the thesis

As key contribution, this thesis proposes *techniques and tools to assess and improve the robustness of software services*. In detail, the main contributions are:

- A new **robustness testing approach for software services**. The approach is based on a combination of valid and invalid (including malicious) inputs that are used during service invocations to characterize their robustness (services can be distinguished according to the observed behavior). This approach is generic and can be used to test implementations of services but also their supporting middleware. Concrete examples of robustness testing for SOAP web services and messaging middleware are presented and discussed.
- The **extension of classical robustness testing** to include the execution of tests in special external contact points (e.g., external web services invoked by the service under testing) and the inclusion of malicious input values in the tests generation. These values typically exploit security vulnerabilities present in the system being tested, simulating command injection attacks (SQL and XPath Injection Attacks, in particular).
- A **three-dimension scheme for the classification of services robustness**. This scheme allows classifying services in terms of failure severity (i.e., using a failure mode scale) and with respect to the tester's visibility on the code being tested (extends the single failure mode scale presented in previous works (P. Koopman et al. 1997; Rodríguez et al. 1999)), but also the observed service behavior and the degree of compliance to a given specification.
- An **online tool**, named *wsrbench* (available at *http://wsrbench.dei.uc.pt*), that implements the services robustness testing approach proposed. This tool fills a gap in current development/testing tools, providing an easy interface for the robustness testing of web services.
- A **large-scale evaluation of public web services and home-implemented web services**. Nearly half-million web service responses were analyzed and several issues were disclosed using the *wsrbench* testing tool, including major security problems. We

show that web services are being deployed on the Internet with several robustness and security issues and that robustness testing can be effectively used during software development.

- A **robustness evaluation of major Java Messaging Middleware (JMS)** providers. The experimental evaluation performed shows the presence of major robustness and security issues in messaging providers. Results also indicate the existence of discrepancies between the different implementations tested and the standard JMS specification, emphasizing the importance of robustness testing in service-based applications.
- A **technique to improve the robustness of web services**. Our technique automatically endows web services with the capability of performing input verification based on a rich domain information descriptor. This descriptor consists of a new service domain description language (designated as Extended Domain Expression Language – EDEL) that is able to fully define service domains, including complex inter-parameter domain dependencies.
- An **automatable workload generation process adapted to web services**. This process is able to generate a set of service invocations based on the extended domain of a service specified using EDEL descriptions. The goal is to accurately exercise the web service implementation and use the generated workload as basis for testing and improving the web service's robustness.
- A **configurable approach to protect web services against malicious inputs** (yet, possibly domain-valid) that exploit common web service vulnerabilities (SQL/XPath Injection related vulnerabilities). The approach is based on learning the profile of genuine requests and uses that information to protect web service applications from attacks targeting SQL/XPath Injection.
- An **adaptation of Test-Driven Development (TDD) to include robustness testing**. TDD requires tests to be created and executed before implementing the actual functionality of an application. On the other hand, robustness tests are typically executed after development. As such, adjustments to TDD are proposed so that it benefits from robustness testing, urging developers to focus on robustness issues during web services development.

Adding to the contributions presented above, this thesis also puts forward *techniques and tools that target the improvement of relevant fault tolerance aspects of web services*. In particular, we make the following additional contributions:

- A **mechanism that allows developers to easily deploy fault-tolerant compositions using diverse web services**. Well-known techniques like diversity and voting schemes were studied to fit the specificities of web services and to deliver typically desired features in services environments. These features include providing high availability, delivering the best available response times, and increasing response correctness.
- A **mechanism for web services deployment that allows runtime detection and prediction of timing failures**. In this approach, when the clients' timing requirements are exceeded or are not possible to (predictably) be guaranteed, the service consistently replies with a well known exceptional behavior. This behavior enables clients to invoke alternative services that can provide responses on time, but is also important for providers to reduce the priority of requests that will not be delivered on due time, thus enabling better resource management.

1.4 Thesis structure

This first chapter introduced the problem addressed and the main contributions of the thesis.

Chapter 2 presents background and the state of the art on assessing and improving the robustness and fault tolerance of software components. Some of the addressed topics include robustness testing, robustness and security improvement, fault injection tools, fault-tolerance techniques and software development processes.

Chapter 3 presents the overall approach for assessing the robustness of services. This includes not only the definition of the required components, but also the definition of the procedure. All parts of the approach are defined in a generic way so that it can be applied to different classes of services (such as web and messaging services). Implications concerning

the visibility of the services being tested (that differ at the client and server sides) are also discussed in this chapter.

Chapter 4 presents specific cases on testing the robustness of services at the client side (i.e., tests executed without direct access to the service). In practice, the robustness testing methodology presented in Chapter 3 is instantiated to two specific technologies: SOAP web services and Java Message Service middleware.

Chapter 5 presents the instantiation of the robustness testing technique to server-side testing of web services (i.e., tests executed with access to the service code or bytecode). The approach for defining web service interfaces (using EDEL) is discussed in detail, not only because it is the basis for server-side robustness testing, but also because it is a starting point for improving services robustness at the server side (as proposed in the following chapter for the particular case of web services).

Chapter 6 proposes a technique for improving the robustness of web services, including protection against malicious inputs, and addresses the adaptation of Test-Driven Development to include robustness testing.

Chapter 7 presents the experimental evaluation targeting the topics discussed in earlier chapters. First, we present the experimental results of client-side robustness tests applied to a large sample of web services and messaging middleware implementations. Then, we discuss the results obtained during the server-side robustness testing and improvement of web services (including security improvement).

Chapter 8 presents the design and application of fault-tolerance mechanisms for the deployment of robust web services (in a broad sense). Specifically, the chapter addresses the use of diversity to improve relevant properties, such as response correctness, availability and response time.

Chapter 9 focuses on service response time and proposes a mechanism that can handle timing requirements. In practice, the mechanism endows services with the capability detecting and predicting timing failures.

The last chapter concludes the thesis and proposes topics for future research directions.

Finally, Annex A provides a description of the *wsrbench* tool (from a technical point-of-view) and Annex B contains detailed information about the public web services tested in the experimental evaluation presented in Chapter 7.

Chapter 2
Background and Related Work

Research on software services has gained ground in the last few years. However, there are many open research aspects that need to be tackled for effectively testing and improving the dependability. These rely obviously on existing concepts on dependability concepts. This way, this chapter presents background and related work on software services, dependability assessment and fault tolerance. More specifically, the topics addressed include software development lifecycles and software testing, fault injection (including the use of Aspect Oriented Programming (Kiczales et al. 1997) as a tool for implementing fault injection) and dependability benchmarking, robustness assessment and improvement (including the removal of software vulnerably), software redundancy and diversity, and timing failures detection and prediction.

The structure of the chapter is the following. Section 2.1 presents concepts on software services, in particular web services, messaging services, and services composition. Section 2.2 discusses development lifecycles and testing approaches for deploying software services. Section 2.3 presents the state of the art on experimental dependability assessment, including fault injection, dependability benchmarking and robustness testing. Section 2.4 presents existing work on robustness improvement, also

considering the mitigation of software vulnerabilities, and on fault tolerance (mainly software diversity and timing failures). Finally, Section 2.5 concludes the chapter.

2.1 Concepts on software services

Service Oriented Architecture (SOA) is an architectural style that steers all aspects of creating and using services throughout their lifecycle, as well as defining and providing the infrastructure that allows heterogeneous applications to exchange data. This communication usually involves the participation in business processes, which are loosely coupled to their underlying implementations. SOA represents a model in which functionality is decomposed into distinct units (services), which can be distributed over a network and can be combined together and reused to create business applications. This type of profile makes natural integration technologies, such as web services and JMS, innate candidates for SOA implementations. Despite this, there are other types of services, such as Enterprise Service Bus systems or RESTful services; however, SOAP-based web services (services that use the SOAP protocol for communication (Curbera et al. 2002)) and messaging services are the types of services focused on this section (and in the thesis, in general), due to their high relevance in services environments (Krafzig, Banke, and Slama 2004; Erl 2005).

2.1.1 Web Services

Web services provide a simple interface between a provider and a consumer and are a strategic mean for data exchange and content distribution (Krafzig, Banke, and Slama 2004). In fact, ranging from on-line stores to media corporations, web services are becoming a key component within organizations information infrastructure (Amazon 2011; Daniel et al. 2009; F. Casati and Machiraju 2003).

The web services framework is divided into three major areas: communication protocols, service descriptions, and service discovery. The main specifications for each area (SOAP, WSDL, and UDDI) are all XML-based (Curbera et al. 2002). XML (eXtensible Markup Language) is now firmly established as a language that enables information and data

encoding, platform independence, and internationalization (Bray et al. 2000).

SOAP (Simple Object Access Protocol) is a protocol for messaging that can be used along with existing transport protocols, such as HTTP, SMTP, and XMPP. WSDL (Web Services Description Language) is used to describe a web service as a collection of communication endpoints that can exchange particular types of messages. That is, a WSDL document describes the interface of the service and provides users (e.g., the web service's clients) with a point of contact. Finally, UDDI (Universal Description, Discovery, and Integration) offers a unified way to find service providers through a centralized registry (Curbera et al. 2002). Figure 2.1 presents a typical web services environment.

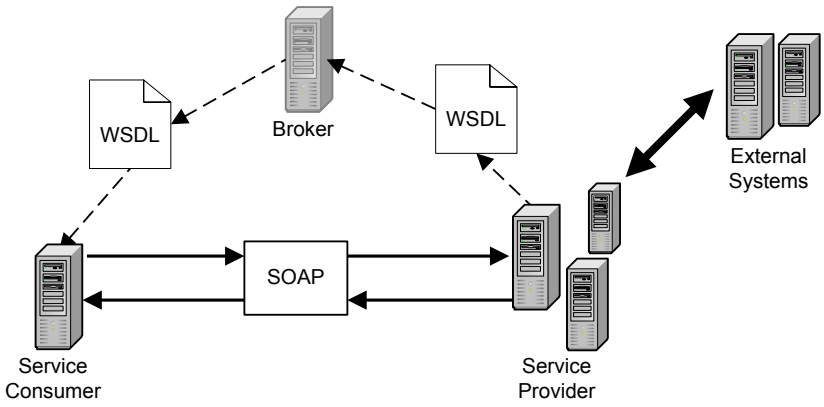


Figure 2.1 – Typical web services environment.

In a typical client-server distributed system model, a client process interacts with a server process in a distinct host computer (Coulouris, Dollimore, and Kindberg 2005). This interaction is frequently done by means of a message that is sent from the client to the server, and typically, in web services environments, the client waits for the server to compute a response that is sent back to the requesting client. In fact, in each interaction the consumer (client) sends a request SOAP message to the provider (the server). After processing the request, the server sends a response message to the client with the results. A web service may include several operations (in practice, each operation is a method with zero or more input parameters).

2.1.2 Messaging Services

Messaging services provide software components or applications with an easy method of communication. Typically, a messaging client connects to an agent that offers services for creating, sending, receiving, and reading messages. Messaging enables loose coupling between the participants in a distributed environment as senders and receivers do not have to be available at the same time. Also, they do not need to have any kind of knowledge of each other: the only agreement made is the message format (Jendrock et al. 2006).

Java Message Service (Sun Microsystems, Inc. 2002) is a Java API that defines the interfaces and semantics needed for the creation of applications that interact within a messaging environment. JMS unifies messaging concepts in a unique and simple, yet sophisticated, programming model that is implemented by multiple providers. Other types of messaging services include MSMQ and Oracle Advance Queuing (Microsoft Corporation 2010; Oracle 2011a). The next paragraphs present an overview of the JMS specification due to its relevance in the thesis context. However, in general, the main concepts presented below are also applicable to other messaging technologies.

In a JMS based environment, a Java middleware component provides services to create, send, receive, and read messages. Client applications produce and consume those messages by interacting with the services supplied by the middleware. Two messaging models are supported by JMS:

- The **point-to-point model** is based on message queues, senders and receivers. Senders send messages to queues, which make those messages available for consumption by receiving clients. Queues retain all messages until these are consumed or until they expire.
- The **publish/subscribe model** is based on the concept of topics. A publisher sends a message to a topic and this resource distributes the message to multiple subscribers (each message can have multiple consumers). Conceptually, the subscriber only receives the message if it is online at time it is published, but the JMS API allows durable subscriptions, loosening this restriction.

Figure 2.2 (adapted from (Jendrock et al. 2006)) shows the typical interaction between participants in a point-to-point messaging environment (a) and in a publish/subscribe environment (b).

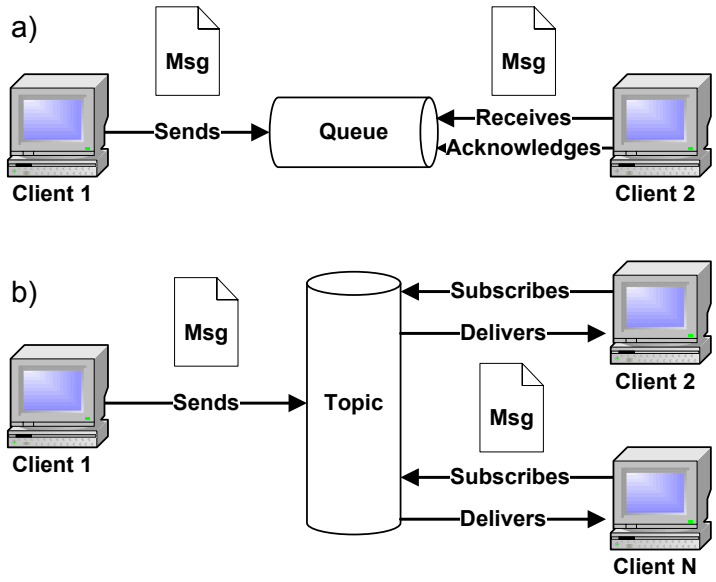


Figure 2.2 – Typical messaging environments.

2.1.3 Services composition

The clear interface and loose coupling that services technology provide promotes the creation of compositions of services, either using one or several technologies. For instance, web services compositions typically consist of a set of component web services that are invoked in a sequence that represents a specific business process. Component services can be provided by the same or different entities (having different providers is a common situation). The composition is then used by client applications to perform a more complex operation (a client application can also be another web service composition). Figure 2.3 presents an example of a typical service oriented environment based on web service compositions.

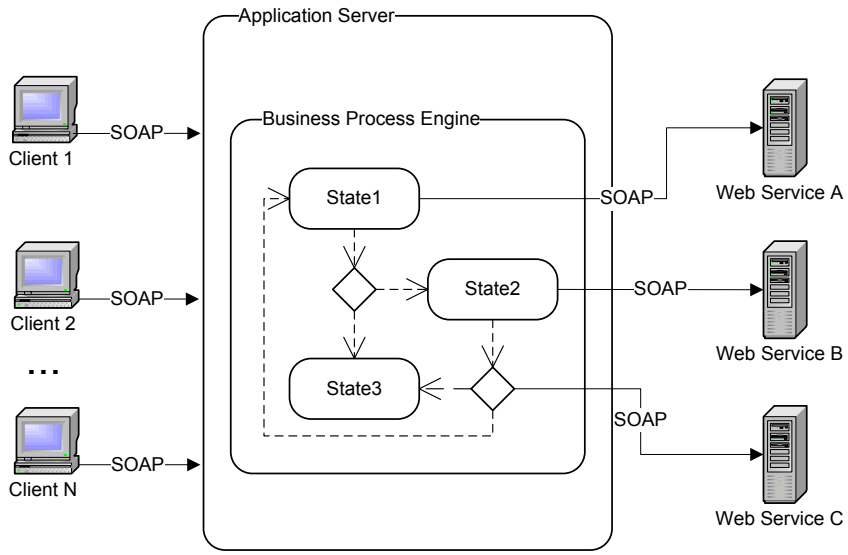


Figure 2.3 – Typical environment using web service compositions.

Several efforts have been undertaken to provide developers with ways for creating dependable services, namely through the standardization of reliable messaging protocols and transactional behavior support. However, dependability properties like robustness or tolerance to content and timing failures still remain unexplored or have been explored in an unsatisfactory way.

2.2 Developing software services

The process of creating services, and software in general, involves a large set of basic tasks that can be organized, controlled, and evaluated in many distinct ways. The software industry has used, since its inception, multiple methodologies or processes with the goal of creating high quality software. Along the years, these methodologies also evolved into forms that adapt better to the type of software being created, the type of physical and human resources involved, the programming techniques involved, among other aspects. In this section we present an overview of software development lifecycles and on software testing methodologies (a vital part of software development). We detail a recent software development process, Test-Driven Development, as it is a test-based methodology (with

particular relevance in the thesis context) that is gaining attention from developers.

2.2.1 Software development lifecycles

Software development processes nowadays range from highly structured to mostly informal. These different classes of methodologies are frequently referred to as prescriptive and agile, respectively (Janzen and Saiedian 2005). A prescriptive methodology prescribes how a new software system should be developed. Prescriptive software development methodologies are used as guidelines or frameworks to organize and structure how software development activities should be carried out, and in what order (Scacchi 2001). On the other hand, an agile methodology is typically a lightweight (in terms of level of formality and also degree of documentation) process for developing software. It values individuals and interactions, working software, customer collaboration and change response in contrast to aspects like processes, documentation, contracts, and plans, respectively (Highsmith and Cockburn 2001).

There are specific types of software development lifecycles such as waterfall, spiral, incremental and iterative, or evolutionary. **Iterative** development is based on the repetition of a set of development tasks, which use a set of requirements as basis. These requirements typically increase with time providing ground for a new iteration. **Evolutionary** software development methodologies imply adaptive and lightweight iterative development. Such techniques use feedback from earlier iterations as a way to improve the software being built (i.e., they are adaptive). They are typically lightweight in the sense of not demanding complete requirements or specifications at the starting point of development, which essentially allows the use of the iterations to guide future development. The **spiral** model is an evolutionary methodology that makes use of prototyping (an early sample or model built to test a concept or process) and the cyclic nature of iterative development. Milestones and iterations that take risk into account are used in this model. The **incremental** methodology can be characterized as a process that typically delivers a set of software releases (i.e., increments), where each release provides more functionality than the previous one (Janzen and Saiedian 2005).

Agile methodologies have roots in the iterative, incremental, and evolutionary process models. For example, SCRUM is an agile software development process for small teams that is based on a series of short development iterations, or sprints, which aim to deliver the product incrementally (Rising and Janoff 2000). Extreme Programming (XP) is also a well-known agile methodology that supports the creation of software in very short iterations. Test-Driven development appeared within the rise of agile methodologies and was earlier described as a XP practice necessary for analysis, design, and testing. In fact, Extreme Programming recommends programmers to adopt Test-Driven Development, among other practices like frequent releases or pair programming (Chromatic 2003). Due to its relevance in the present work, Test-Driven Development is detailed in Section 2.2.3.

2.2.2 Software testing

Software testing allows the dynamic verification of an application based on a set of tests, with the intent of finding faults or defects (IEEE Computer Society 2004; Myers 2004). According to this definition, the fact that the program is executed distinguishes testing from code inspection, in which the source code of an application is typically read by developers and analyzed in a static manner (and is not executed) (Myers 2004). A particular type of testing is robustness testing, which is detailed in Section 2.3.3.

Despite being a simple definition, testing software can be a quite complex process, involving different techniques and different testing granularities. With the increasing complexity of software systems, tests tend also to become more relevant, taking increasing amounts of time in the software development process.

An object that plays an important role in development and testing is the software specification. In general, this artifact defines correct behavior so that incorrect behavior is easier to identify. With the complexity of current systems, a specification can detail a system in different levels. A software system can be seen as an aggregation of subsystems and to specify and design it, we must specify desired external interactions, specify a decomposition (we specify the system's components and, for each component, we can specify the way it interacts with its environment to

realize overall system behavior), and allocate interactions with the environment to specific component interactions (Wieringa 1998).

In the scenario described above, it is easy to understand that many issues can arise during and after development. Problems can appear simply because the specification was produced in an incorrect manner, resulting in a document (or other type of artifact) with errors. Other issues include ambiguous specifications that can be interpreted in multiple ways, or simply correct specifications, but poorly understood by developers. An illustrative example of these situations can be found in the web services protocols specifications themselves (Curbera et al. 2002). There are issues in these specifications (e.g., ambiguities, errors) that keep the interoperability goal of web services from being fully achieved. To counteract such issues, the Web Services Interoperability Organization (WS-I) emerged and concentrated efforts in eliminating ambiguities, omissions, and errors from the web services specification documents. However, even if a perfect specification exists, implementation issues can still make their way into the final software product, justifying the need for having one or more testing phases.

Software testing can be classified according to the visibility that the tester has of the code being tested, being the tests designated as **black-box** or **white-box**. The testing activity can also be classified according to the granularity of the tests: **unit, integration**, or **system testing**, which in simple terms respectively translates from fine-grained to coarsed-grained tests, in what concerns the complexity of software and the number of components involved in the tests (Whittaker 2000).

White-box testing (Ostrand 2002a), also known as structural testing, refers to methods that have knowledge about the internal structure of the software. Thus it requires access to the source-code and information about the software's internals, such as data structures used, branches, functions, etc. In practice, this information may not always be available for developers, particularly in services environments, where the use of third-party software or services is common.

Black-box testing (Ostrand 2002b) is also known as functional testing and is based on the selection (or generation) and execution of tests without knowledge of the software's internals. This type of testing can use the specification of the software being tested. In simple terms, the software specification is examined (for instance, a tester can examine a WSDL document of a web service, or the specification document for Java

Message Service middleware) and tests are created in a way that the software's external functions are exercised. In order to select or create the tests, developers or testers can consider the program's purpose (if available), the input and output parameters, possible uses of the software and the ways it may fail (Ostrand 2002b; Myers 2004).

Tests can be performed at **different levels** during the development process. In practice, the tests target can vary in terms of granularity or specificity, focusing on a single module, a group of modules (associated by purpose, use, behavior, or structure), or a complete system. **Unit testing** is the finest level of testing, in the sense that it aims to verify the isolated execution of software pieces, which can be tested in separate. Obviously, the size of such units may vary, depending on the application being tested. This activity typically occurs with access to the code being tested (IEEE Computer Society 2004). A large number of unit testing frameworks are currently available for developers to create and run sets of test cases, e.g., JUnit (http://junit.org/), CppUnit (http://sourceforge.net/projects/cppunit/), or JUnitEE (http://www.junitee.org/).

Integration testing has a different target when compared to unit testing. In integration testing the goal is to verify the interaction between software components (that could have been tested using unit testing). Bottom-up or top-down approaches can be followed (for hierarchically structured software); however, current integration strategies are more architecture-driven and tend to integrate components by function. This type of testing is typically not concentrated on one stage of the software development process, but currently it is performed continuously, being also frequently referred to as continuous integration (IEEE Computer Society 2004; Duvall, Matyas, and Glover 2007).

In terms of test target granularity, **system testing** is a more coarsed-grained activity, as it aims to test the behavior of the whole system. This testing activity can also be adequate to compare the actual system being tested to non-functional requirements, such as security or performance (IEEE Computer Society 2004) and is frequently the support for executing acceptance tests. These consist of comparing the program to its initial requirements and the current needs of the end users. They are frequently executed by the customer or end user and usually are not considered the responsibility of the development organization (Myers 2004).

2.2.3 Test-driven development

Test-Driven Development (TDD) (K. Beck 2003) is an agile software development technique based on predefined test cases that define desired improvements or new functions (i.e., automated unit tests that specify code requirements and that are implemented before writing the code itself). Development is conducted in short iterations in which the code necessary to pass the tests is developed. Code refactoring is performed to accommodate changes and improve code quality. Despite this, defining test cases that guarantee high coverage is quite difficult and developers tend to focus on positive test cases (i.e., tests that do not try to break the feature, but simply to demonstrate that it works on normal situations) and often disregard negative test cases, such as the ones targeting robustness validation.

TDD has received a considerable amount of attention in recent years. A singular characteristic of TDD is the fact that tests are defined before the actual functionality development takes place. Tests and development proceed in an iterative fashion (in short iterations) and the combined whole makes the development task more productive, reduces response time to requirements changes, and facilitates continuous regression testing.

The tests specify the requirements and contain assertions that can be true or false. Running the tests allows developers to quickly validate the expected behavior as code development evolves. It is important to emphasize that Test-Driven Development is a software design method and not simply a testing technique. Tests actually drive the development and are continuously applied by developers to validate if the implementation fulfills the software requirements (that are specified by the tests themselves). As in other agile software development techniques, the short iterations facilitate managing requirements changes (K. Beck 2003). TDD is also frequently applied in projects whose goal is to improve legacy code developed with older techniques and technologies (Feathers 2004).

Test-driven development is an iterative process that consists of following a set of well-defined steps (based on the book "Test-Driven Development by Example") (K. Beck 2003):

- **Step #1 – Add a test**: the process starts by writing or modifying a test (or set of tests) that validates the feature to be developed or

modified. Obviously, if executed immediately this test must inevitably fail as it is written before implementing the corresponding feature (if the test does not fail then either the feature has already been implemented or the test is not correct and does not specify/validate the required feature). The developer must clearly understand the requirements before writing the test. Use cases (Bittner and Spence 2003), user stories (Cohn 2004), and other requirements specification techniques can be used as support for understanding the requirements. It is important to emphasize that predefined tests force the developer to focus on the requirements before writing the code. This is different from typical unit testing in which the developer writes tests after developing the code. Obviously, in this case, tests are strongly influenced by the code that has been previously developed.

- **Step #2 – Run all tests and verify if the new ones fail**: the goal is to validate that the testing infrastructure is working correctly and that the new tests effectively fail (and do not incorrectly pass without requiring any new code). Obviously, the new tests should fail for the expected reason (i.e., because the feature has not been implemented before) and not due to problems in the testing infrastructure or development environment. Note that this step partially validates the tests as it guarantees that the new tests will not pass until the required code is implemented.
- **Step #3 – Write code**: the developer writes the code required for the tests to pass. In many situations, the new code written in this step is not final and may, for example, pass the tests in a clumsy manner. However, the code will be improved at later steps. An important aspect is that it is important to guarantee that the code written is meant only to pass the tests and no additional (and therefore untested) functionality should be considered.
- **Step #4 – Run the automated tests**: if all test cases pass then the code meets all the tested requirements. If some tests fail, then the development process should go back to step 3. Note that, in some exceptional cases, developers may realize at this step that the tests created do not effectively describe the intended requirement (in the same way tests can be used to verify the code, code can be used to verify the tests). In this case, the tests should be revisited and the process goes back to step 1.

- **Step #5 – Refactor code**: after passing the tests, the code is ready to be cleaned up to improve quality (e.g., improve readability, remove duplication). To guarantee that the refactoring process does not harm the functionality, developers should re-run the test cases as needed. In fact, the refactoring process can be considered complete only after all test cases complete successfully.

A new iteration starts with another new test (or set of tests) for a new feature (or set of features). The size of the steps should be the one the developer feels comfortable with. Typically, less experienced developers use shorter cycles that get larger as they get more confident.

Robustness testing is a technique that is to be applied after development, which implies that, in its original form, it is not used along with TDD. However, considering the benefits of both, it can be beneficial to provide developers with a way of using the robustness testing approach in a Test-Driven Development context.

2.3 Dependability assessment

Dependability can be defined as the ability to deliver service that can justifiably be trusted (Algirdas Avizienis et al. 2004). The capability of a system to avoid service failures is thus crucial to the system's dependability itself. Robustness is a particular aspect of dependability and can be defined as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions (i.e., external faults) (IEEE 1990; Algirdas Avizienis et al. 2004). The following paragraphs present a brief overview of basic dependability-related notions (the concepts presented are, in most part, adapted from (Algirdas Avizienis et al. 2004)).

A **failure** occurs when the service being delivered by a given system deviates from fulfilling the system's goal (i.e., it deviates from correct service). The distinct ways in which such deviation is manifested correspond to the system's service **failure modes**. The failure itself can be characterized according to the domain, and in this sense, we distinguish the following two types of failures:

- **Content failures:** the content of the information that is being delivered by the service does not implement the service function

correctly (e.g., the client obtains a service response that deviates from the service's regular output domain).

- **Timing failures:** the time of arrival of the service response deviates from implementing the system function (i.e., the timing of service delivery is not adequate to the client's requirements).

An **error** is the part of the total state of the system that may lead to its subsequent service failure. The cause for an error is called a **fault**, which can be internal or external to the system. In particular, a **vulnerability** is an internal fault that enables an external fault to harm the system (i.e., to cause an error and possibly a failure).

A system may provide multiple services (e.g., a web service provides multiple operations to its users). When one or more of the services that implement a complete system fail, it may leave the system in a **degraded mode**, where a part of the service is still available to the end-user. Such modes may be directly identified by the system's specification (e.g., slower performance, limited concurrency, etc.).

Assessing computer dependability has been addressed in the dependability research community by using both **modeling** and **empirical** techniques. The former include *analytical* (Trivedi et al. 1994) and *simulation* (Jenn et al. 1994) models, and the latter include *field measurement* (J. Gray 1990), *fault injection-based* techniques (Segall et al. 1988; Han, Rosenberg, and Shin 1993; Carreira, Madeira, and Silva 1998; Hsueh, Tsai, and Iyer 1997) and *robustness testing* (D. P. Siewiorek et al. 1993; P. Koopman et al. 1997; Mukherjee and Siewiorek 1997; P. Koopman and DeVale 1999; Rodríguez et al. 1999).

Dependability benchmarking, mainly inspired on experimental dependability evaluation techniques, is a concrete example of an approach that is used to assess and compare systems in terms of dependability and shares similar concepts with robustness testing. The next sections present basic fault injection concepts (a key tool for evaluating the robustness of systems), introduce the building blocks of dependability benchmarking (due to its relevance in the context of robustness testing, many times referred as robustness benchmarking), and describe recent techniques, used in the services context, with the purpose of assessing the robustness and Quality of Service of web and messaging services.

2.3.1 Fault injection

Fault injection has become an important tool when the goal is to evaluate dependability properties. In general, fault injection can be used to study a system's behavior in the presence of faults, evaluate the error handling mechanisms, or evaluate fault tolerance mechanisms (e.g., reconfiguration schemes) and performance loss. The following concepts are mostly adapted from (Hsueh, Tsai, and Iyer 1997).

Faults can be injected in two different levels of the system being tested, depending on the experimental goal: hardware or software. **Hardware fault injection** can be grouped in two classes: with and without contact. This essentially means that in the former case the system or subsystem that injects the faults (i.e., the fault injector) has direct contact with the system under test and can produce, for instance, voltage changes. In the latter case, there is no physical contact with the system being tested, however, faults can be injected by using environmental changes like electromagnetic interference or heavy-ion radiation.

In **software fault injection**, faults are injected at the software level and these techniques can be classified in two categories: compile-time and runtime. In **compile-time fault injection** the application's instructions are modified before the application image is loaded and executed. This method injects errors into the source code or assembly code of the target program to emulate the effect of faults. On the other hand, **runtime fault injection** techniques can be divided depending to the type of fault emulated: hardware faults (software-based techniques used to introduce disturbances that emulate hardware faults, such as bit-flip or stuck-at), and software faults (software-based techniques are used to introduce changes that emulate program defects or bugs). Also, runtime fault injection techniques can be classified according to mechanism used to trigger the injection itself. Such mechanisms include:

- **Time-out:** this method injects faults based on time, as opposed to inject faults according to specific events or system state. The results can be unpredictable and, in result, experiments using this technique may have low reproducibility. In practice, this results in the need for conducting a large number of experiments so that statistically representative results can be obtained.

- **Exception/Trap:** in this case faults are injected when a hardware exception or software trap occurs. This may happen only when certain events or conditions are present.
- **Code insertion:** this technique adds specific instructions to the target application, that allow for fault injection to occur before specific instructions (e.g., before a method call). In practice, this can be similar to the compile-time code modification technique, however code insertion injects faults at runtime and typically instructions are added, rather than modified.

In general, each method has advantages and disadvantages, so the testing goals, the practicality of the technique (in the context of the overall experiment), the technique cost, etc., are all aspects that can ultimately influence the selection of the fault injection method.

Fault injection literature is rich in works on physical (hardware) fault injection and emulation; however, only few studies have addressed the problem of injection of software faults. This can be explained by the fact that the knowledge on the software faults experienced by systems in the operational phase (i.e., in the field) is very limited, which makes the definition of meaningful sets of faults (or errors) to inject rather difficult. We focus on software faults in the following paragraphs due to their relevance in this thesis context, which mostly deals with problems that have a strong association with software problems. Hardware faults are not as relevant in this context, and therefore are not further detailed.

The emulation of software faults to assess the impact of residual bugs or validate software fault tolerance mechanisms is extremely important. In fact, several studies refer to software faults (J. Gray 1990; Sullivan and Chillarege 1992; I. Lee and Iyer 1995; Kalyanakrishnam, Kalbarczyk, and Iyer 1999) as a significant cause for computer failures and, given the huge complexity of today's software, the weight of software faults on overall system dependability will tend to increase.

Most of the studies on software faults have been related to the software development phase, as software faults are originated during the different steps of this phase (requirements, specification, design, coding, testing, etc.). This is an important area of software engineering and many studies have contributed to the improvement of the software development methodologies, with particular emphasis on testing, reliability modeling and reliability risk analysis (Lyu 1996; Musa 1998).

The study of the software faults during the operational phase (i.e., after the product deployment) is substantially different from the software development phase. In fact, The operational environment and the software maturity are different during the operational phase, and the software reliability should be studied in the context of the whole system (and not just in the context of a given application). The difficulties are not only in the instrumentation required to collect data on software faults but also in the fact that the software faults must be analyzed taking into account the system architecture (hardware and software) and not only software modules. Maybe these difficulties account for the fact that the number of works on software faults during the operational phase is lower than the studies available for the development phase.

Two significant studies of software dependability in Tandem systems are presented in (J. Gray 1990; I. Lee and Iyer 1995). The impact of software defects on the availability of a large IBM system is presented in (Sullivan and Chillarege 1991). An early study (R. K. Iyer 1995) investigated the effect of the workload on the reliability of an IBM operating system based on data collected from field.

An important contribution to promote the collection and study of observed faults is the Orthogonal Defect Classification (ODC) (Ram Chillarege et al. 1992). ODC is a classification schema for software faults (i.e., defects) in which defects are classified into non-overlapping attributes and used as a source of information to understand and improve the software product and the software development process. Although ODC is intended to provide feedback on to the development process, it also provides a useful defect classification concerning the problem of the emulation of software faults by fault injection.

Mutation testing is a specific form of fault injection that consists of creating different versions of a program (mutants) by making small syntactic changes (Budd 1981; DeMillo et al. 1988). Mutation can be considered as a static fault injection technique, as the source code is changed instead of the program/system's state, as happens in classical fault injection (considered dynamic fault injection in this view). Mutation has been largely used for software testing to identify the best set of test cases or to study the error propagation process from fault activation to an eventual wrong program output. An experimental comparison of the errors and failure modes generated by actual software faults and mutations is presented in (Daran and Thévenod-Fosse 1996). The results

favor the idea that mutations produce error patterns and erroneous program behaviors similar to actual software faults.

In (Voas et al. 1997) fault injection is proposed for the quantification of the risks created by the software component of a system (experimental software risk assessment). Given the difficulties of knowing what kinds of software faults are most likely to be hidden in the code and their probability of future manifestation, the results of fault injection cannot be used as an absolute measure of risk. Instead, the authors suggest the use of fault injection for the prediction of worst-case scenarios (in terms of software risks).

In (Wee Teck Ng and Chen 1999) random code corruption and faults meant to emulate specific programming errors were used to improve the design of a reliable write-back file cache. This paper is a quite convincing example of how SWIFI (Software Implemented Fault Injection) tools can be used to improve fault tolerance techniques during the design phase.

The problem of the accurate emulation of software faults by fault injection was first addressed by (Christmansson and Chillarege 1996; Christmansson and Santhanam 1996). These studies propose the same basic set of rules for the generation of errors that emulate software faults. These rules were obtained from the analysis of field data about discovered software faults that have been classified using ODC. In the former paper (Christmansson and Chillarege 1996) the rules for error generation for fault injection were proposed for fault forecast and the latter paper (Christmansson and Santhanam 1996) addresses the problem of error generation for fault removal.

(Christmansson and Chillarege 1996; Christmansson and Santhanam 1996) are seminal papers in this topic, as they give a first contribution to the solution of the problem of emulation of software faults by fault injection. Nevertheless, they also raised some new questions related to the accuracy of the emulation and on the adequacy of existing SWIFI tools to perform the injection of software faults.

An experimental study on the emulation of software faults by fault injection is presented in (Madeira, Costa, and Vieira 2000). In a first experiment, a set of real software faults has been compared with faults injected by a SWIFI tool (Xception) to evaluate the accuracy of the injected faults. Results revealed the limitations of Xception (and other SWIFI tools) in the emulation of different classes of software faults (about 44% of the software faults could not be emulated). The use of field data about real

faults is discussed and software metrics are suggested as an alternative to guide the injection process when field data is not available. In a second experiment, a set of rules for the injection of errors meant to emulate classes of software faults is evaluated. The fault triggers used seem to be the cause for the observed strong impact of the faults in the target system and in the program results. The results also show the influence in the fault emulation of aspects such as code size, complexity of data structures, and recursive versus sequential execution.

As SWIFI tools are not the best choice to emulate software faults, (Duraes and Madeira 2002) proposes a new technique named G-SWFIT (Generic Software Fault Injection Technique), which represents the first technique proposed for the injection of software faults. When compared to SWIFI tools, G-SWFIT follows a completely different approach. It works at the object code level (i.e., executable code) and it consists of finding key programming structures (code patterns) at the machine code-level where high-level software faults can be emulated. As soon as these programming structures are identified the injection of the faults consists of applying mutated patterns directly in the object code. To make the technique practical, G-SWFIT is heavily based on a library of machine-code level structures (or patterns) and possible software faults that, once introduced in such programming structures, can emulate specific classes of high-level software faults.

One important aspect related to the representativeness of the faults injected using the G-SWFIT technique is the choice of the high-level software faults that are described in the G-SWFIT library. (Duraes and Madeira 2003) analyzed several sources and conducted an extensive field study in order to build the list of bugs that can reasonably be expected to occur frequently. These faults are called “educated mutations” to emphasize that they include inputs from experience and field data on real software faults. Based on this field analysis, (Duraes and Madeira 2004) proposes a set of guidelines for the definition of faultloads based on software faults for dependability benchmarking.

An analysis of how software faults can be injected in a source-code independent manner is presented in (Durães and Madeira 2006). In particular, the authors address important aspects such as fault representativeness and emulation accuracy. A large set of real software faults was analyzed and found to fit in well-defined classes and can be characterized very precisely, which allows the injection of software faults using a small set of emulation operators. The fault emulation accuracy of

the fault injection technique used (G-SWIFT) is shown and key aspects, that may influence it, are discussed.

Aspect Oriented Programming (AOP) (Kiczales et al. 1997) is a technique that allows the injection of code in applications, and has recently gained attention in the fault injection and testing domains (Copty and Ur 2005; Rui Wang and Ning Huang 2008; Coelho et al. 2006). A large number of AOP frameworks have been created (Kersten 2005), however AspectJ is currently very popular among developers (Eclipse Foundation 2008).

The use of AOP involves understanding a few key concepts (SpringSource 2010):

- **Aspect:** a concern that cuts across multiple objects.
- **Join point:** a point during the execution of a program (e.g., the execution of a method or the handling of an exception).
- **Advice:** action taken by an aspect at a particular joinpoint. Types of advice include: 'before', 'after', and 'around' (i.e., before and after).
- **Pointcut:** a predicate that matches join points. An advice is associated with a pointcut expression and runs at any join point matched by the pointcut (e.g., the execution of a method with a certain name).

By using AOP we can inject crosscutting concerns into any application in a non-intrusive way. For instance it is possible to create a generic fault injector and then use AOP to merge this injector with a target application, enabling the injection of faults at runtime. The merging process consists simply of using the target application and the injector (or any other kind of application that benefits from using cross-cutting concerns) and then use an AOP compiler (frequently known as weaver) to merge both. Figure 2.4 illustrates this procedure.

Using the setup depicted in Figure 2.4, it is possible to inject faults, for instance, when specific methods are called in the target application and the technique requires no source code access (bytecode access is sufficient). It is possible to use this setup if, for instance, we want to inject faults every time a web service operation is called. As we will see in the following chapters, we have used this technique through the work presented in this thesis. The main reason for this is that AOP provides a

fast, non-intrusive and developer-transparent technique that enables the addition of crosscutting features to applications in an easy way.

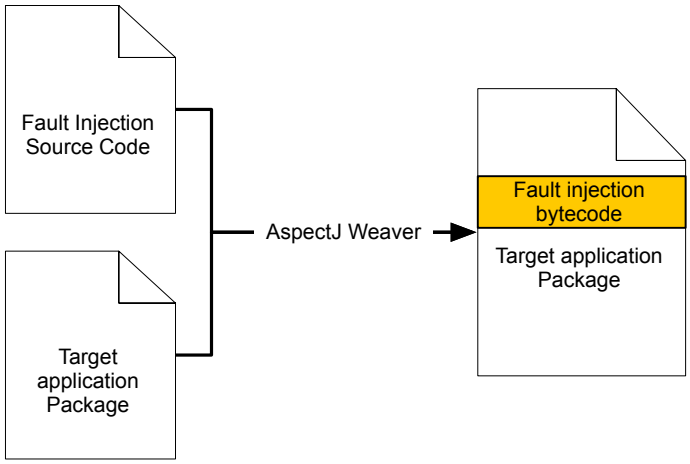


Figure 2.4 – Integrating fault injection capabilities in an application.

2.3.2 Dependability benchmarking

Performance benchmarks, widely known in the computer industry, are standard procedures and tools that aim to assess (for comparative purposes) different systems or components in a specific domain (e.g., databases, operating systems, hardware, etc.) according to specific performance measures. There are two major organizations in the performance benchmarking business: TPC (Transaction Processing Performance Council) and SPEC (Standard Performance Evaluation Corporation).

TPC and SPEC proposed key benchmarks in the services domain. The TPC Benchmark App (**TPC-App**) (Transaction Processing Performance Council 2008) is a performance benchmark for web services infrastructures. This benchmark simulates the activities of a business-to-business transactional application server, a retail distributor on the Internet with ordering and product browsing. The application represented by the benchmark accepts incoming web service requests from other businesses (or a store frontend) to place orders, view catalog items and make changes to the catalog, update or add customer information, among others. The benchmark measures the average number of Service

Interactions Per Second (SIPS) completed by the system during the measurement interval.

The SPEC **jAppServer2004** benchmark (Standard Performance Evaluation Corporation 2007a) was designed to measure the performance of J2EE 1.3 application servers (SPEC holds similar benchmarks for previous versions of J2EE). JOPS is the performance measure provided and corresponds to the jAppServer Operations Per Second completed during the measurement interval. In addition to standard benchmarks like TPC-App and SPEC jAppServer2004, main vendors have recently undertaken efforts towards the comparison of web services performance (Microsoft Corporation 2004; Sun Microsystems Inc. 2004). Regarding the messaging middleware domain, SPEC released the **SPECjms2007** benchmark (Standard Performance Evaluation Corporation 2007b), whose goal is to evaluate the performance of enterprise message oriented middleware servers based on JMS.

A dependability benchmark is a specification of a standard procedure to assess not only the performance, but also the dependability of computer systems or components. The main goal is to provide a standardized way to assess and compare different systems or components from a dependability point-of-view. (Kanoun and Spainhower 2008) presents a large set of dependability benchmarks proposed by the industry and academia and explains the various principles and concepts of dependability benchmarking. It also provides a large set of examples and recommendations for defining dependability benchmarks.

Comparing to typical and well-known performance benchmarks, such as the TPC and SPEC benchmarks, which consist mainly on a workload and a set of performance measures, a dependability benchmark adds two new elements: 1) measures related to dependability; and 2) a faultload that emulates real faults experienced by systems in the field (Vieira 2005).

In addition to the measures, workload, and faultload, a dependability benchmark specification must also clearly define the procedure and rules that have to be followed during the benchmark implementation and execution. Furthermore, the components of the experimental setup needed to run the benchmark have also to be described. This way, the main elements of a dependability benchmark are:

- **Measures**: characterize the performance and dependability of the system under benchmarking in the presence of the faultload when executing the workload. The measures must be easy to understand

and must allow the comparison between different systems in the benchmark domain.

- **Workload:** represents the work that the system must perform during the benchmark run. The workload emulates real work performed by systems in the field.
- **Faultload:** represents a set of faults and stressful conditions that emulate real faults experienced by systems in the field.
- **Benchmark procedure and rules:** description of the procedures and rules that must be followed during the benchmark implementation and execution.
- **Experimental setup:** describes the setup required to run the benchmark.

Classic robustness assessment methodologies can be seen as form of a dependability benchmark, as it shares similar components and also allows comparing systems based on a specific dependability attribute. In fact, several authors refer to robustness testing as robustness benchmarking, due to the similarity of the components required in both methodologies. The following section provides an overview of robustness testing methodologies found in the literature.

2.3.3 Assessing robustness

This section presents an overview of classic robustness assessment methodologies, i.e., techniques that assess the behavior of a given system when in presence of invalid inputs. Robustness testing (also referred to as robustness benchmarking in the literature) (Mukherjee and Siewiorek 1997), typically consists of applying a suite of robustness tests or stimuli, stimulating a target system in a way that triggers internal errors, and exposing both programming and design errors. Systems can be differentiated according to the number of errors uncovered.

Many relevant studies (D. P. Siewiorek et al. 1993; B. P. Miller et al. 1995; Carrette 1996; P. Koopman et al. 1997; Fabre et al. 1999) evaluate the robustness of software systems, nevertheless, (P. Koopman and DeVale 1999) and (Rodríguez et al. 1999) are the ones that present the most popular robustness testing tools, respectively Ballista and MAFALDA.

Ballista is a tool that combines software testing and fault injection techniques. The main goal is to test software components for robustness (P. Koopman and DeVale 1999), focusing specially on operating systems. Tests are created using combinations of exceptional and acceptable input values of parameters of kernel system calls. The parameter values are extracted randomly from a database of predefined tests and a set of values of a certain data type is associated to each parameter. The robustness of the target operating system (OS) is classified according to the CRASH scale that distinguishes the following failure modes: *Catastrophic* (OS becomes corrupted or the machine crashes or reboots), *Restart* (application hangs and must be terminated by force), *Abort* (abnormal termination of the application), *Silent* (no error is indicated by the OS on an operation that cannot be performed), and *Hindering* (the error code returned is not correct).

Initially, Ballista was developed for POSIX APIs (including real time extensions). Further work has been developed to adapt it to Windows operating systems (Shelton, Koopman, and Devale 2000). In this study the authors present the results of executing Ballista-generated exception handling tests over several functions and system calls in Windows 95, 98, CE, NT, 2000, and Linux. The authors were able to trigger system crashes in Windows 95, 98, and CE. The other systems also revealed robustness problems, but not complete system crashes. Ballista was also adapted to various CORBA ORB implementations (Pan et al. 2001). For the CORBA ORB implementations the failure modes were adapted to better characterize the CORBA context.

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) is a tool that allows the characterization of the behavior of microkernels in the presence of faults (Rodríguez et al. 1999). MAFALDA supports fault injection both into the parameters of system calls and into the memory segments implementing the target microkernel. However, in what concerns to robustness testing, only the fault injection into the parameters of system calls is relevant. MAFALDA has been posteriorly improved (MAFALDA-RT) (Rodríguez, Albinet, and Arlat 2002) to extend the analysis of the faulty behaviors in order to include the measurement of response times, deadline misses, etc. These measurements are quite important for real-time systems and are possible due to a technique used to reduce the intrusiveness related to the fault injection and monitoring events. Another study has been carried to extend MAFALDA in order to

allow the characterization of CORBA-based middleware (Marsden and Fabre 2001).

The success of robustness testing on operating systems was so great that studies have appeared on particular components of OSs. In (Mendonça and Neves 2007) a study on the robustness properties of Windows device drivers is presented. Recent OSs kernels tend to become thinner by delegating capacities on device drivers (which presently represent a substantial part of the OS code, since nowadays computer interact with many devices), and a large part of system crashes can be attributed to these device drivers because of residual implementation bugs (Mendonça and Neves 2007). Considering this, the authors of this study present an evaluation of the robustness properties of Windows (XP, 2003 Server and Vista) concluding that in general these OSs appear to be vulnerable, and that some of the injected faults can cause systems to crash or hang, which highlights the importance of robustness testing.

One of the first examples that can be found in the literature using invalid or mutated data to test web services is (Jeff Offutt and Xu 2004). In this work web services are tested using data perturbation, which consists of modifying values in request messages and analyzing the responses. The types specified in the XML schema or Document Type Definition (DTD) documents (W3C 2008a; Bray et al. 2000) are used as basis to generate test cases. In (J. Offutt, Wuzhi Xu, and Juan Luo 2005) a similar approach is presented. Still, the coupling that is done to the XML (eXtensible Markup Language) technology invalidates possible test generalizations (i.e., it does not apply to other technologies as the approach is tightly coupled to XML).

An approach for testing web services using mutation operators is presented in (Siblini and Mansour 2005). This paper proposes a technique to test web services using parameter mutation analysis. The web services description file (defined using WSDL) is parsed initially and mutation operators are applied to it resulting in several mutated documents that will be used to test the service. In spite of the effort set on this approach, these parameter mutation operators are very limited and consist basically in switching, adding, deleting elements, or setting complex types to null.

A technique for generating test cases based on the WSDL specification is presented in (Xiaoying Bai et al. 2005). The specification is parsed and test cases are generated considering: test data generation, test operation generation, operation flow generation, and test specification generation.

Test data is generated by analyzing messages based on XML schema data type definition and considers simple, complex, aggregated and user-defined data types, complex data type, aggregated data type and user-derived data type. Although this represents a fairly complete effort, the framework is too complex to setup and up until now, no adoption of the authors' tests specification language has been found in the literature.

WebSob is a tool for robustness testing of web services (E. Martin, Basu, and Xie 2007). This tool requires code access as it uses JCrasher to generate unit tests for Java classes. The implementation is designed to work for the Java programming language, preventing tests from being executed in other programming languages. The analysis of responses is quite limited focusing on three types of HTTP status codes: 404 File Not Found, 405 Method Not Allowed, and 500 Internal Server Error (Fielding et al. 1999). A service hang is also considered whenever a service takes more than 30 seconds to reply.

A distinct approach enabling a testing methodology for white-box coverage testing of error handling code is presented in (Fu et al. 2004). The approach uses a compile-time analysis that allows compiler-generated instrumentation to control a fault injection process and keep track of the exercised recovery code. The approach is designed for Java applications and it was tested with applications that can be used within a network; namely, an FTP server (FTPD), a File Server (JNFS), Haboob (a web server), and Muffin (a proxy server). The methodology used leaves, on average, approximately 16% of the exception-catch links uncovered, which translates into the need of further examination by a human tester. The limitation is that the tool requires code access and focuses on web services written in Java.

Considering that the quality of a composed service may depend on the ability of its component services to react to unforeseen situations, such as data quality problems and service coordination problems, an approach is proposed in (Fugini, Pernici, and Ramoni 2009) to analyze the quality of composed services using fault injection techniques. The technique is based in two main aspects; in particular, the reactions of the composed service to data faults and the effect of delays on composed services. The data faults include substituting digits in numbers, reversing characters in strings and dates, representing dates in different formats. However, limit conditions have not been considered in this study (for instance like the ones used in our work that consider data domains and explore their limits).

A framework (based in our own published work) for the robustness testing of service compositions built with WS-BPEL is presented in (Seung Hak Kuk and Hyeon Soo Kim 2009). The composition of multiple services may result in a compound service that carries more robustness problems than a single service, due to the complexity usually involved in this kind of environments. The execution of tests using a number of faulty conditions or exceptional cases is then crucial to build robust compositions of services. However, testing the service composition for robustness is more difficult than testing regular applications because the number of test scenarios and the cost of testing greatly increase with the increasing number of component services. The testing framework presented introduces a virtual service, which corresponds to a real component service. It simulates abnormal situations within the real service and allows the verification of the robustness of web services compositions against various errors and/or exceptional cases. The fact that this approach extends our own work illustrates the extensibility or adaptability of the approach presented in this thesis for testing single services, and its utility in testing more complex scenarios.

Regarding messaging middleware, there are several studies that focus on performance or Quality of Service (QoS) assessment. In (Crimson Consulting Group 2003) the performance of two popular JMS implementations (Sun Java System Message Queue and IBM WebSphere MQ) is assessed using multiple configurations, including both point-to-point and publish/subscribe environments. Additionally, tests are executed with and without persistence, a basic feature that can help deliver a message in the event of a provider failure. Performance testing has also been applied in publish/subscribe domains in the presence of filters (Menth et al. 2006). The drive behind this study is the fact that if filters are set on the JMS provider, the provider can be used as a routing platform; however there is no guarantee that the provider's message throughput is enough to support large-scale systems. The authors test the message throughput capacity of FioranoMQ, SunMQ, and WebsphereMQ and show that filtering has a significant impact on performance.

In (Chen and Greenfield 2004) is presented an empirical methodology for evaluating the QoS of JMS implementations. Several test scenarios are presented and metrics are defined for evaluating the performance (throughput and latency) and message persistence capabilities of JMS providers. The experimental evaluation shows differences between the two JMS providers tested in overall performance and Quality of Service

(QoS) attributes, in particular persistence message recovery was tested in the presence of distinct types of server failures (server kill, reboot, and power off).

An approach to evaluate the performance of Message Oriented Middleware platforms is presented in (Sachs et al. 2009). The authors use the SPECjms2007 (Standard Performance Evaluation Corporation 2007b) standard benchmark as basis for their approach and make the following contributions: 1) the workload used in the benchmark is characterized in detail to provide users with a better understanding of its internal components; 2) it is shown how the workload can be customized to assess different aspects of the middleware performance; 3) a case study using the BEA WebLogic application server is presented and a performance analysis is carried out using several workload and configuration scenarios. The approach and results presented can be used to define a workload configuration that selectively targets different aspects of the middleware and can help understanding official benchmark results.

Despite the fact that several messaging middleware implementations have been studied for performance and Quality of Service, up until now no strategy has been developed that can be used to assess the robustness properties of messaging middleware. Similarly, the approaches presented for testing the robustness of web services carry limitations in several dimensions. These include the limited set of mutations or faults, immature or reduced classification schemes for the tests results, or lack of extensibility of the approaches.

2.4 Robustness improvement and fault tolerance

This section introduces several wrapper-based techniques designed to increase the robustness of software systems. Additionally, we examine several approaches to detect and remove security vulnerabilities, with emphasis on frequently observed issues in software services. Finally, we summarize the state of the art in fault tolerance techniques (aiming to avoid service failures in the presence of faults (Algirdas Avizienis et al. 2004)) that can be helpful in avoiding or mitigating content and timing failures. In particular, we present diversity based techniques that can be used to select the most adequate response to a given service request, but

we also focus on redundancy as a means to mitigate possible failures of a given service that can result in no response being delivered to a requesting client.

2.4.1 Removing robustness issues

Several approaches aiming at increasing the robustness of systems by the use of wrappers have been proposed in the past. Classically, a wrapper is an object that has (at least) a similar interface to the object it contains. Calls to the wrapper are relayed to the target object, and the wrapper can add its own functionality before or after the call. This provides flexibility since we can change the functionality at the wrapper without changing the contained component (Gamma et al. 1994).

The HEALERS toolkit (Fetzer and Zhen Xiao 2003) can be used to enhance the robustness and security of applications. The goal is to protect applications from errors related to C library functions. It does so by intercepting calls to the C library, based on the argument that a large quantity of software failures is caused by C library API failures (function calls with invalid arguments). HEALERS also uses automated fault-injection to discover robustness and security problems in third-party software. It makes use of this information to generate a set of fault-containment wrappers that try to correct most of these problems. This wrapper generation is flexible and highly configurable, and the generated wrappers basically sit between a given application and its shared libraries.

Although a very interesting approach, HEALERS revealed a few deficiencies regarding the extensibility and performance of the toolkit. This way, further work has been presented in (Susskraut and Fetzer 2007) to address those issues. The main contributions are a fault injection tool that makes use of static analysis to perform fault injection on arbitrary functions and a table-based approach that generates protection hypotheses (based on observed behavior) for the protective wrappers. These hypotheses are basically Boolean expressions over predicates (called checks) on the arguments to a function.

In (Popov et al. 2001) authors propose a wrapper-based technique for protecting off-the-shelf components. The authors describe a general approach for the development of protective wrappers at application level. These wrappers fight regular problems found in off-the-shelf components: incomplete or incorrect component specifications, non-standard features

that may be provided by the component for the complete system, and the output of incorrect commands (from the component) to the environment. In this study traceability techniques are used to collect information for the wrappers development, rigorous models are developed for the analysis of the improvement gain, and implementation techniques are provided for current technologies. The authors argue that developers should form a view of what the components and the whole system should do (designated as 'acceptable behavior constraints'). This view has to be formally and systematically developed and should be integrated in the protector.

In (Ghosh, Schmid, and Hill 1995) is discussed an approach to assess the robustness of Windows NT software and to provide robustness wrappers for off-the-shelf software. Software wrappers are developed in this context to handle operating system failures gracefully, so that catastrophic application failures are diminished.

The Service Modeling Language (SML) provides a powerful way for creating models of complex services (Pandit, Popescu, and Smith 2009). These models may include information such as configuration, deployment, service level agreements, etc. SML uses Schematron and XML Schema (Vlist 2007; W3C 2008a). Schematron and XPath functions (W3C 1999) are used for the language's rules, which are Boolean expressions that constrain the structure and content of documents. SML scenarios require, however, several features that either do not exist or are not fully supported in XML Schema. The complexity of SML is its large disadvantage. When the goal is to improve web services robustness, it can be useful to keep simplicity standards conformity as top requirements.

The Bean Validation Java Specification Request (Bernard 2010) is a recent effort to specify class level constraint declaration and validation facilities for Java developers. Bean validation is a new validation model available as part of Java EE 6 platform and is supported by constraints in the form of annotations set on fields, methods, or classes of a JavaBeans component, such as a managed bean (Jendrock et al. 2011). Input validation is indeed an essential tool when the goal is to protect software components against invalid input parameters.

Validation wrappers and the validation models and techniques presented are in fact well-known solutions to improve the robustness of systems. However, research on robustness improvement is still an open issue in software services, as it needs to target their specific characteristics.

2.4.2 Detecting and removing vulnerabilities

Robustness is the response of a system to external faults, in particular invalid inputs. These external faults may also be malicious inputs (although they may be valid from a domain point of view) that can be used to exploit security vulnerabilities (internal faults that enable external faults to harm the system) (Algirdas Avizienis et al. 2004). When an application does not check these malicious inputs the system may be harmed if vulnerabilities are present. In fact, unchecked inputs are widely recognized as the most common reason for security vulnerabilities in web applications.

Different techniques for the identification of security vulnerabilities have been proposed in the past (Stuttard and Pinto 2007), including:

- **Static vulnerability scanning**: consists in analyzing the source code of the application looking for potential vulnerabilities. It is a "white-box" approach that can be done manually or by using automated code analysis.
- **Penetration testing**: widely used technique that tries to disclose security vulnerabilities in web applications (including web services). The testing tool stresses the application from the point of view of the attacker ("black-box" approach) and tries to penetrate it by issuing a huge amount of interactions.

Typical white-box and black-box approaches used to test web applications for vulnerabilities, also apply to web services. Concerning white-box approaches, various static code analysis tools have been developed (Stuttard and Pinto 2007). This type of analysis consists in inspecting the program's source code in a static manner (i.e., without executing the program) to detect code patterns that are prone to vulnerabilities. A static analysis technique capable of detecting many application vulnerabilities, including SQL injection, is presented in (Livshits and Lam 2005). FindBugs (Hovemeyer and Pugh 2004) is a concrete example of a well-known tool that uses static analysis to inspect Java code for occurrences of bug patterns, including SQL Injection vulnerabilities. Other examples of security tools used by web applications/services developers to detect security vulnerabilities include FORTIFY, Pixy, or PHP-sat (Fortify Software 2011; Jovanovic 2011; Bouwers 2011).

As previously referred, black-box testing does not require the application's source code. Instead it is based on the execution of a set of runtime tests, where a usually large number of requests is created and delivered to the service under testing. The responses are later analyzed to disclose any existing vulnerabilities. This type of security testing is also known as penetration testing, and currently there is a wide range of tools (vulnerability scanners and fuzzers) that can be used to detect vulnerabilities in web services. Some examples include commercial vulnerability scanners such as Acunetix Web Vulnerability Scanner, HP Webinspect, IBM Rational AppScan, Burp Suite (Acunetix 2011; Hewlett Packard 2011; IBM 2011; PortSwigger 2011) and open source scanners such as Gamja and Wapiti (Jeon 2011; Surribas 2011).

Despite of the popularity and high number of penetration testing tools, research indicates that automatic approaches for vulnerabilities detection are frequently unable to produce accurate results (Vieira, Antunes, and Madeira 2009). Thus, human code inspections for vulnerabilities disclosure are frequently used to obtain more accurate results (Fagan 2002).

The identification of vulnerabilities is a key step when the goal is to deploy a secure application. However, it is also of extreme importance to have easy (automatic) ways to remove the disclosed vulnerabilities. In the particular case of SQL/XPath Injection vulnerabilities (predominant vulnerability types in web applications and services) a common approach is to change the vulnerable code and separate the query structure from the input data by using parameterized queries. Such queries are available for typical databases under the form of prepared statements (an SQL statement structure with placeholders for variables), but also for XML databases (or simply applications that use XPath) under the form of XPath parameterized expressions (Stuttard and Pinto 2007).

An automated approach that tries to convert plain text SQL statements into prepared statements is presented in (Thomas and Williams 2007). The strategy is to remove SQL vulnerabilities by replacing vulnerable code with generated secure code. The presented prototype was able to remove SQL injection vulnerabilities in five different statement configurations contained in five custom-built toy projects. The generated prepared statements were verified to be functionally equivalent to the original statements. However, the conversion algorithms are quite limited and need to be largely improved to reduce the large number of SQL statements that cannot be handled by the proposed approach.

The approach presented in (Thomas, Williams, and Xie 2009) describes a replacement algorithm and its corresponding automation for removing SQL injection vulnerabilities from SQL statements. The approach consists of replacing the SQL statements by secure prepared statements. Four case studies were conducted on open-source projects. Code inspection and static analysis were used to disclose code prone to SQL injection, which was then replaced by secure code automatically generated. The whole process was able to correct 94% of the vulnerabilities found in 20 files. However, several aspects related with non-explicit setting, non-string, or iterator-based SQL structure still remained unsolved. Overcoming these limitations, by not making assumptions about the structures used to build SQL statements can result in an advantage in a vulnerability removal or mitigation approach.

Based in the fact that software written in one language often needs to construct sentences in another languages (such as SQL, XQuery, or XPath queries), in (Bravenboer, Dolstra, and Visser 2007) is presented an approach for attack injection prevention, preventing vulnerabilities by construction (a programming style alternative to methods that use string manipulation or high-level APIs). The proposed methodology consists of embedding the syntax of the guest languages into the syntax of the host language (e.g., SQL in Java) and automatically generating code that maps the embedded language to constructs in the host language that reconstruct the embedded sentences, adding escaping functions where appropriate. Although, the approach is generic enough to be adapted to various languages, it obviously adds complexity to the development phase.

AMNESIA (Analysis and Monitoring for NEutralizing SQL-Injection Attacks) (W. G. J. Halfond and Orso 2006) is a tool that uses a model-based approach specifically designed to detect SQL injection attacks, and combines static analysis and runtime monitoring. Static analysis is used to analyze the source code of a given web application, automatically building a model of the legitimate queries that such application can generate. At runtime, AMNESIA monitors all dynamically generated queries and checks them for compliance with the statically generated model. When a query that violates the model is detected, it is classified as an attack, and is prevented from accessing the database. Although this approach uses static analysis as basis, learning the profile of legitimate queries at runtime may represent a richer, more realistic profile learning, overcoming the intrinsic limitations of the static analysis technique (e.g., requiring access to source code).

An anomaly-based system that learns the profile of normal database accesses in web-based applications is presented in (Valeur, Mutz, and Vigna 2005). This system uses a composition of several models that allow the detection of unknown attacks with reduced false positives. However, this approach is currently unable to display information about the coverage space of the training data, which impacts its effectiveness.

The work presented in this thesis focuses on protecting web services against SQL/XPath Injection attacks. The approaches presented above carry limitations associated with the types of SQL commands supported, complexity associated with the practical use of the approach, or the need to have access to the source code. Our approach mitigates the issues identified in the previous paragraphs, including the possibility of using such approach in legacy services, where no source code is available.

2.4.3 Redundancy and diversity

Application development is coupled to source code residual bugs (Jim Gray 1985). At runtime, these faults may be activated to produce an internal condition where the system deviates from the correct internal condition (i.e., it produces an error or multiple errors), and this can then evolve into a failure that can reduce a system's availability (Algirdas Avizienis et al. 2004). Web services are no exception, given the time-to-market circumstances that programmers frequently experience when developing this type of software.

Software bugs, a relevant cause of failure in software applications (Jim Gray 1985) may manifest consistently in well-known conditions, which means that they can be easily reproducible. Fault removal is the best option to deal with this type of bugs (by software tests, code-reviews, walk-troughs, etc.). On the other hand, other types of bugs can have complex causes resulting in apparently chaotic behavior or non-deterministic, and techniques as retry and replication are usually the best options to use. The key reason to use retry or runtime replication is that the fault activation conditions may be so difficult to replicate that a second opportunity may reveal successful on its own (Grottke and Trivedi 2007). Some of these transient bugs may also be related with aging processes (i.e., their failure manifestation increases with the number of executions or execution time), and for these cases, rejuvenation techniques apply (Grottke and Trivedi 2007; Huang et al. 1995).

When the goal is to maximize the availability of systems, replication is frequently the technique used. There are two basic replication schemes (Engelmann and Scott 2005):

- **Active/Passive:** in this type of configuration, also referred to as active/standby (servers are also frequently referred to as primary and secondary or backup), there is one primary node, which provides all available services to requesting clients, and a secondary server that usually remains idle and available to replace the primary server's functions when needed (i.e., when the primary server experiences a failure). Sub-forms of this scheme are:
    - **Active/cold-passive:** there is no replication of any component state. In case of a failure all component state is lost;
    - **Active/warm-passive:** component state is regularly replicated to the standby (i.e., state is copied using *passive replication*). In case of a failure, the passive component replaces the failed one, but it loses all state changes that occurred between the last replication and failure time;
    - **Active/hot-passive:** component state is replicated on any change (i.e., state is copied using *active replication*);
- **Active/Active:** more than one redundant system component is active in this scheme, which allows a better resource usage since there are no idle components. Requests may be accepted and executed by any member of a replicated component group. Two sub-forms exist:
    - **Asymmetric active-active:** when the software redundancy it provides is not supported by component state replication;
    - **Symmetric active-active:** when component state is replicated within a system component group (typically using advanced commit protocols);

Redundancy is indeed effective when we aim to provide highly available systems; it can also be helpful in failure detection and recovery. However, as mentioned, redundancy replicates design faults (introduced by human error or defective design tools). Indeed, systems will not provide extra fault tolerance by being redundant, since a failure resulting from the activation of a design fault will probably be equally experienced by the redundant systems, when systems experience the same fault activation conditions. Design diversity is an approach where the components used for computation are not exact copies, but are independently designed to fulfill the same requirements. Different designers and tools are used in each system, and shared features are avoided (A. Avizienis and Kelly 1984).

Originally applied to hardware, design diversity has evolved to N-version software. An N-version software unit is a fault tolerant unit that relies on a generic decision algorithm that determines a consensus from the results delivered by two or more member versions of the N-version software unit (A. Avizienis 1995). The process by which the versions are produced is called N-version programming. N-version programming can be defined as the independent generation (by N individuals or groups that do not interact with respect to the programming process) of N functionally equivalent programs from the initial specification. The N programs allow concurrent execution and explicitly include particular verification points (A. Avizienis 1995).

Besides software diversity, data diversity can also be used as a fault tolerance strategy. Software faults often lead to failures only under special conditions, and for some applications, there might be a way of expressing their input and internal state in several different but logically equivalent ways. This suggests obtaining a related set of points in the data space, executing the same software on these points, and then employing a decision algorithm to determine the system's output. These kinds of techniques make use of data diversity to tolerate residual faults (Ammann and Knight 1988).

In (Looker, Munro, and Jie Xu 2005) an N-version based mechanism (WS-FTM) is used along with a voting strategy to increase reliability in web services. WS-FTM achieves its transparent usage of replicated web services by use of a modified stub (a stub generator tool was created for this purpose). By using a voting mechanism, some individual failures can be eliminated so that the integrity of the final result is not compromised.

The voter (a simple majority voter) can assure to an agreed level of integrity that a correct result or an error is returned.

WS-FTM is built upon a layered architecture that extends Apache Axis 1.1 (Apache Software Foundation 2008a) standard classes. It currently assures no compliance with the WS-I Basic Profile standard (Ferris et al. 2007) that promotes interoperability for web services development. It also couples the developer to the Axis SOAP stack and associated tools, which may be undesirable in some cases and brings complexity to the development model. No support for diversity at the interface level of the component services is supplied. Note that this type of support is extremely useful when selecting alternatives for a service using third party components, as they usually provide their functionality through different interfaces.

In (Salatge and Fabre 2007) a mechanism is proposed that enables the creation of connectors that provide fault tolerance to unreliable web services. These connectors can be designed by clients, providers or other users (e.g., dependability experts), starting from the WSDL service description. These are created in a specific language (DeWel – Dependable Web service Language) that was designed taking in consideration particular fault tolerance characteristics. It uses compile-time and runtime verification of a set of language constructs that serve the purpose of declaring recovery strategies and writing runtime assertions. The core of the presented mechanism consists of:

- **Runtime assertions** (specified by the user) that check input and output messages. Error detection and signaling mechanisms are also used (SOAP exceptions returned to clients);
- **Built-in recovery actions** based on replication models that support identical web services. Equivalent services are also supported by the notion of abstract web services. Obviously, the selection of a particular replication mode and recovery action depends on the services state handling issues and features. The following models are supported:
    - **Passive replication** (corresponds to the previously exposed active-passive model) provides three models: a basic scheme; a stateful replication strategy, which can be used when the original web services provide specific state handling operations (*SaveState* and *RestoreState* operations); and log-based replication, where requests are saved to logs

and replayed upon failure. For this latter case, the connectors provide *StartSession* and *EndSession* primitives;

- **Active replication** (corresponds to the active-active model). It features a basic scheme that delivers the first message obtained from the replicas to the client, but also offers the possibility of using voting algorithms over the obtained results;

- **Monitoring and error diagnosis** of the target web services. Errors are collected in target access points and lead to extended error reports.

The approach appears promising; however the actual development model is far from being simple. In fact, developers have to learn a new language – DeWel (DEpendable Web service Language) and learn how to use the associated compiler and tools.

In (Gorbenko, Kharchenko, and Romanovsky 2007) the authors discuss vertical and horizontal composition in Service Oriented Architectures (SOA). The former is used for functionality extension (it defines the workflow of systems) and the latter for dependability improvement by dealing with a set of redundant (possibly diverse) web services. Several web service modes are analyzed, namely:

- **Parallel execution for maximum reliability**: the middleware produces an adjudicated response;
- **Parallel execution for maximum responsiveness:** the first answer (that is not clearly incorrect) will be the one to return to the client;
- **Configured parallel execution:** the middleware waits for a configured number of responses;
- **Sequential execution for minimal service capacity:** services are executed sequentially. Subsequent services are only executed when a response is obviously incorrect;

All of these modes were simulated in a MATLAB environment, taking into consideration empirical assumptions. The authors briefly analyze current workflow language support for these configurations, and propose

developing new language constructs and patterns to be executed in different popular workflow engines.

Current studies that focus on improving the availability of services make limited use of redundancy and diversity to tolerate failures in specific components. In this thesis we propose a mechanism that was designed to overcome the major limitations found in current approaches. In practice, we focus on providing a mechanism that is compliant with current web services development models, is practical and does not bring excessive complexity to the developer and also provides support for response correctness verification.

2.4.4 Timing failures

In this section we present an overview of research studies that focus on the detection of timing failures in time-critical systems and on generic data-driven prediction techniques. In real-time software services, the correctness of the system depends not only of the logical result of the computation (i.e., the content of a response) but also on the time at which the results are produced (Stankovic 1988). Nowadays, web services are frequently used in time-critical business and enterprise applications (Amazon 2010); in these environments a correct output that is not produced before a deadline may be completely useless.

Real-time systems can have their time limit restrictions specified in two ways, which differ in terms of the amount of flexibility that is placed on the software system regarding the fulfillment of the timing requirements. **Hard real-time** deadlines are those whose correctness depends on the logical result of computation as well as the time in which those results are produced. Indeed, a failure occurs when a hard deadline is missed in a particular system. On the other hand, **soft real-time** deadlines also have a dependency on time constraints, however, when a system is unable to meet a given time limit, a failure does not necessarily occur (Pekilis and Seviora 1997). This applies to many kinds of application types such as videoconference for example, where some job loss is permissible. Here, some frame or packet delays (or even losses) are permissible and may even go unnoticed by the user (Sha et al. 2004). This kind of behavior may be desired or sufficient for web service clients, and must also be supported by providers.

Fundamental requirements of time-critical systems are supported by two important properties: dependability and predictability (Halang et al. 2000). Work on classic dependability issues (e.g., such as availability or reliability), has already been produced (Long, Carroll, and Park 1990; Looker, Munro, and Jie Xu 2005; May 2009; Salas et al. 2006) and is also explored in this thesis. However, additional aspects directly related to temporal properties, such as accurate temporal failure detection and prediction, have been scarcely explored in web services environments.

The problem of timing failure detection in database applications is discussed in (Vieira, Costa, and Madeira 2006). The authors propose a transaction programming approach to help developers in programming database applications with time constraints. Three classes of transactions are considered concerning temporal requirements: transactions with no temporal requirements (typical ACID transactions), transactions with strict temporal requirements, and transactions with relaxed temporal requirements. The approach proposed implements these classes of transactions by allowing concurrent detection of timing failures during transaction execution. Timing failure detection can be performed at the database clients' interface, in the database server, or in a distributed manner. A performance benchmark for real-time database applications is used to validate the approach and to show the advantage of timing failures detection. The results presented show that it is useful to consider a new transaction programming approach aimed at supporting timing specifications for the execution of transactions.

(Pekilis and Seviora 1997) propose that a separate unit, which observes the inputs and outputs of the target software, should do the detection of temporal failures in real-time software. The authors claim that automatic detection of such failures is influenced by state dependencies, which require the unit to track a target's state and the elapsed time. A black-box approach is proposed for detecting real-time failures and quality of service degradations of session-oriented real-time software. The approach combines the target software's formal behavioral specification with its response time specifications into a new model – the timepost-model. The temporal failure detection unit interprets this model, and uses it for target state tracking and for determining the start and stop times of timing measurements. The approach was evaluated by empirically testing a small private branch telephone exchange capable of supporting multiple telephones and simultaneous calls. Comparing to usual white-box

approaches it was found that very complex and specific timing intervals could be tracked and measured.

Several works have been proposed that focus prediction using data-driven approaches, which can also be used for predicting timing failures. Among them are support vector machines (SVM) that in this context have been used mainly for software reliability prediction. An SVM is a learning system that uses a hypothesis space of linear functions in a high dimensional feature space, which is trained with a learning algorithm from optimization theory that implements a learning bias derived from statistical learning theory (Bo and Xiang 2007). A relevant aspect is the presentation of approach issues that affect the prediction accuracy. These issues include: whether all historical failure data should be used and what type of failure data is more appropriate to use in terms of prediction accuracy. The prediction accuracy of software reliability prediction models based on SVM and artificial neural networks are also compared in this work.

Local prediction schemes only use the most recent information (and ignore information bearing on far away data). As a result, the accuracy of local prediction schemes may be limited. Considering this, it is proposed in (Shun-Feng Su, Chan-Ben Lin, and Yen-Tseng Hsu 2002), a novel prediction approach termed as the Markov–Fourier gray Model. The approach builds a gray model from a set of the most recent data and a Fourier series is used to fit the residuals produced by this gray model. Then, the Markov matrixes are employed to encode possible global information generated also by the residuals. The authors conclude that the global information encoded in the Markov matrices can provide useful information for predictions.

A framework that incorporates multiple prediction algorithms to enable navigation of autonomous vehicles in real-life on-road traffic situations is presented in (Kootbally, Madhavan, and Schlenoff 2006). At the lower levels, the authors use estimation theoretic short-term predictions over sensor data to predict the future location of moving objects with an associated confidence measure. At the higher levels, the authors use a long-term situation-based probabilistic prediction using spatiotemporal relations and situation recognition. Interesting conclusions include the identification of the different time periods where the two algorithms provide better estimates and thus demonstrate the ability to use the results of the short-term prediction algorithm to strengthen/weaken the estimates of the long-term prediction algorithm at different time periods.

The importance of the time dimension in real-time systems is emphasized in (Halang et al. 2000). In fact, this relation is expressed by two fundamental user requirements: timeliness and simultaneity, which must be fulfilled even in extreme load conditions. The authors also advocate that predictability and dependability supplement the former two requirements, and highlight the importance of several qualitative performance criteria (over quantitative criteria) from which we would like to emphasize the following: timeliness – the ability to meet all deadlines; functional correctness; permanent readiness; dependability; behavioral predictability; simplicity; reliability; robustness; fault tolerance; graceful degradation upon malfunctions; and portability.

Current work shows that it is useful to provide support for timing specifications and make this support available for service clients. In fact, although studies exist in related domains (e.g., databases), web services still do not have practical mechanisms that can be used to detect (and predict) timing failures. This thesis proposes a mechanism that is inspired in current works, for instance by providing support for the acquisition of short-term and long-term data that can be used for predicting timing failures in a more effective way.

2.5 Conclusion

This chapter presented and discussed the state of the art on assessing and improving robustness properties of software services. Additionally, it presented an overview of fault tolerance techniques, with respect to content and timing failures and focusing in the web services context.

Software services, such as those typically used in SOAs, are systems that incorporate a large set of technologies and evolve very quickly. Testing such systems is of extreme importance to businesses and, in particular, providing robust services to clients or business partners is a crucial step. Several approaches sharing similarities with robustness testing have been proposed in the web services context. However, in general, they are too specific to be generically applied to different types of services, or do not provide, with enough detail, a classification approach for observed problems. Issues like the approach practicality or the absence of a concrete tool have also often been observed. On the other hand, messaging middleware has frequently been studied for performance, while robustness has been largely disregarded. As an important means to

achieve loosely coupled communication between software parties, and regarding that robustness issues can also represent severe security issues, it is important to provide developers with a practical technique to test such systems. Ideally, an approach for robustness testing of services should be generic enough (but also descriptive enough) to easily be applied to any kind of software service.

After creating a robustness assessment methodology, the next obvious step is to improve the robustness of services. There are numerous studies of robustness and security improvement of software applications using wrappers; however, a simple automatic approach focused on the web services technology and that takes advantage of its description language (WSDL) and associated technologies (e.g., XML Schema), is yet to be defined. Another aspect associated with robustness is the security of web services. Web applications are still being deployed with security vulnerabilities, allowing the successful execution of security attacks (SQL and XPath injection attacks are relevant). A high performance and low intrusiveness approach is extremely important, as opposed to approaches that require, for instance, source code access, or are limited with respect to specific code structures being used. Finally, it is important to emphasize that a simple tool for robustness assessment can also be used for improvement, if its use is well integrated in the software development process. Despite this, recent software development techniques are unable to integrate robustness testing well.

In business critical systems, and in addition to robustness and security, properties like availability, correctness, and timeliness are of vital importance. Redundancy and diversity are well-known techniques to improve availability and deal with design faults. Despite this, existent tools are not practical and simple enough for developers to use. While some lack diversity support, or are featured limited, others require complex configurations in order to be used. Simplicity is a definitely requirement when the goal is to build a tool that can be used by developers in already complex business critical environments. Furthermore, in business environments time is critical. Several studies aim to detect and predict timing failures in real time systems, yet the web services technology is still lacking capabilities that endow services with timing detection (or prediction) features.

This thesis focuses on the above issues, adding contributions in each of these key topics.

Chapter 3
Approach for Testing the Robustness of Software Services

In this chapter we propose an approach for robustness testing of software services. The approach is described in abstract terms so that it can be applied to various classes of services. Concrete instantiations of the approach to web services and messaging middleware are presented in Chapter 4.

Our proposal for robustness testing of services has its foundations on previous work (P. Koopman and DeVale 1999; Rodríguez et al. 1999; Mukherjee and Siewiorek 1997) and uses erroneous and malicious call parameters to test a given system. The robustness tests consist of combinations of acceptable, exceptional, and malicious input values that are generated by applying a set of predefined rules, according to the data type and domain of each parameter.

The approach defines the several **components** and the **procedure**. In these terms, there is some similarity to the dependability benchmarking field (as discussed in Section 2.3.2). In fact, classical robustness testing methodologies are referred as robustness benchmarking by several

authors (P. Koopman and DeVale 1999; Rodríguez et al. 1999; Mukherjee and Siewiorek 1997). In practice, we propose the following major components for robustness benchmarking of services:

- **Workload:** represents the work that the service must perform during the benchmark execution (e.g., arithmetic calculations, database accesses, etc.). Multiple types of workloads are available (e.g., real workloads, generated, etc.), however in practical terms, not all types may be accessible or easy to apply in a particular scenario, as described ahead.
- **Robustness tests:** a faultload that consists of a set of invalid and malicious (but possibly valid from a domain view-point) call parameters to expose robustness problems in the target service. Invalid and malicious parameters respectively extend previous work on robustness and penetration testing (P. Koopman and DeVale 1999; Stuttard and Pinto 2007), and are typically applied using fault injection techniques (Hsueh, Tsai, and Iyer 1997).
- **Service classification:** a three dimensional characterization of the behavior of the service under testing, including: 1) a classic severity scale that characterizes the behavior of the service while executing the workload in presence of the robustness tests (i.e., a failure mode scale) (P. Koopman and DeVale 1999); 2) a detailed service characterization using behavior tags (a set of keywords that describe a particular service behavior); and 3) a compliance scale that classifies the level of compliance of the service to its specification. Not all perspectives can be used in all scenarios (e.g., if the service specification is unknown to the tester, the compliance scale cannot be used), still these different views are extremely helpful when describing the service in detail, enabling a comparison with other services.

In addition to these components, a benchmark must also specify the **testing procedure** (which includes the required setup). This procedure is a description of the steps and rules that have to be followed during the benchmark/tests implementation and execution, and integrates the previously described components. For testing the robustness of services, we propose the following set of steps:

1. **Analysis of the interfaces under testing:** analysis of the service under test in order to gather relevant information to execute the tests (e.g., existing operations, parameters, data types).
2. **Workload generation and execution**: generation of a workload to exercise the services under testing. Workload execution may be useful to assess code coverage.
3. **Robustness tests definition and execution**: execution of the robustness tests in order to trigger faulty behaviors that may disclose robustness problems.
4. **Service robustness characterization**: failure modes and service behavior identification based on the data collected in steps 2 and 3.

This chapter is organized as follows. Section 3.1 describes the interface analysis step and Section 3.2 discusses the types of workloads that can be used. Section 3.3 presents the robustness tests generation and execution step, including a discussion on aspects like the selection of the fault injection strategy, the fault model, the overall procedure, among others. Section 3.4 addresses the characterization of the behavior of services in terms of robustness and Section 3.5 concludes the chapter.

3.1 Analysis of the interfaces under testing

Before generating and executing the workload and the robustness tests we need to obtain some definitions about the service operations, parameters, data types, and input domains. The information about the parameter domains of a particular service operation is crucial for the two next steps of the robustness tests (workload generation and tests definition). In fact, an adequate definition of the input domain may allow achieving higher source code coverage[1]. Obviously, covering more code allows to reach more critical code sections, enabling the disclosure of additional bugs. When the domains of operations are not available, the benchmark user should provide them. However, even if no extra domain information is available (e.g., the user is testing an unknown service), a completely random workload may also be used, as we will see ahead.

[1] We use the terms code coverage throughout this thesis to refer to statement coverage (Myers 2004).

Providing information about operations and parameter data types and domains obviously depends on the type of service being tested. For instance, when considering typical service technologies like web services or messaging middleware it is possible to obtain information concerning operations and parameter data types easily. In the case of web services such information is available in the WSDL document, while in the case of messaging middleware, the API (Application Programming Interface) contains this kind of information and is available for any developer. However, it is rare to have the parameter domains available. In the case of web services, although they can be referenced in the WSDL document, practice shows that developers frequently disregard this definition. Similarly, if we consider typical messaging solutions, such as JMS, the operation domains of running applications are virtually unknown and the domains of the messaging middleware itself can only be found in the technology specification document (Sun Microsystems, Inc. 2002), which is written in natural language and may change over time. This makes the creation of an automatic domain-extraction mechanism difficult, particularly in the case of messaging-based services. Despite these difficulties, the information provided by the service, complemented with information provided by the user, can be effectively used in the next two steps of the proposed robustness testing approach: workload generation and tests definition.

3.2 Workload generation and execution

The workload defines the work that has to be done by the service during the benchmark execution. Three different types of workload can be considered for robustness benchmarking purposes: real workloads, realistic workloads, and synthetic workloads (Jim Gray 1992).

Real workloads are composed of applications used in real environments. Results of benchmarks using real workloads are normally quite representative. However, several applications are needed to achieve good representativeness and those applications frequently require some adaptation. Additionally, the workload portability is dependent on the portability requirements of all the applications used in the workload.

Realistic workloads are artificial workloads that are based on a subset of representative operations performed by real systems. Although artificial, realistic workloads still reflect real situations and can be quite

representative and even more portable than real workloads. Performance benchmarks frequently use realistic workloads as a key component to exercise the target system (Transaction Processing Performance Council 2008; Transaction Processing Performance Council 2011; Standard Performance Evaluation Corporation 2007a; Standard Performance Evaluation Corporation; Standard Performance Evaluation Corporation 2007b).

In a services environment, a **synthetic workload** can be a set of ***randomly*** selected service calls or a set of ***user defined*** calls. Synthetic workloads are easier to use and typically provide better repeatability and portability when comparing to realistic or real workloads. Despite the fact that the representativeness of a random workload is doubtful, it can be seen as a good alternative, as a real or realistic workload may be difficult to obtain or apply to a testing environment.

An important component in testing environments is, in fact, the test data (i.e., the workload). Distinct approaches have been studied with the goal of creating tools that automatically generate test data for a specific application (see (Edvardsson 1999) for a survey on automatic test data generation techniques). An example of a possible approach is the generation of the workload based in the automated analysis of the service source code and on some simple definitions provided by the benchmark user (obviously, as the source code is needed, this alternative can only be used by service providers). In (Santiago et al. 2006) state charts are used for automated test case generation. Another approach is to generate the workload using the characterization of real load patterns through the application of Markov Chains (Barros et al. 2007).

In some cases, before proceeding to the next step of the approach (defining and executing the robustness tests), the user may be interested in executing the workload with the goal of understanding the service behavior (useful, for instance, in the absence of a service specification). This workload execution may be also useful to obtain basic information about its properties, such as the amount of code that the workload can cover. Moreover, if the workload is being generated (i.e., it is not a pre-defined or a user-provided workload), information about its coverage can be used as feedback to the workload generation process. This way, based on the coverage results and whenever possible, a workload generation procedure can be tuned to produce a higher quality workload, i.e., with higher coverage.

3.3 Robustness tests definition and execution

Robustness testing involves parameter tampering at some interface level, i.e., applying a mutation to a parameter involved in a given operation. This section describes several aspects related with the definition and execution of the robustness tests. It starts with a description of the typical domain relations that exist in a service application. Several fault injection strategies for robustness testing of software services are then discussed, followed by the procedure for executing the robustness tests. The section ends with a discussion of several aspects that should be tuned according to specific testing goals.

3.3.1 Services and domains

In order to have an effective strategy to mutate parameters it is necessary to understand the typical structure of a service application (concerning architecture and the use of domain space). Figure 3.1 presents an example of this common structure and the typical relation with the domain space of a generic operation input parameter.

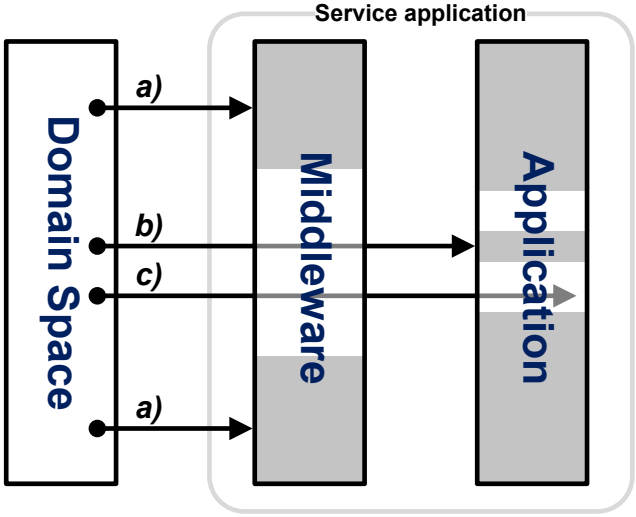


Figure 3.1 – Generic relation between a parameter domain and typical application structure.

The three rectangles in the figure represent, respectively and from the left hand side to the right hand side: domain space outside the service application, domain space at the middleware, and domain space at the application. White rectangular space is valid domain, whereas grey is invalid (i.e., not usable at least at the entrance of that particular layer). As we can see in the example presented in Figure 3.1, the available domain space is typically incrementally reduced as it passes through the software layers involved in a services environment. At the beginning, all domain space is available to be used. However, the middleware layer typically introduces some limitations to this space and the same applies to the application layer, which adds further constraints.

Consider, for instance, that Figure 3.1 represents a server-side view of a web service that provides an operation that accepts an integer parameter as input. The domain space on the left hand side of the figure represents, in this case, the space that holds all numbers. However, not all numbers will be accepted as part of the operation invocation at the middleware layer. In the case of an integer, the server-side middleware will typically only accept limited values (e.g., between -2147483648 and 2147483647). In this context, a significant amount of values belonging to the complete numeric domain space will be rejected by a correctly implemented middleware, as this layer, by request of the underlying application, is now only accepting valid integer values for that particular operation (this is represented by '**a)**' in Figure 3.1).

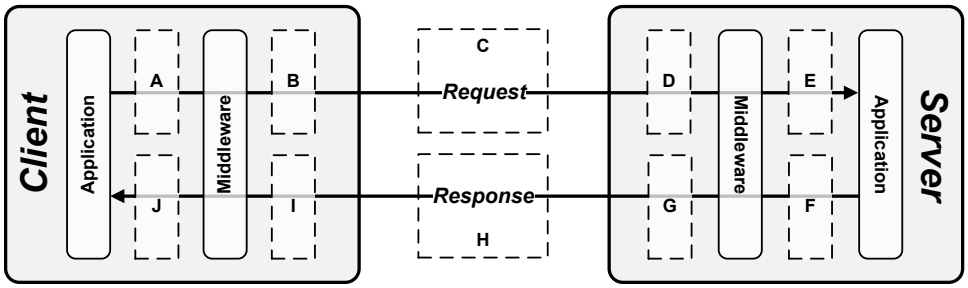
Adding to the restrictions imposed by the middleware being used, it is very frequent that a given application uses a much more limited domain than the one provided by the underlying middleware. Such kind of relation is observable in '**b)**', where a value that has already passed the middleware layer is not accepted at the application level (consider, as an example, a size limitation on a user password or person age). Despite this, there should be a valid interval where we can observe a valid value flow starting at the complete domain space and ending at the application layer '**c)**'. It is important to notice that, some values extracted from the left hand side of the figure may go through a conversion process at the middleware level (depending on the specific middleware being used) and, in this sense, the value mapping may not be direct, but normally the domain limitations apply in a similar manner.

An important aspect subjacent to the illustration presented in Figure 3.1 is that it clearly maps to a typical application entry point – the user input via a client application. However, it can also be read in the reverse way, in a

service-to-service call context. That is, a service behaving as a client of another service (for instance, two interacting services in a service composition). This issue makes the interfaces' analysis more difficult, as there are multiple application entry points, which is obviously a major issue in robustness testing.

3.3.2 Fault injection strategies

When considering a typical services environment (i.e., client-server scenario), the exact location for parameter tampering may depend on several factors, such as the testing conditions (e.g., no access to the service provider), the type of fault to inject, the technologies involved, the part of the system to be tested, etc. For instance, Figure 3.2 represents a typical client-server model, including a full end-to-end interaction. Each host (client or server) runs an application that uses a middleware component (e.g., a web services stack). Depicted in the figure there are several fault injection locations represented by the letters 'A' to 'J'. The table in the lower part of the figure indicates the possible fault injection locations adequate for testing each different target subsystem (the client application itself, the server middleware, etc.).



Fault injection location	Fault injection target			
	Client		Server	
	Application	*Middleware*	*Middleware*	*Application*
Client	I, J	A, I	B	A, B
Server	F, G	G	D, F	D, E
Network	H	H	C	C

Figure 3.2 – Fault injection locations

In the conceptual scenario represented above, there are three main fault injection locations: the client (represented as 'A', 'B', 'J', and 'I' in the figure), the server ('D', 'E', 'G', and 'F'), and the network ('C' and 'H'). In each host there are two more available locations: in the interaction flow between the main application and the middleware in use ('A' and 'J' at the client; or 'E' and 'F' at the server), or the interaction flow between the middleware and the network ('B' and 'I' at the client; 'D' and 'G' at the server).

In this context, multiple software fault injection techniques can be used. For instance, network-level fault injection (De, Neogi, and Chiueh 2003) can be applied in locations 'C' and 'H' and, in fact, in some cases it may be adequate to install an interceptor proxy to capture requests/responses and modify the captured object according the fault being injected. On the other hand, when the injection location is at the host (locations 'A', 'B', 'J', 'I', and 'D', 'E', 'G', 'F'), it may be difficult to introduce a new component between the target application and the middleware or between the middleware and the outgoing/incoming information. Despite this, in many cases, a code insertion solution may be an adequate option (Hsueh, Tsai, and Iyer 1997). As an example, code or bytecode instrumentation (Kiczales et al. 1997) provides a low intrusiveness solution that can be easily used to modify the target application or middleware, allowing fault injection to occur before specific instructions.

Let us suppose that the fault injection technique is not a limitation and that, for instance, the goal is to test client-side middleware while having only full access to the client (e.g., due to security reasons there is no extra access to the server or network). In this case, the only available injection locations are 'A' and 'I'. Obviously, choosing one of these locations depends on the type of middleware or on the goal of the test itself. In several technologies (e.g., web services) the middleware's main function is frequently to transform an application message into some standard message to be transmitted via network (e.g., transforming a programming language level operation call into a SOAP request) and backwards (e.g., receiving a SOAP response from the network and deserializing it to programming language level object). In such case, the injection location would be 'A' if the user wants to test the client-side serialization process or 'J' if the goal is to test the client-side deserialization.

There are, in fact, many combinations that can be considered when creating a test environment in a services context. The interaction presented in Figure 3.2 represents the use of a single technology on both

ends. However it is possible to extend this scenario to, for instance: distinct middleware components being used at each side, the server behaving as a client of another server and using an extra middleware layer (developed by a different provider), etc. All of these conditions add complexity to the final service being delivered and increase the number of testing possibilities. Whenever there is no possibility to test the whole system, the user has to make choices and should take into account the critical points of the system.

In the typical client-server topology frequently observed in service-based applications, the server-side middleware is a critical component. In fact, in these environments, rich resources are frequently maintained at the server side (e.g., persistent storage with client information), and there is frequently only one service endpoint (although the service can be maintained internally by multiple computers). A problem in a client may impair, for instance, a business transaction involving only that client; however, a problem at the server-side may affect thousands of clients that are trying to legitimately use the service. Note also that the server-side middleware may be built differently (in many cases it may not use the same version or even be provided by the same party as the one at the client) or perform different functions than those executed at client-side. In this sense, testing the client-side middleware is obviously not sufficient to guarantee an adequate server-side behavior.

The concrete case studies presented in chapters 4 and 5 represent tests over web services implementations and server-side middleware. As referred, additional research studies could be carried on other instantiations, such as tests for client-side middleware or even client implementations. Such studies may be executed in a large number of conditions (fault-injection method and location, technologies used, techniques applied, etc.), however we believe that the actual conditions used in our work are already quite representative, and more importantly, serve the main purpose of illustrating different concrete instantiations of the robustness testing approach.

3.3.3 Fault model

In order to perform fault injection, a fault set must be defined (taking into account domains, as discussed in Section 3.3.1). In robustness testing of services, this is a list of unique faults that focus limit conditions (e.g., limit

domain values) that typically represent difficult validation aspects, or common development assumptions (which are frequently the source of robustness problems). Table 3.I presents the generic set of faults proposed for robustness testing of services, grouped by fault category, including a description for each type of fault along with some illustrative examples. As previously mentioned, this fault model was defined based on previous work on robustness testing (P. Koopman and DeVale 1999), and extended to include malicious values, typically used in command injection attacks (Stuttard and Pinto 2007).

When defining a robustness testing approach for a concrete type of service (e.g., web services implementations or JMS middleware) a set of specific mutations should be generated (as defined in Table 3.I) for each available data type, including any input resulting from the invocation of an external service (specific instantiations of the fault model defined in Table 3.I and their application in real services systems can be found in Chapter 4). For illustrative purposes, Table 3.II presents a possible instantiation of the rules applied to an integer parameter in an imaginary SOAP web service operation that accepts an integer between 10 and 20 (inclusively) and is initialized with the value '15'.

As we can see, the effect of some of the rules depends on the generated parameter (adding or subtracting a value), while other rules can only be used with particular injection methods (e.g., setting an empty integer on an object cannot usually be achieved with a high-level code insertion technique, as the programming language may enforce the definition of a valid integer, at compile time). The application of some of these rules may also not reach the target implementation. For example, in web services, setting an empty number, adding one to the maximum value type, or subtracting one from the minimum value type, are conditions that will be typically caught at the middleware, and thus generally do not reproduce a problem that resides at the service implementation level. At most, these faults may disclose a problem in the middleware layer; however, due to the large diversity of systems using services, it is not very unlikely that conversion issues exist. In these cases, these types of faults and their effect on the middleware layer are an indicator that further middleware tests are required.

Table 3.I – Description of the robustness tests fault model.

Fault Category	Description	Examples
Omission	Null and empty values	• Null integer, empty string
Size	Maximum and minimum valid values in parameter's the domain	• Set the maximum value valid for the parameter or the minimum value valid for the parameter
	Values outside the maximum and minimum valid values of the parameter's domain	• Maximum value valid for a number plus one • Minimum value valid for a number minus one • Add characters to overflow a string's maximum size • Add elements to a list
Type	Maximum and minimum values of the data type	• Setting 127 or -128 to a Java byte type
	Values that cause data type overflow or underflow	• The maximum number valid for the type plus one • The maximum number valid for the type minus one
	Exception objects	• Replacing the response of a external service invocation with a Runtime Exception • Replacing the response of an external service with a Exception hierarchically related to an Exception declared by the service
Format	Values with special characteristics or format	• Nonprintable characters in strings • Strings non-compliant with the required format • Setting 0,1,or -1 • Limit dates in different formats: Last day of the millennium, first day of the millennium
	Invalid values with special characteristics	• Invalid dates using different formats: – 2/29/1984 – 29-2-1984
Security	Special characters or expressions that are typical sources of robustness issues and also represent potential security issues	• Typically malicious expressions in data access statements as, for instance, in the SQL language: – " or 1=1 – ' or username like '% • The ' or " characters
Specific	Other technology / implementation specific values	• Important middleware characters or expressions ('<' or '>' in SOAP web services) • Verifying middleware data conversion strategies • Setting value zero for 'xsd:gYear' when testing SOAP middleware

Table 3.II – An example of application of the fault model to a web service operation.

Data type	Fault designation	Parameter mutation
Integer	NumNull	Non-existent XML SOAP tag
	NumEmpty	Empty XML SOAP tag
	NumAbsoluteMinusOne	-1
	NumAbsoluteOne	1
	NumAbsoluteZero	0
	NumAddOne	16
	NumSubtractOne	14
	NumMax	2147483647
	NumMin	-2147483648
	NumMaxPlusOne	2147483648
	MumMinMinusOne	-2147483649
	NumMaxRange	20
	NumMinRange	10
	NumMaxRangePlusOne	21
	NumMinRangeMinusOne	9

The faultload has obviously to follow the evolution of systems and technologies. In the particular case of services (typically systems that change fast and evolve), the fault set should be continuously extended to accommodate more typical causes of robustness problems and to address new techniques able to explore a wider range or new types of robustness problems.

3.3.4 Tests execution

Using the components described in the previous sections (service interface information, workload, and fault model), it is possible to define a generic profile for the execution of robustness tests. The user of the benchmark may focus on testing only a small part of a system, but the tests can also target the whole system (in terms of testable operations). To improve the tests coverage and representativeness, when the goal is to fully assess a system, the robustness tests must be performed in such way that fulfills the following set of key goals/rules:

- All the operations provided by the web service must be tested;
- For each operation, all parameters must be tested;

- For each operation, any known external invocations must be tested;
- For each parameter, all the applicable tests must be considered. For instance, when considering a numeric parameter, all tests described in Table 3.I that can be applied to numbers must be performed.

Taking these rules in consideration, our proposal for robustness testing includes three major phases (see also Figure 3.3):

- **Phase A (pre-injection period):** valid service parameters, generated in the workload generation step, are used in one or more executions of a given service operation;
- **Phase B (injection period):** faults (invalid and malicious values) are injected in the parameters being tested, which may belong to the public operation interface, but can also be return values from external services invocations. The operation is executed in the presence of these faulty parameters;
- **Phase C (post-injection period):** regular parameters, generated in the workload generation step, are used in the operation with the goal of exposing errors not revealed in Phase B (i.e., caused by the faults injected in Phase B, but that did not manifest immediately, remaining latent).

Phase A is executed once before testing an operation (a service may provide multiple operations). For each parameter, phases B and C are repeated several times. In fact, these two phases are executed at least N times (N is equal to the number of different faults that can be applied to that parameter[2]). Note however that, this number may increase according to the user's preferences or the system's requirements (for instance, if the user wants to consecutively inject the same fault twice and verify the result). The global testing profile is illustrated in Figure 3.3, which is followed by a detailed description of the three major test phases and key insights on advanced configuration aspects of the tests (in Section 3.3.5).

[2] The term *parameter* is used, in this section, in a broad sense and refers not only to public interface parameters but also to any return values resulting from external service invocations.

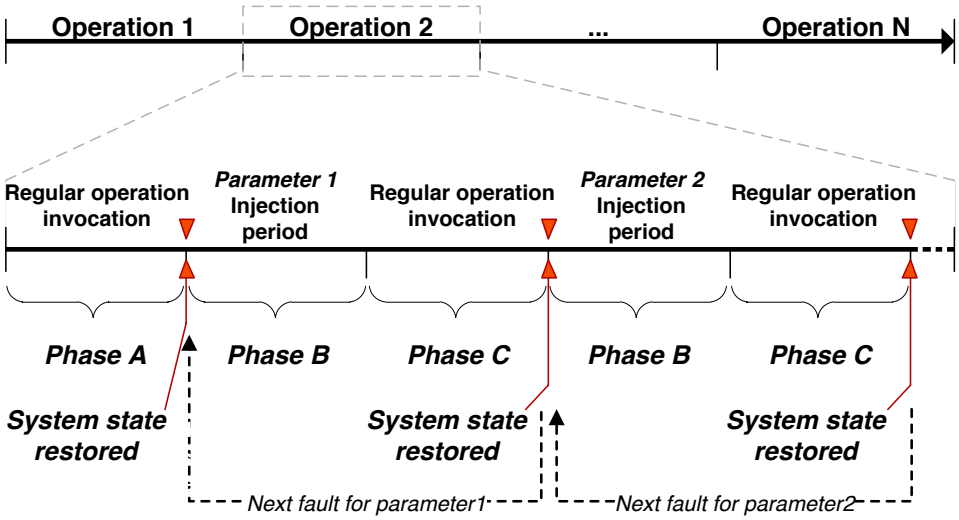


Figure 3.3 – Robustness testing profile.

Phase A consists of invoking the target operation with a set of parameters defined by the workload, and verifying the service's response. This phase is executed once per available operation and the goal is to understand the behavior of the target service without (artificial) faults. The execution of the workload (without considering invalid call parameters) includes several tests where each test corresponds to several executions of a given operation of the service. The number of times the operation is executed during each test depends on the size of the generated workload (that should be specified by the benchmark user). In practice, this represents the gold run that allows checking if subsequent robustness tests affect regular operation invocations. After this phase, the benchmark user may want to restore the system's state so that before proceeding the next phase, the service being tested is exactly in the same state as it was before executing Phase A. Obviously, this step can only be if the user has control over the service's infrastructure.

Phase B represents the execution of the robustness tests and consists of running the workload in presence of the invalid and malicious call parameters. For this purpose, a parameter is chosen from the list of the available and not yet tested operation parameters. As referred, according to the type of the selected parameter, a set of rules is applicable. For

instance, if the parameter is numeric, then all the mutations that are applicable to numbers (see Table 3.I) should be considered. However, only one of the rules is selected in each injection period and all invocations executed during this period are tampered according to that particular rule. The service response is then verified for correctness and consistency.

In the third phase – **Phase C**, the system runs in order to check for abnormal behavior unnoticed in Phase B. The existence of Phase C is fundamental as a fault may be silently triggered during Phase B but only be exposed in Phase C. Additionally, even if a failure is clearly exposed during the injection period, it may or may not persist to Phase C (or to subsequent test phases). Longer failure persistence may be an indication of presence of a more severe issue.

After phase C, the system may be restored (in some cases, when the tester does not have control over the server infrastructure, it is not possible to restore the system), and testing continues. At this point, if there are faults that have not yet been injected into the parameter being tested, another injection period is executed (for this same parameter) and the next available fault is applied; after this new injection period, phase C is executed for service verification. Phases B and C are, in fact, repeated until all faults have been applied to a particular parameter (again, this includes return parameters of external services). When there are no more faults to apply, the next parameter is tested and when all parameters have been tested, the testing process moves on to the next operation and proceeds as before (starting with phase A).

3.3.5 Advanced configuration of the tests

Considering the procedure described above, there are several aspects that can be configured or tuned according to the tester's goals or to the specificities of the system being tested. Among others, we highlight the following (detailed in the subsequent paragraphs):

- Duration of phases A, C, and B (pre, post, and injection periods, respectively)
- Replicating faults in phase B (injection period)
- Restoring the system state

- Replaying the tests for failure analysis

The **duration of the three phases A, B, and C** can be obviously adjusted to the goals of the experiment, ranging from one single operation invocation to a larger number of invocations. Many factors contribute for this decision, including: the available testing time, the knowledge of the system being tested, etc. In practice, the number of invocations/tests that are generated may vary according to the knowledge that the tester has about the service. When in presence of a service whose execution coverage largely depends on the input parameters, it may be helpful to generate more values in order to have a higher coverage for the complete set of tests (and possibly disclose more robustness issues). Typically, service providers are in a better position to decide the extension of the tests; however, this should be a configuration parameter that may also be set without requiring extra knowledge about the service.

In an ideal scenario, each of the testing phases would last the time needed to cover 100% of the code being tested in that particular operation. However, in practical terms this may not be feasible due to time constraints, to the complexity of the service being tested, or even because the tester may not have information about the amount of code being covered by the tests (e.g., if the tester is not the service provider). As we will see ahead, in many cases a single invocation is enough to detect major severe problems. Obviously, more extensive testing should be allowed if required by the tester's goals.

When testing some services, particularly asynchronous systems, the user may need to execute more than one call in sequence, as this may have influence on the behavior of the target service. For instance, when testing services that use unique identifiers for each service call, it is of interest to test how the system handles duplicate identifiers (i.e., distinct invocations of a given operation using a constant parameter); for these cases, two sequential invocations, that use the same fault (e.g., category *Specific* in Table 3.I), can be used. The target system may then be prepared to handle this **duplicated fault** adequately or not. Based on the characteristics of the system being tested and on the test goals, the tester should be allowed to decide the number of invocations/messages sent to the target system in Phases B and C (i.e., the user should set the number of times each distinct fault is applied).

After any execution of Phase C, the **system state may be restored** so that tests can continue with no accumulated fault effects. However, it is not always possible to execute this restoration step, which is usually applicable only when the benchmark is executed by the service provider, as in many cases consumers are not able to restore the service state (since it is being tested remotely). In fact, a potential problem related to the use of robustness benchmarking by consumers is that they cannot control the state of the services under test and, in some cases, are not even allowed to test those services. In addition, some tests may change the state of the service (e.g., change the data used), which is also a problem (obviously not relevant for stateless services). These problems are minimized when the provider offers a parallel infrastructure for testing purposes (frequently required by consumers before starting to use a paid service). Note that, these problems do not affect the ability of providers to test stateful services, as they are able to test the code in isolated environments, which is a typical testing approach for a software artifact.

When state restoration is possible, the user should be able to select an appropriate strategy to increase the effectiveness/effort ratio, as this process may require some effort or time. Choosing a high granularity for system restoration (restoring the system less times – for instance, only after completing the tests for an operation) allows decreasing the tests overall duration. On the other hand, choosing a finer granularity for restoring the system state (e.g., after completion of each phase C) increases the tests duration, but may provide information that is easier to analyze when the goal is to understand the causes of failures.

A sensible approach is to apply system restoration after finishing all tests for a given parameter (and before starting the tests for the next available parameter). The main disadvantage of this approach is that, as indicated, when a failure is detected, and depending on the system, it may not be possible to understand if a particular injected fault was responsible for exposing the problem, or if it was due to a sequence of faults. In some cases it is easy for a developer to understand the cause of a failure; however, when more information about the failure cause is needed, ***reverse-replaying*** the tests can be used.

Reverse-replaying the tests basically results in executing test N first (N represents the number of an executed test that exposed a failure); then test N-1 and test N; then test N-2, test N-1, and test N, and so on. If possible, system restoration should be applied always after executing test N. This enables the user to understand which fault caused the failure, or which

fault sequence was responsible for it. In fact, starting from the same system state and using this technique, if the same failure consistently occurs, for instance, after applying test N-1 and test N (but not when performing only test N), it is possible to conclude that there is a robustness issue associated with the consecutive application of these two tests. Obviously some failures are difficult to re-expose (e.g., transient failures), or even impossible due to the software's specific structure. Despite this, empiric evidence indicates that the repeatability of the results of robustness tests is quite high (Kropp, Koopman, and Siewiorek 1998).

3.4 Service robustness characterization

The final step of the process is the characterization of the observed failures. Depending on the service being tested and on the testing conditions, the tests can be used to describe the system (although, in different degrees) in terms of:

1) Severity of the observed failures;
2) Detailed service behavior;
3) Compliance to a service specification.

Keeping these 3 perspectives in mind, we propose that the robustness benchmark should use of a classification scheme to distinguish the **observed failures** in terms of severity in a global manner (i.e., a set of failure modes). A possible approach is to use a well-known classification, such as the CRASH scale (P. Koopman and DeVale 1999), as basis for services characterization and tailor this scale according to the specificities of the class of services targeted. The CRASH scale has been originally created to classify the robustness of operating systems; however, the following points show how the CRASH failure modes can be adapted to a generic services environment:

- **Catastrophic:** the service supplier becomes corrupted, or the server or operating system crashes or reboots;
- **Restart:** the service supplier becomes unresponsive and must be terminated by force;

- **Abort:** an abnormal termination is detected when executing the service. For instance, abnormal behavior occurs when an unexpected exception is thrown by a service;
- **Silent:** no error is indicated by the service on an operation that cannot be concluded or is concluded in an abnormal way;
- **Hindering:** the returned error code is incorrect.

This scale, based on solid work in robustness testing can, in fact, be applied to services environments, as we will see in Chapter 4. However, although it is wide enough to fit a large variety of potential problems, it is unlikely that a single scale can fit all classes of existing services (and also future new classes). Also, as we will see, in many cases not all failure modes are observable, depending on the tester point of view. For instance, without access to the service infrastructure it may not be possible to distinguish between a Catastrophic and a Restart failure.

Considering the previous issues, it is convenient to limit the failure modes to the ones that are in fact observable, or even transform the failure modes in order to fit the specificities of the class of services being tested. However, this may not be possible before having enough testing data (i.e., a large amount of information regarding the typical behavior of a particular class of services). An illustrative example of this is presented in Section 4.1.4 where we introduced a simpler failure mode scale with the goal of describing client-side testing results in a more adequate manner. In practice, although the failure mode scale is simpler, all failure modes are observable and distinguishable from a client point-of-view.

Despite the fact that, in general, only one scale is typically used in classic robustness testing studies, we believe that different perspectives are helpful, for instance, in dealing with the diversity of failures that can occur when testing software services. This way, our second proposal consists on the use of a **tag-based service behavior classification** scheme. This should allow describing the service behavior in a more detailed manner, complementing the previous severity scale. Also, when in presence of extensive tests, this behavior classification scheme can aggregate results and help the tester or developer to quickly understand the cause of failures. If, for instance, we consider the variety of ways a database can fail (e.g., bad SQL grammar, data integrity violations, etc.), a quick tag can eventually direct the developer to the software components that handle the database connections.

Table 3.III presents a set of behavior tag classes that was iteratively built, based on a large experimental evaluation (see Chapter 7). At this point, the main goal is to present a generic set of tag classes. In practice, these classes have to be set to specific instances according to the class of service being tested, the degree of classification detail desired, or on any other service specificity (refer to Section 4.1.4 for a particular instantiation of these classes). As an example, we refer the '*Arithmetic operations*' tag, where it is possible to imagine a given service failing after trying to perform a division by zero, or after adding two values whose result overflows the data type being used.

Table 3.III – Generic behavior tag classes

Tag class	Description	Example
Incorrect object handling	An object is improperly used, resulting in a robustness problem	A null reference occurs during the execution of an operation
Data access operations	A problem related with the use of a persistence engine or persistent data occurs (e.g., a read or write operation in a database)	An SQL Exception is thrown while executing an operation
Data type usage issues	An inadequate data type is used at the service interface	An operation requiring a date for execution displays a String at the service interface (it internally tries to handle the String as a date)
Information disclosure	Privileged server information is disclosed to the client	A username or password is disclosed
Arithmetic operations	The service is unable to execute an arithmetic operation	The operation tries to divide a value by zero
Invalid input format	The service requires an input parameter in a given format but is not prepared to deal with invalid formats	The service requires a globally unique identifier (GUID) and fails when receiving a value in a different format
Other	Other types of issues observed that do not fit in the above classes	–

As we can see, the ground for the service behavior characterization is a set of seven base tag classes. An important aspect is that a given service response may be analyzed and marked with multiple concrete tags and also with tags belonging to multiple classes. We do not propose this as a closed tag classification system, as it is adaptable and extensible, and its use is optional. Despite this, since it was built based on the analysis of a large set of services responses, it represents a starting point for the process of characterizing the behavior of services.

The third perspective of our approach is the level of compliance between a service implementation and its specification. A compliance or conformity problem is observed when an acceptable input (generated as defined in a specification) results in a robustness problem (e.g., an out-of-domain or unexpected service response). This way, we propose a **three-degree severity scale** that enables the comparison of the distinct services in terms of conformity to a specification, in the following terms:

- **Level 1 (severe):** the detected issue affects regular service operations (e.g., simple invocations) and is directly related with not following the specification.
- **Level 2 (medium):** the issue does not affect the regular behavior of the service (e.g., service does not become unavailable), however it is related to a restriction of the specification (e.g. restricting the domain value of a variable to an interval smaller than what is in the specification). It is also not related with extra functionality (functionality not specified by the specification that was introduced by the provider).
- **Level 3 (light):** this level includes non-conformities observed that are the result of an effort to extend the service specification (e.g., handling additional data types, or enabling the use of a larger domain in a given operation).

Classifying a given non-conformity is a complex task that may also depend on the person that is performing the analysis; as such, we opted for a basic scale for compliance problems. Our goal is to have a useful generic scale that fits, or at least is adaptable to more than one class of services. With this in mind, we opted for the minimum scale steps possible that could still be descriptive enough and useful for comparative purposes. More scale steps would increase the complexity of an already complex process, which could subvert the purpose of using such scale

(due to the nature of the process itself, as it may require human intervention in some classes of services). Despite these issues, this simple scale can be adapted to more than one service class and is extremely useful to understand the criticality of specification compliance problems, as shown in Section 4.2.4.

3.5 Conclusion

This chapter presented a generic approach for testing the robustness of software services. The approach can be instantiated to technologies typically used to build services or Service Oriented Architectures, such as web or messaging services, and consists of four essential steps. First, the **service interface** is analyzed to obtain a list of operations and parameter information to be used in a following **workload generation** step. Afterwards, **robustness tests are generated and executed** and the tests **results are characterized** according to three perspectives (failure severity, service behavior, and specification compliance). Aspects like the fault injection location and the fault model influence the generation and execution of tests. However, the execution profile should generally follow three phases: pre-injection, injection, and post-injection. Compliance with this execution profile and with the four major steps enables an easier standardization of tests results, which is useful when there is the intent of comparing software services.

The approach proposed is generic and concrete instantiations for web services and Java Message Service middleware are presented in Chapter 4. Here, tests are performed at the client-side and show the utility of testing services for robustness. On the other hand, Chapter 5 shows the application of our testing approach at the server-side, illustrating its utility in a development environment, and in particular the advantage of using the extra information available at the server-side. Both chapters contribute with a wide range of robustness testing application scenarios and have their usefulness later confirmed in the experimental evaluation presented in Chapter 7.

Chapter 4
Client-side Robustness Testing of Software Services

In this chapter we present the instantiation of the proposed robustness testing approach for two key technologies used in services environments: SOAP web service implementations and messaging middleware, Java Message Service (JMS), in particular (Erl 2005). The proposal applies client-side generation of robustness tests, which essentially consist of specific instantiations of the fault model presented in the previous chapter. The results of the tests are classified using all perspectives presented earlier, whenever possible. As we will see in the experimental evaluation presented in Chapter 7, even with faults injected at the client-side, the proposed approach is very effective in disclosing major robustness problems in both classes of services.

The chapter is organized as follows. Section 4.1 illustrates the implementation of our approach in SOAP web service implementations. Section 4.2 focuses on robustness testing of Java Message Service middleware. In particular, these sections show the adaptation of each of the main components of the robustness testing procedure (interface

analysis, workload generation, tests generation and execution, and service characterization) to these particular technologies. Section 4.3 concludes the chapter.

4.1 Testing web services for robustness

The instantiation to SOAP web services of the generic approach presented earlier in Chapter 3 has been implemented in a concrete testing tool, designated *wsrbench* (available at http://wsrbench.dei.uc.pt), which is able to test services from a client point of view. The tool provides a web-based interface that allows users to configure and execute tests, and also to visualize and analyze the results. *wsrbench* is free, open-source, and easy to use, requiring only a simple registration and posterior authentication process. The steps of the testing approach are presented in the following section and the detailed technical aspects about *wsrbench* can be found in Annex A.

4.1.1 Analysis of the web service interface

Before generating and executing the workload and the robustness tests (see sections 4.1.2 and 4.1.3, respectively) we need to obtain some definitions about the web service operations, parameters, data types, and domains (when available). As mentioned before, a web service interface is described as a WSDL file. This XML file is automatically processed to obtain the list of operations, parameters and associated data types. The information describing the structure and type of all inputs and outputs of each operation is usually found in a XML Schema file (a XSD file that describes the structure of an XML object), which is referenced by the original WSDL (W3C 2008a; Curbera et al. 2002).

The next step consists of gathering information on the valid domains for all input and output objects. For this purpose, the XSD file, that describes all parameters, is searched. Although this file may include information about valid values of each parameter (provided that XSD schema restrictions are defined), practice shows that it is rare to find those values expressed in a WSDL/XSD pair (frequently due to development tools limitations). This way, the tester is allowed to provide information on the valid domains for each parameter, which include values for parameters

based on complex data types (that are decomposed in a set of individual parameters).

Table 4.I shows an example of how the benchmark user can specify the domains for each parameter. This is the basic information needed to support the workload generation and tests execution.

Table 4.I – Example of the specification of parameters for web services operations.

Parameter	Data type	Domain specification	Description
name	String	[a-z|A-Z]{2,16}	The valid domain includes all the strings with a minimum of 2 and a maximum of 16 characters. The valid characters are only the letters from A to Z (both uppercase and lowercase)
number	Integer	[0-199]	The valid values are integer numbers from 0 to 199
return	String	[OK|NOK]	The method return value admits only two strings: OK and NOK

Figure 4.1 presents an excerpt of a WSDL file for a web service named *ValidateService*. The service provides the following operation to clients: *ValidateObject (String name, int number)*. The figure illustrates how the information introduced by the user maps to the WSDL definitions. Essentially, the service definitions consist of the name and address of the service, which are associated to a *binding* element that basically contains a list of all operations that compose the service and includes references to their input and output parameters. The structure and type of these parameters are described in *message* elements. In this case, all information is described in the WSDL file; however, it is frequent to find only a reference to the input/output parameters, which are then described in a separate XML Schema file.

4.1.2 Workload generation

As described in Section 3.2, there are various types of workloads that can be used in the services context. In web services, a real workload can be an option if the service being tested is already under real usage and real

requests can be extracted (which is rarely the case). Also, a realistic workload can be used, but that also requires knowing the typical usage profile of the service under testing. The problem is that there is no workload that can fit all web services specificities (because different web services have different interfaces and behaviors). In this sense, a robustness testing approach should provide a way for automatically generating synthetic workloads based on the services definitions and domains.

In our approach the following options can be used for the generation of a workload for web services:

- **User defined workload:** the benchmark user implements (or reuses) a workload emulation tool. This tool, based on knowledge about the service being tested, should generate a realistic or synthetic workload and can be integrated in the proposed robustness testing setup. To simplify the creation of the workload

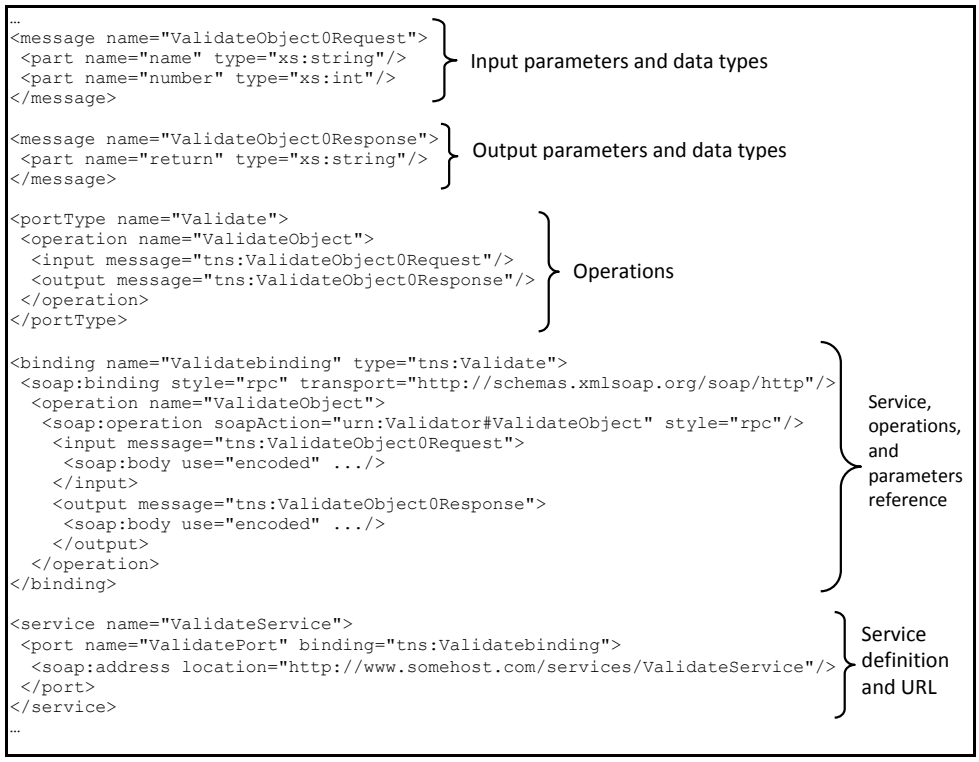


Figure 4.1 – Example of a WSDL file.

emulator, there are some client emulation tools like soapUI (Eviware 2011) that can be easily used.

- **Random workload:** this workload is generated automatically using the web service definitions mentioned before. For each operation, several sets of valid input values are randomly generated (taking into account the input domain of each parameter), following a uniform distribution. The user must specify the intended total number of executions for each operation, which determines the size of the workload.

It is important to emphasize that one of the problems related to the random workload is that the representativeness of the web service calls is not guaranteed. For instance, some services may require highly diverse inputs to achieve satisfactory code coverage. Depending on the service under test, the user can define a more extensive workload in order to achieve higher code coverage (when the code coverage depends on the input values). This definition can be more effective than a random workload if the user has some specific knowledge of the services being tested (e.g., if he is the service provider). Note however that, in this chapter we are considering tests executed from a client point-of-view, so no access to information about the coverage of the workload is assumed (a similar process considering access to additional information and workload coverage is proposed in Chapter 5).

4.1.3 Robustness tests definition and execution

Figure 4.2 presents the proposed **setup** to test the robustness of web services from a client-side perspective. As in this case we assume no control of the service at the server, fault injection is executed at the client-side (represented by the red square in the figure). Obviously, with this setup, the source code of the web service being tested is not required for the robustness tests generation and execution.

At runtime, during the execution of the robustness tests, the fault injector component (at location B, in Figure 4.2) intercepts all SOAP messages sent to the server by the emulated clients (created by the '*Application*' component). The XML is modified according to the robustness test being performed and then forwarded to the server. Each server response is logged by the robustness-testing tool and used later on to analyze the

behavior of the web service in the presence of the invalid call parameters injected.

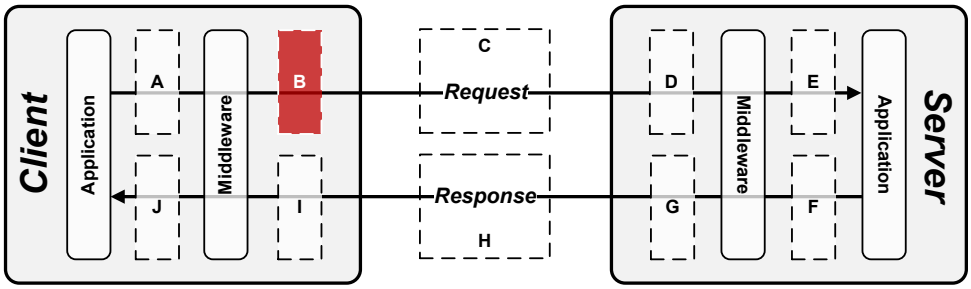


Figure 4.2 – Fault injection location used for the client-side web services robustness tests.

The generation of robustness tests depends on the **definition of the fault set**, based on a fault model. Following the rules described in Section 3.3.3, we created a set of specific mutations for web services, as described in Table 4.II.

Table 4.II – Parameter mutation rules.

Parameter type	Parameter mutation
String	Replace by null value
	Replace by empty string
	Replace by predefined string
	Replace by string with nonprintable characters
	Add nonprintable characters to the string
	Replace by alphanumeric string
	Set a number of characters below the minimum acceptable size
	Add characters to overflow max size
	Malicious predefined string (see examples in Table 4.III)
Boolean	Replace by null value
	Replace by empty value
	Replace by predefined value
	Add characters to overflow max size

Parameter type	Parameter mutation
Number	Replace by null value
	Replace by empty value
	Replace by -1
	Replace by 1
	Replace by 0
	Add one
	Subtract 1
	Replace by maximum number valid for the type
	Replace by minimum number valid for the type
	Replace by maximum number valid for the type plus one
	Replace by minimum number valid for the type minus one
	Replace by maximum value valid for the parameter
	Replace by minimum value valid for the parameter
	Replace by maximum value valid for the parameter plus one
	Replace by minimum value valid for the parameter minus one
List	Replace by null value
	Remove element from the list
	Add element to the list
	Duplicate elements of the list
	Remove all elements from the list except the first one
	Remove all elements from the list
Date	Replace by null value
	Replace by empty date
	Replace by maximum date valid for the parameter
	Replace by minimum date valid for the parameter
	Replace by maximum date valid for the parameter plus one day
	Replace by minimum date valid for the parameter minus one day
	Add 100 years to the date
	Subtract 100 years to the date
	Replace by the following invalid date: 2/29/1984
	Replace by the following invalid date: 4/31/1998
	Replace by the following invalid date: 13/1/1997
	Replace by the following invalid date: 12/0/1994
	Replace by the following invalid date: 8/31/1992
	Replace by the following invalid date: 8/32/1993
	Replace by the last day of the previous millennium
	Replace by the first day of this millennium

Regarding the detection of robustness problems that may also represent security vulnerabilities, we propose an attack list focusing on SQL Injection and XPath Injection, two key vulnerabilities frequently found in web services (Vieira, Antunes, and Madeira 2009). The set of attacks considered is based on the compilation of the types used by a large set of commercial and open-source scanners, namely the latest versions of HP WebInspect, IBM Rational AppScan, Acunetix Web Vulnerability Scanner, Foundstone WSDigger, and wsfuzzer. This list was analyzed and complemented using practical experience and based on information on web application attack methods available in the literature (e.g., (The Open Web Application Security Project (OWASP) 2010; Stuttard and Pinto 2007; The Web Application Security Consortium (WASC) 2010)). For the time being, the list includes 146 attack types, available at (Laranjeiro 2011) (see Table 4.III for examples on SQL Injection).

Table 4.III – Examples of SQL Injection attack types.

Parameter Mutations
" or 1=0 --
" or 1=1 --
" or 1=1 or ""="
' or (EXISTS)
' or uname like '%
' or userid like '%
' or username like '%
' UNION ALL SELECT
' UNION SELECT
char%2839%29%2b%28SELECT
char%4039%41%2b%40SELECT
" or 1=1 or ""="
' or ''='

As discussed previously, besides the main profile aspects, such as the type of workload and the type of setup to use, there are some **configuration aspects that can be tuned**, according to the tester's goals or the service being tested. These aspects are, as discussed in Section 3.3.5: the duration of the pre-injection, injection, and post-injection periods; replication of faults in each injection period; system restoration; and a strategy choice to localize the cause of failures.

The duration of the pre-injection, injection, and post-injection periods depends on the number of service calls, which should be defined by the user. For example, in the web service tests presented in Chapter 7, we opted for a single call in each of the three distinct periods referred above. Other values could have been chosen; however, as we will be executing the tests at the client-side we have no additional knowledge that can enable us to make an informed choice for these values. Also, in this black-box experiment, we were unaware of particular parameters that could be sensible to tests using each fault more than once. As such, and in order for us to maintain the same test profile for all services, we opted for no fault replication for all executed tests.

At the client side, there is no option to restore the remote system state. This can obviously limit the understanding or any localization process of the failure cause, as there is no access to the server or source code. Replaying the tests can also be of little use as there is no option to restore the system state. Still, the tests continue to serve their main purpose of disclosing and also describing robustness issues (as shown in Chapter 7). Despite the fact that with the client-side setup some configurations are not allowed, there is still a large set of configuration combinations that can be used and can provide the tester with more (or less) robustness issues disclosed. The tester has the final decision on which parameters to use and, frequently, a better knowledge of the services being tested can guide the tester in using configuration parameters that are different from the ones presented here.

4.1.4 Web services robustness characterization

Following the characterization scheme described in Section 3.4, we propose a concrete scheme that enabled us to characterize a large set of public web services in our experimental evaluation (presented in Section 7.2). Due to the fact that a categorization scheme can become tightly coupled to the class of services being tested, it is important that it is as generic as possible, while maintaining its descriptive qualities.

The robustness of web services, tested at the client-side, can be classified based on two perspectives: failure severity and detailed behavior. In practice, this corresponds to a description of the test results using set of **failure modes**, complemented with a detailed **tag-based categorization of the service responses**. Notice that the third perspective (compliance to a

service specification), presented in Section 3.4, cannot be forced, as there is no guarantee that the benchmark user has access to the services specifications.

A possible approach would be to adapt and use the CRASH scale (P. Koopman and DeVale 1999) to characterize the service behavior. However, as referred in Chapter 3, this scale is not adequate for tests that are executed at the client-side. This way, our proposal for a **failure mode classification scale** includes only those failures that can be effectively observed while conducting robustness testing from the consumer point-of-view. Based on our observations during a large set of web services robustness testing experiments (see Section 7.2), we propose the use of the CCE scale, which consists of the following failure modes:

- **Correct:** the web service response in the presence of an invalid input (or a domain-valid but malicious input) is correct, i.e. the web service responds with an expected exception or error code.
- **Crash:** an unexpected exception is raised by the web service and sent to the client application.
- **Error:** the service replies with an expected object that however encapsulates an error message that indicates the occurrence of an unexpected internal problem.

In practical terms, robustness characterization includes the automated analysis of the responses obtained in order to distinguish valid replies from replies that reveal robustness problems in the service being tested (e.g., null references, SQL exceptions, array out of bounds exceptions, etc.). A key difficulty is that, in some cases, automated identification is not enough to decide if a given response is due to a robustness problem or not (e.g., in many cases it is difficult to automatically decide whether a given response represents an expected or unexpected behavior). This way, the benchmark user may need to perform a manual identification of the doubtful cases. Although it might evolve some effort, manual validation is typically a quite easy task. In fact, it is normally straightforward for developers and testers to classify a given response as expected or not expected.

The second analysis perspective referred – **tag-based categorization of the service responses**, is useful to understand the source of the failures (i.e., defects). Obviously, that depends on several specificities of the services

being tested (e.g., programming language, external interfaces, operational environment), which complicates the definition of a generic classification scale. Based on our experiments on public web services (see Chapter 7) and following on the proposal presented in Chapter 3, we propose the use of the following tag-based behavior classification system presented in Table 4.IV.

The tags included in our classification system were designed to be as comprehensive and generic as possible. On one hand there was an effort to minimize the number of tags, but, on the other hand, we wanted our tags to be as descriptive as possible. In fact, for every problem, any tag or tag combination can be decisive in helping a developer to produce an adequate fix. For instance, when the problem is marked with the *data access operations* tag, this is a strong indicator that the developer should focus his attention on the persistence modules of his application. Due to the importance and difficulty of creating clear and generic, but also descriptive tags, the tag-based classification was created iteratively while analyzing near half-million distinct service responses obtained during the experimental evaluation presented in Section 7.2.1. It is important to emphasize that the out-of-domain tag was not observed in the experimental evaluation targeting public web services (we had no information about the domains of the services being tested). However, we decided to include it, as it is an obvious behavior that can be observed when testing services and when there is accurate service domain information.

As a final note, it is fairly evident that it is not feasible to design a generic classification system that is able to describe all existing web service behaviors. In fact, because web services generally encompass the use of highly diverse systems and are based in technology that suffers continuous advance and transformations, a tag-based system like the one we propose will always need to be extended or adapted in the future. In this sense, we do not provide this as a closed classification. Instead, the benchmark user is free to use this classification, extend it, or devise any other classification tailored for the specific web services being tested.

Table 4.IV – Tag classification system for web services.

Tag	Description
Server resource disclosure	Information about the server's filesystem or a physical resource is disclosed
Conversion issues	A conversion problem exists in the service
Wrong type definition	The service operation expects a value whose type is not consistent with what is announced in the service's WSDL file
Data access operations	A problem exists related with data access operations
Specific server failure message	An exception is thrown and application or development specific information is revealed. This information is however generally too vague or too context-specific to allow us an association with another tag
Persistence error	An exception is thrown indicating a persistence-related problem. This is typically an SQL exception that is thrown as a consequence of improper parameter handling
Argument out of format	The operation requires a restriction on a parameter's format. However, no restriction is specified in the WSDL file, allowing clients to invoke the operation with an out-of-format parameter
Wrapped error information	An error response is wrapped in an expected object. The response indicates the occurrence of an internal error
Array out of bounds	Occurrence of an array access with an index that exceeds the limits of the array (upper or lower)
Null references	A null pointer or reference exception is thrown by the server application
Command or schema disclosure	An internal command is totally or partially disclosed (e.g., an SQL statement is revealed), or the data schema is revealed (e.g., the table names in a relational database are revealed)
Arithmetic operations	An indication of an arithmetic error is returned by the operation
Division by zero	The service indicates that a division by zero has been attempted
Internal function name disclosure	The name of an internal or system procedure is disclosed (e.g. a database stored procedure)
System vendor disclosure	System vendor information is disclosed (e.g. database or operating system vendor)
Overflow	The service operation is unable to properly handle a value that is larger than the capacity of its container, indicating the occurrence of an overflow error.
System instance name disclosure	The name of a system instance is revealed to the client (e.g., a database instance name)
System user disclosure	A system username or password is exposed to the client (e.g., the username used to connect to a database or the operating system username)
Out-of-domain responses	A response outside the valid domain is delivered to the client
Other	Any other service response that does not fit into any of the previous categories

4.2 Testing messaging middleware for robustness

Services environments frequently make use of messaging services. In practice, these are communication frameworks capable of providing a loosely coupled relationship between clients and servers (Erl 2005). JBossMQ, Active MQ, or Open Message Queue (Red Hat Middleware 2008a; Sun Microsystems Inc. 2007a; Apache Software Foundation 2008b) are examples of specific implementations of a well-known messaging framework, the Java Message Service API (Sun Microsystems, Inc. 2002). Another example of a well-known messaging framework implementation is the Microsoft Message Queuing, also known as MSMQ (Microsoft Corporation 2010). In this section, we propose a method that allows testing the robustness of server-side middleware (JMS) while applying robustness tests at the client-side.

4.2.1 Analysis of the middleware interface

In order to test the robustness of messaging middleware solutions it is vital to understand the regular behavior of messaging frameworks. As seen earlier in Section 2.1.2 the JMS Message is a central component in what concerns the interaction between distinct parties (i.e., JMS producers and consumers). Therefore, it is an interesting interface for applying the mutation rules (i.e. submitting the robustness tests). Considering this, and because we are targeting the middleware, it is important to understand the operation mode and domains of that middleware, as these may provide important insights to the tester.

When the goal is to apply robustness tests to a middleware component accessed by an application layer (see Figure 3.1 in Section 3.3.1), usually there is little need to understand the domains of the underlying application. In fact, this underlying application layer can even be specifically created only to extract output information of the robustness tests (e.g., understanding if a message was handled correctly by the middleware). Still, it is important to know the domains of the middleware component itself, so that the robustness tests can focus more effectively on real limit conditions (e.g., testing if the middleware can handle a number outside the valid domain limit). Currently, in the case of JMS, domains can only be found in the JMS specification document (Sun Microsystems, Inc. 2002). This document is written in natural language, which

complicates the automatic integration of the JMS domains in a testing procedure, such as ours. The solution is to manually provide the domains of the objects being tested and use this information to generate the workload and robustness tests.

Another key aspect is that, as we will see ahead, the JMS message has a complex structure, with several parameters aggregated in three distinct parts. More importantly, the JMS API includes several different ways of setting these parameters (e.g., multiple methods exist for setting particular message parameters) and this is also information that has to be extracted from the specification document and incorporated into the testing procedure, so that it is considered when executing the tests.

4.2.2 Workload generation

To exercise the JMS provider we need to have a workload composed of JMS messages. This workload is the work to be performed during the testing phase and can be any of the four types described earlier in Section 3.2, namely: real, realistic, user defined, or random workloads (the latter two represent synthetic workload types).

As 'realistic' workloads, it is important to indicate, for instance, the SPECjms2007 benchmark (Standard Performance Evaluation Corporation 2007b). This is a standard benchmark developed by SPEC that aims at evaluating the performance of enterprise message-oriented middleware servers based on JMS, and it guarantees a large degree of representativeness and portability. Despite the fact that SPECjms2007 can be a strong option for the workload, not all users may have access to it (as it is not freely available). An easy alternative is to use a synthetic workload (either user defined or random), which are usually simpler to manage and use than realistic or real workloads (this is the case in the experimental evaluation presented in Section 7.2.2).

4.2.3 Robustness tests definition and execution

In a Java Message Service (JMS) environment a connection factory creates connections that can be used to produce sessions. Each session is able to create message producers or consumers, which in turn have the ability to send/receive messages to/from a destination. Figure 4.3, adapted from

(Jendrock et al. 2006), illustrates the **JMS environment and setup for robustness tests**.

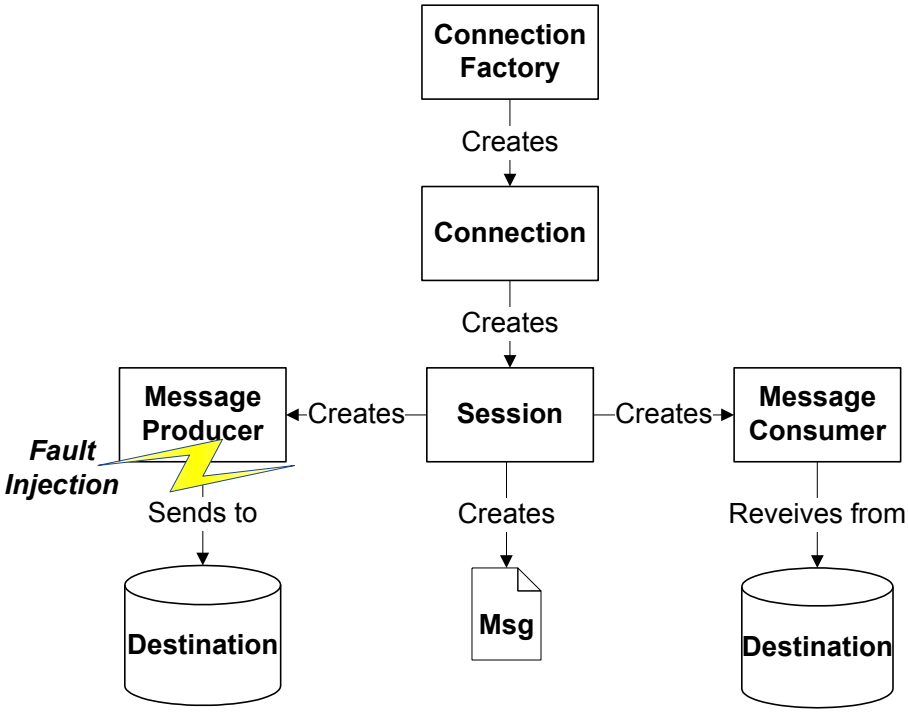


Figure 4.3 - The JMS API programming model.

Our approach for testing the robustness of JMS middleware depends on the JMS environment specificities (coarsely depicted in Figure 4.3) and consists of modifying messages immediately before they are sent from a producer to a server that may be linked to one consumer (in the case of point-to-point messaging) or to several consumers (in the case of publish/subscribe messaging). As in the web services example, fault injection is performed at the client-side and generally the setup is the same as the one depicted earlier in Figure 4.2 (Section 4.1.3). With this setup, any occurring parameter manipulation will take place at the client side; our target, however, is the server-side middleware and the main goal is to understand the behavior of the middleware service in the presence of faults (i.e., in the presence of the mutations applied to the JMS messages).

Message parameter modification for JMS environments is based on the abstract model presented in Section 3.3.3. The specific **fault set** is

essentially an adaptation to the JMS middleware context and it focuses the three distinct JMS Message parts, as described in the following points:

- **Header:** contains several fields used by clients and providers to identify and route messages.
- **Properties:** mainly used as an extension to the header, it provides additional functionality when required by the business logic of client applications (i.e., the properties in each message are application dependent).
- **Body:** the message content itself.

The fault set proposed considers not only the parameters included in these three parts, but also the multiple ways of setting them (e.g., setting properties using the several methods available: setBooleanProperty, setStringProperty, setObjectProperty, etc.).

The JMS specification defines several message *header* fields. Some of those fields are intended to be set by the client application (see Table 4.V). For instance, a client may need to set the JMSReplyTo field to specify the object to which a message reply should be sent. However, there are other fields that are automatically set by the JMS implementation just after the client sends the message (by executing the 'send' or 'publish' method) and before that message is sent to the destination. The JMS implementation must handle and correctly set all fields that are of its own responsibility, even if the client did set them. For example, even if a client sets the JMSExpiration field, the JMS send or publish method will overwrite it. Table 4.V presents the whole set of header fields available, their types and how they are set (Jendrock et al. 2006). It is important to emphasize that header fields are clearly parameters that robustness testing must cover, as they define middleware level behaviors that must be tested for correctness under faulty conditions.

Message properties are also relevant for robustness testing. All allowed types and conversions are defined by the JMS specification and summarized in Table 4.VI, which shows that a value written using a given data type (a row in the table) can be read as one or more data types (shown by marks in the corresponding columns intersections). For example, after setting a Boolean property (first row in table) it is possible to retrieve it using a getBooleanProperty method or a getStringProperty

method (in this case it returns a string representation of the Boolean value).

Table 4.V – Header fields present in a JMS message

Header field	Type	Description	Set by
JMSDestination	javax.jms. Destination	Contains the destination to which a message is being sent	Send or publish method
JMSDeliveryMode	int	Delivery can be persistent (to be used when loosing messages is expensive) or non-persistent (when lost messages are tolerable)	
JMSExpiration	long	Defines the message expiration time limit. It is the sum of the time-to-live specified by the client and the GMT at the time of send or publish	
JMSPriority	int	Defines the priority level for the message (from 0 to 9)	
JMSMessageID	String	Contains a value that uniquely identifies each message	
JMSTimestamp	long	Contains the timestamp of the moment when the message was handed off to a provider to be sent	
JMSCorrelationID	String	Used by clients to link different messages	Client
JMSReplyTo	javax.jms. Destination	Represents an object to which a reply to this message should be sent	
JMSType	String	Contains the message type (may be required by some providers)	
JMSRedelivered	Boolean	Specifies if this message is being redelivered. If a client receives a message with this field set, it is possible that it was delivered earlier but its receipt was not acknowledged	JMS Provider

The unmarked cases must throw a JMSException as they represent exceptional behavior (e.g., setting a byte property and then retrieving the value as a Boolean, float or double should raise a JMSException). In addition it is possible to set an object of an unspecified type by using a setObjectProperty method (the acceptable object types are limited to the

primitive objectified types) and retrieve it with a suitable available getter method. In other words, the setObjectProperty method can be used to set, for instance, a Boolean, that can then be retrieved either by using the getObjectProperty method or getBooleanProperty.

Table 4.VI – Message properties types and conversion

set\get	boolean	byte	short	int	long	float	double	string
boolean	X							X
byte		X	X	X	X			X
short			X	X	X			X
int				X	X			X
long					X			X
float						X	X	X
double							X	X
string	X	X	X	X	X	X	X	X

Testing the correctness of these conversions requires a later verification of the value set. The goal is to check if the JMS middleware is accurately following the specification. Note that non-compliance between the JMS specification and the way it is implemented may compromise the client application's business logic. Thus, properties conversion testing is a key aspect that should be included in a robustness test.

Several data types are allowed for the *message body*: Stream, Map, Text, Object or Bytes. At first sight, it may seem less interesting to mutate the message body to exceptional values. In fact, this is most likely to trigger problems in the client applications instead of triggering faults in the middleware itself. Nevertheless, for the sake of completeness we have decided to include tests that use valid or null body objects in the fault model.

As previously, we propose a set of mutation rules based on the fault model described in Section 3.3.3. These rules are described in Table 4.VII and include a generic 'Object' data type. The new rules associated with this data type are useful to test some limit conditions used by the JMS specification document.

Table 4.VII – Parameter mutation rules

Parameter type	Description
String	Replace by null value
	Replace by empty string
	Replace by predefined string
	Replace by string with nonprintable characters
	Add nonprintable characters to the string
	Replace by alphanumeric string
	Set a number of characters below the minimum acceptable size
	Add characters to overflow max size
	Malicious predefined string (see examples in Table 4.III)
Number	Set 0
	Set +1
	Set -1
	Set maximum number valid for the type
	Set minimum number valid for the type
	Set maximum number valid for the domain
	Set minimum number valid for the domain
Object	Set a null object
	Set a non serializable object
	Set a correct target class empty object
	Set an objectified primitive data type (Boolean, Byte, Short, Int, Float, Double and String)
	Set common dataype (List, Map and Date)

The **tests execution profile** is based on the 3-phased (A, B, and C – pre-injection, injection, and post-injection periods) profile presented earlier in Figure 3.3 (Chapter 3). Specifically, the JMS middleware tests closely follow the configuration for the web services tests presented in Section 4.1. The main differences to note are: only one operation is tested; the setup enables us to restore the system after each sequence of three phases (A, B, and C); and the JMSMessageID parameter is tested with duplicate faults (i.e., the same type of fault is applied to two consecutive messages). These aspects are described in the following paragraphs.

In the case of JMS, and, in particular, in the experimental evaluation presented in Chapter 7, we focus on testing a single operation. Among the several operations that could be considered for testing, the 'send' operation is the most crucial one as it enables communication between the

distinct interacting parties (this operation is responsible for sending the JMS message to a destination). So, from the tests execution profile point-of-view, it is a view of a service with a single operation. Despite this, all parameters are tested (including header fields, properties, and body) and message delivery is always verified at the consumer for correctness and consistency.

Tests are executed at the client side; however, as we will see in the experimental evaluation, having access to the server infrastructure allows us restore the system state every time after phase C (post-injection period), enabling a more isolated view of the effects of applying each fault and eliminating the need for reverse-replaying the tests. Finally, the detailed knowledge (obtained from the JMS specification) of the roles of the message parameters involved allows us to test one of the parameters (JMSMessageID) more exhaustively. In fact, in all three periods, after each message is sent, reception follows up and response content is verified. The only exception to this rule occurs when mutations are applied to the JMSMessageID. As this is a unique identifier, it is interesting to send two consecutive messages with the same identifier before trying to receive any. In this way we can see if duplicate identifiers affect message delivery.

4.2.4 Middleware behavior characterization

Regarding behavior characterization, in the particular case of JMS middleware we make complete use of our three-dimensions approach (i.e., severity, behavior, and compliance). Taking in consideration that we have access to the server infrastructure, an adaptation of the CRASH scale (P. Koopman et al. 1997) is usable as **failure model** (as proposed in Section 3.4). Also, we can add that the experimental evaluation presented ahead validates that the majority of the CRASH failure modes are in fact observable, confirming the scale as an adequate choice.

The second perspective, **behavior classification**, is also considered. However, in this case there is no possibility of creating a detailed set of tags, as the experiments using JMS are more limited (in Chapter 7, we present tests executed over three JMS providers), resulting in a smaller number of tests, when compared with the web services tests, as discussed before. A very large experimental evaluation of JMS providers is not feasible as there are not many JMS implementations available to use.

There is also no evidence that the tags created for web services (see Section 4.1.4) are directly usable in the JMS context. In fact, a very large number of the web services behavior tags will not be observable in the JMS experimental evaluation. However, the base set of tags presented in Section 3.4 is perfectly usable and can be applied to an experimental environment such as the one presented here.

Finally, as in this case we have access to complete knowledge about parameters roles and domains, it is possible to apply the three level **compliance classification** proposed in Section 3.4. As indicated earlier, a compliance or conformity problem is observed when acceptable inputs (as defined by the specification) trigger a problem that should not occur (the inputs were acceptable). For illustrative purposes, consider the following example of a severe non-compliance: the specification states that a JMSException must be thrown if the JMS provider fails to send a message due to some internal error. If one of the injected faults causes this behavior and the parameter value is acceptable (as defined by the specification) this represents non-conformity to the specification and can be classified as a Level 1 non-conformity (i.e., severe, as it affects regular service operations).

4.2.5 JMS testing and Aspect Oriented Programming

The JMS environment poses some interesting questions when implementing robustness testing. First, although an API is defined, including the format of the message to be sent (in the form of a Java interface), there is no lower level specification for the message communication format (each vendor uses its own preferred format). This means that the same message sent using different JMS implementations can be translated into different byte sequences during transportation, as the vendor may consider diverse methods for message transmission (e.g., in JBossMQ (Red Hat Middleware 2008a) some of the message properties are serialized in a vendor specific sequence prior to the body). Since robustness testing is based on parameter tampering, this makes extremely difficult the creation of a generic proxy capable of intercepting and modifying messages. The same happens if we want to modify the message directly at the JMS provider repository. In fact, the provider may keep sent (but not yet delivered) messages in a database, a binary file or in any other repository or format. Thus, a vendor independent approach is quite difficult to achieve or simply not feasible.

Our approach uses Aspect Oriented Programming (AOP) (Kiczales et al. 1997) and AspectJ in particular (Eclipse Foundation 2008). AspectJ is an AOP implementation built for the Java programming language. We make use of this strong programming paradigm to intercept the message immediately before it is sent to the destination and after its properties are set by the JMS '*send*' or '*publish*' methods. Note that, as shown in Table 4.V, some message properties are not controlled by the client, thus mutations have to be performed after the execution of the send or publish methods.

As explained in Section 2.3.1, the use of AOP allows us to inject cross-cutting concerns into any application in a non-intrusive way. In practice, starting with the fault-injection code (that implements the robustness testing approach presented) and the original JMS middleware implementation, we can easily create a modified JMS implementation that integrates our fault-injection tool in a transparent way. The goal is to provide a way to perform fault injection immediately before messages are sent to their destination. This approach has some advantages over the use of proxies for message interception or mutations at the JMS repository:

- It provides a non intrusive way (from the programmer point-of-view) to intercept messages;
- It enables message tampering in a generic way that can be applied to any messaging implementation that follows the JMS specification;
- It does not require an extra software application in the setup (e.g., a proxy) and the associated changes in the middleware default configuration.

Our robustness testing implementation is generic. The only aspect that is dependent on the JMS middleware being used is the exact signature of the Java method to be intercepted. If this is not disclosed by the vendor in the documentation, it is necessary to search the JMS implementation source code to identify the method that implements the API '*send*' or '*publish*' and select the last method signature before the message is actually sent to the destination. Anyway, this is typically an easy task that represents only a few minutes effort. An illustrative example can be found in Figure 4.4. Keep in consideration that this example represents the client-side middleware layer code and that normally the client only knows the API

methods, which are later bound to their implementations typically by using JNDI (Java Naming and Directory Interface).

```
class ProviderSender implements javax.jms.MessageProducer
{
    public void send(Message msg)          <-- API method
    {
        msg.setFieldX();
        msg.setFieldY();
        doSendMessage(msg);                <-- Method to intercept
    }

    public void doSendMessage(Message msg)
    {
        // actually send the message to
        // its destination
    }
}
```

Figure 4.4 – Identification of the method to intercept.

4.3 Conclusion

This chapter presented the adaptation of our robustness testing approach to two popular choices for implementing services: web services and messaging middleware (JMS, in particular). Both instantiations share the four basic steps: interface analysis, workload generation, tests generation and execution, and service characterization. Despite this, there are differences that result from the specificities of each technology.

Clearly, the interface analysis step has to be performed in a different way in each technology, as different artifacts are used to specify the interfaces. In web services, the WSDL document is responsible for describing the service, whereas in JMS the API documentation describes all interfaces and behaviors. Distinct options also exist for the workload generation step (although they fit in the major categories described), and there are differences regarding the tests generation and execution. Among these differences we highlight the more reduced number of different data types

available in a JMS message (when compared to the data types available for use in a web service operation), which limits the number of faults present in the fault set. In addition, there is a particular parameter (JMSMessageID) for which we consider replicated faults. Although replicated faults may be useful in some web services scenarios, their utility is clear in JMS middleware.

Another difference between the two technologies is associated with the fact that there is a detailed service specification for JMS middleware, and there may not be one when testing web services. This results in the possibility of characterizing JMS middleware implementations in terms of specification compliance, which may not be possible when applying the tests to web services (such as when tests are executed on public web services). Despite this, both can be characterized in terms of failure severity and service behavior (and also in terms of compliance when a specification exists).

Regardless of the specific issues associated with the use of our approach in these two very popular service technologies, we show that it is in fact possible to adapt our approach at the client-side to distinct environments. This provides strong indications regarding the use of our approach in services environments. As we will see in Chapter 7, even with faults injected at the client-side, our approach can be very effective in disclosing major robustness problems in both classes of services. The next chapter discusses how the approach can be adapted to test services at the server-side in the context of web services, allowing a more detailed analysis of their behavior.

Chapter 5
Server-side Robustness Testing of Web Services

This chapter proposes the adaptation of the robustness testing approach presented in Chapter 3 to server-side testing of web services, one of the most popular technology for implementing Internet services. Our testing technique can now take advantage of additional information that is available when tests are executed at the server-side; in particular, information about the service's domains and internal structure, and also infrastructure access. A domain expression language is proposed allowing the full definition of service domains, which can be used to generate tests that focus on more complex input validation conditions. Thus, this chapter introduces significant features to the robustness testing approach (available when tests are executed at the server-side).

Testing services for robustness requires performing several choices, according to the purposes of the tests themselves. Client-side testing is effective in many cases (and it is the only option in some cases); however, server-side testing brings several advantages to the tests. In particular, by having a greater knowledge about the service (e.g., there is access to the service code or bytecode), it is possible to redefine the services inputs to include domain information that is typically not available in regular web services (e.g., domain dependencies between different input parameters).

This extra information can be used to generate tests that focus on difficult input validation conditions in a more effective manner. Additionally, such information allows generating a workload able to cover the code in a more complete way (access to the code enables us to obtain a measure of the workload coverage).

At the server side it is possible to view the service as a white-box that accepts inputs from different sources. The main source that can trigger the execution of a service operation is obviously the client input, however and considering that services can invoke other external services during their regular operation, the result of invoking those external services can also be seen as input to the main service, and thus object of testing. A related aspect is that these external contact points can provide regular inputs to the service, but can also provide exceptional inputs under the form of Exceptions (Goodenough 1975). Therefore it is reasonable to extend the robustness tests fault model to include such kind of exceptional behaviors.

A feature that gains a higher dimension when executing tests at the provider side is that the results of the tests can be observed in a more detailed manner. As an example, when a service does not reply to a particular test it is possible to assess if the service crashed or simply hanged. This means that more failure modes can be distinguished, thus enabling a finer-grained categorization of the services under testing.

In summary, our proposal for server-side robustness testing is based on the four phases presented in Chapter 3 (interface analysis, workload generation, robustness tests generation and execution, and service characterization) and introduces the following additional features:

- Tests are defined based on extra domain information;
- The service under test is seen has having more than one entry point whenever it makes use of external services;
- Besides the typical invalid values used in robustness testing, it is now possible to inject Exceptions as return values of external contact points;
- A more detailed service behavior characterization is made.

This chapter is organized as follows. Section 5.1 discusses how to extend web services domain information descriptors to include domain dependencies between parameters. Section 5.2 proposes a workload

generation technique that uses feedback from code coverage tools. Section 5.3 details the generation and execution of robustness tests and Section 5.4 discusses the characterization of services robustness at the server-side. Finally, Section 5.5 concludes this chapter.

5.1 Extending input domains definition

The regular tests preparation phase previously presented in Section 4.1.1 also applies to server-side testing of web services. The main difference is that domain and domain dependencies can now be defined. In summary, this first phase consists of:

- Obtaining a list of operations, parameters, and data types;
- Obtaining or defining parameter domains;
- Defining parameter domain dependencies.

As in the approach proposed in Section 4.1, we first need to collect relevant definitions of the service under testing; this includes information about operations, parameters, data types and domains. The web service WSDL/XSD file is searched for this information. Despite this, and as previously mentioned, it is uncommon to find the valid values for each parameter expressed in a WSDL/XSD pair. Currently, this is mainly due to:

- Lack of integrated tools (and programming language support) that could be easily used to add the domain values to the service's WSDL descriptor.
- Inexistence of support for expressing dependencies between two (or more) parameters of a given service operation using WSDL or XSD. This absent feature impairs the full definition of a domain and, in what concerns robustness, a partial domain definition is ultimately useless.

To fill the gap that hinders current web service applications from fully expressing and disclosing business logic domains, we propose an extension to XML Schema (XSD) that not only allows service developers to provide detailed information on the valid domains for each parameter, but also provides strong semantics for expressing domain dependencies

between multiple parameters. This is particularly important when a client application uses an unknown or undocumented service, as it provides data that informs the client of the valid and invalid inputs and outputs, promoting interoperability. Obviously, this information can also be used to prevent robustness problems. An important aspect is that we are using existing XSD features to provide those domain expression semantics, hence maintaining retro-compatibility with existing service implementations and supporting stack tools.

Basic domain information is typically expressed by means of standard XSD restriction elements (W3C 2008a). In our case, these elements must have an *id* attribute (it is an optional attribute in XSD) so that they can be referenced when defining parameters inter-dependencies. To state these parameters inter-dependencies, while maintaining retro-compatibility with any XML Schema reader, we make use of the XSD element *appinfo*, in which we include our extended domain representation. The syntax for this representation must follow the proposed language, which we have designated as '**Extended Domain Expression Language – EDEL**'. Figure 5.1 represents the relation between EDEL and the main web service descriptors – WSDL and XSD.

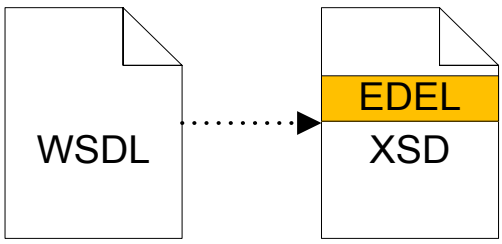


Figure 5.1 – Relation between EDEL, XSD, WSDL.

Figure 5.2 shows an example of the proposed language syntax. A schema describing the complete language can be found at (Laranjeiro 2011). As we can see, the extended domain representation consists of a set of dependencies, each one expressing a relation between two or more parameters. Keep in mind that, at this point, parameters already have their individual domains defined in standard XSD restrictions that are identified by a unique *id* attribute.

```
...
<dependencies>
  <dependency id="1">
    <function id="f1" name="aggregator" strategy="and">
      <param index="0" name="r1" />
      <param index="0" name="r3" />
    </function>
  </dependency>
  <dependency id="2">
    <function id="f2" name="starts-with" strategy="or">
      <param index="0" name="r2" />
      <param index="0" name="r4" />
      <param index="0" name="r5" />
    </function>
  </dependency>
</dependencies>
...
```

Figure 5.2 – EDEL example.

Each relation (the *function* element in Figure 5.2) is a function that uses individual restrictions to ultimately produce a Boolean output and is composed of three attributes: an *id* that uniquely identifies it; a *name* that indicates the behavior to apply to each restriction argument (each sub-element of the *function* element); and a *strategy* that specifies how the named function should be applied. The sub-elements of the *function* element (*param* elements) are basically arguments for its parent. Each *param* element specifies the name of a restriction (by referring the restriction's *id*) and an *index* attribute for the cases where a service accepts multiple complex objects of the same type. The latest EDEL version defines the *dependency id* attribute, the *function id* attribute, and the *param index* attribute as optional (*index* is zero by default), simplifying the developer's task.

For instance, in Figure 5.2, the *aggregator* function with the *and* strategy indicates that a logical AND must be applied to the values of parameters associated with restrictions *r1* and *r3*. *r1* could, for example, define that some numeric parameter should be greater than zero and *r3* could define that a string parameter should match a given regular expression. In this two-element case, the *strategy* attribute has no effect and can be removed (it is *and* by default). However, the *strategy* element makes a significant difference in the next *dependency* (*id* 2 in the figure). Here, we are defining that at least one of the parameters associated with *r4* and *r5* must start

with the value obtained at runtime for the parameter associated with *r2*. The first *param* element in all functions is the one that serves as reference for applying the function through all of the remaining children.

In order to reuse developers' knowledge we aim to provide a comprehensive set of functions based on the XPath 2.0 and XQuery 1.0 function reference list (W3C 2008b). In this way, developers do not need to learn a new language when specifying domain dependencies. Virtually any XPath reference function that returns a result is a candidate function that can be used in EDEL. However, the final result of a dependency element must be a Boolean so that our EDEL engine can logically evaluate it. Thus, all non-Boolean results provided by XPath functions that are used to express the domains of a web service should be used as parameters to functions that can compute a Boolean result.

EDEL conforms to XML Schema and is not as cumbersome as possible alternatives such as Schematron (Vlist 2007). In fact, it is specifically designed to reuse XML Schema validation aspects as much as possible, hence keeping its footprint to a minimum. Comparing to the Bean Validation framework (Bernard 2010), our work introduces dependency constraints and links validation aspects to the web service technology. A recent open source community effort created a reference implementation for the Bean Validation framework (JBoss Community 2011). This tool enables the validation of JavaBeans (Jendrock et al. 2011) and, in the future, can be connected to the web services technology, by exporting the validation restrictions to WSDL/XSD files (as XSD restrictions) in a given web service. Ideally, in the near future, developers should also be able to use their favorite programming language to automatically export validation restrictions under the form of EDEL dependencies for complete domain expression and announcement.

5.2 Workload generation and execution

The different workload options presented in Section 4.1.2 are also applicable to server-side robustness testing. However, at this stage, there are some points that can be used in favor of the testing procedure, namely:

1) There is now access to the code of the service (source code or bytecode);

2) Measures of the value of the generated workload can be extracted (e.g., the amount of code the workload can exercise);
3) The service can now have a full domain definition, including parameter dependencies;
4) It is now possible to detect if external services are being invoked.

This way, we propose here a method to generate an improved synthetic workload, trying to take advantage of the previous points. In particular, our method is able to understand how much service code is being exercised by the generated input values (i.e., the workload). Thus, the workload generation phase consists of a series of steps, as summarized in the following points and detailed in the subsequent paragraphs:

1. Choosing and configuring one or more workload generation strategies;
2. Generating a service workload;
3. Running the workload and measure the workload coverage. Return to 2 until an acceptable percentage is achieved, or go back to 1 and reconfigure/change the workload generation strategy;
4. Add domain information about any existing external service calls.

When generating input values, it is very important to get a distribution that covers the input parameter domain space effectively. In fact, for most cases it is not practical to explore the domain space exhaustively, due to the very large number of possibilities or parameter combinations. This would result in a very long testing phase that may not coexist with today's demand for fast application development and testing.

Our server-side approach supports generating valid domain values according to multiple **input generation distributions**. We make use of these distributions during the workload generation phase, which enables us to better explore the valid parameter space, ultimately resulting in higher code coverage (i.e., more application statements can be reached). Obviously this may allow uncovering more robustness problems during the testing phase, as more (potentially problematic) code areas can be reached.

The generation strategies have been adapted and uniformized from the original set supported by *benerator* (Bergmann 2008), a tool for workload

generation, and from Apache Commons Math (Apache Software Foundation 2010a), a well-known open-source mathematics package. A tool (named *wlgenerator*) implementing this set of strategies can be found at (Laranjeiro 2011). Random value generation is the default option, but depending on the service being tested, the user can opt for generating numbers and dates using any of the strategies included in the following three categories:

A. Deterministic generation

- **Step:** depending on the value of an '*increment*' parameter, it starts with the minimum or maximum value of the specified range. With each further invocation, the 'increment' value is added. For instance, for a range of numbers between 10 and 20, if '*increment*' is set to 3, the generated numbers would be 10, 13, 16 and 19.
- **Shuffle:** covers a large range avoiding duplicate values. It starts from an offset of 0 and iterates the number range with a fixed increment. After the range is covered, it increases the offset by one and reiterates the range. For instance, for an increment of 3 in a range 1 to 7, the generated numbers would be 1, 4, 7, 2, 5, 3, 6, 1, 4, (and so on; it stops when the number of generated values reaches a user defined parameter).
- **Wedge:** starting first with the lowest and then the highest available number, this strategy alternatively provides increasing small numbers and decreasing large numbers until they converge in the middle. For instance, for a number range of 1 to 7, the generated numbers would be: 1, 7, 2, 6, 3, 5, 4, 1, 7, (and so on; it stops when the number of generated values reaches a user defined parameter).
- **Multi-strategy deterministic generation:** a combination between any of the above strategies.

B. Stochastic generation

- **Random:** creates uniformly distributed random values.
- **Random Step**: depending on the settings of a provided minimum step and maximum step, it starts with the min, max or medium value of the specified range. In each further invocation, a random value is added, which is between minimum step and maximum step.

- **Gaussian, poisson, or exponential:** generates random values that follow a specific distribution. Exponential and Poisson distributions require a '*mean*' parameter that provides an indicator for the final average of the generated values. The Gaussian function requires a standard deviation parameter besides the '*mean*' parameter.
- **Multi-strategy stochastic generation:** a combination between any of category B strategies.

C. Hybrid generation

- Any mixture between strategies that fall in classes A (deterministic) and B (stochastic). For instance, for a domain interval of 10.0 to 20.0, a developer may be interested in having a Gaussian distribution from 10.0 to 15.0, and a step distribution from 15.0 to 20.0.

Obviously, the approach also supports non-numeric data types as Booleans and Strings. Strings are exceptional cases in the sense that, in our methodology, only a regular expression pattern (Friedl 2006) needs to be defined, so that our tool can generate valid Strings, according to that expression. For instance, if the user defines *ab+c** as a regular expression that defines the acceptable values for a particular String, our tool simply generates a value that matches the defined pattern (*abbccc*, for example) and uses it for the workload. Similarly to other data types, this regular expression can be defined in the parameter's XML Schema under the form of a Schema Restriction.

For the **workload generation** we integrated a set of well-known tools to support the process proposed in Figure 5.3. Some of these tools are specific to Java, but similar ones exist for all major languages. Additionally, Java alternatives to the tools presented in Figure 5.3 (and described in the next paragraphs) also exist and can be used. This is a possible setup that is already integrated in our own custom tool.

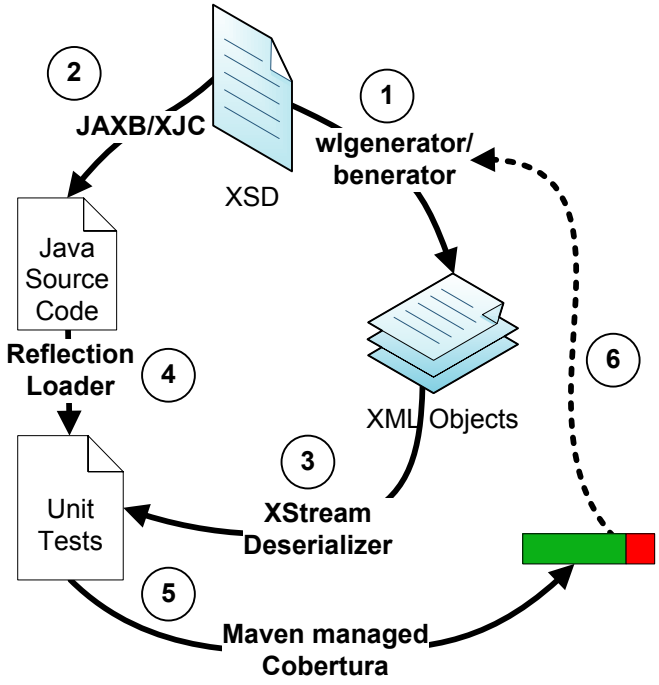


Figure 5.3 – Workload generation and execution.

Using the XSD file as starting point, we automatically generate a synthetic workload by using *benerator* (stage 1). Benerator (Bergmann 2008) is able to read XML Schema files and, using the domain information present in each schema, generates a set of XML files containing values that will be used later on to exercise the target service. However, as benerator does not implement all the generation strategies mentioned before (e.g., poison, multi-strategy, and hybrid), the recommended alternative is to use *wlgenerator*, a custom built tool that integrates benerator and Apache Commons Math. Our tool was specifically designed to automatically introspect and fill complex objects. Its default behavior is to work with class instances directly (i.e., can be used to fill objects resident in memory), but like benerator, it can also produce XML documents.

In order to use the generated values, we need to create programming language level objects that accurately represent the structures found in the XSD file (stage 2). JAXB's binding compiler (xjc) can be used for this purpose (Oracle 2011b). At this point, we use XStream (Walnes 2011) to deserialize the produced XML into the corresponding generated Java objects, creating this way a list of objects that form our final workload

(stage 3). This is a process that uses reflection (McCluskey 1998) to load classes by name and builds a list of objects that are integrated into one unit test case per each service operation (stage 4).

Most tools (like benerator) are, up to this date, unable to consider multiple domain relations for the input parameters. In fact, to generate the input values this tool only allows the definition of a single domain restriction. Although this restriction can also be a union of restrictions, inter-parameter restrictions are not taken into account, hence not usable. This way, the generated workload may include invalid web service calls that have to be identified and discarded (based on the domains specified using EDEL).

A key difficulty related to the workload generation is that the coverage of the web service calls is not easy to guarantee (e.g., it can be extremely difficult to generate a workload that exercises all the web service code). Our proposal includes executing the workload and using a test coverage analysis tool to get a metric of the code coverage, such as Cobertura (Doliner 2008) (stage 5). If the developer is not satisfied with the coverage then more web service calls are required. Calls must be added to the workload until the code coverage reaches the level the developer desires (stage 6).

As mentioned before, composite web services use external services to execute a given process. The results of the execution of these external components can be seen as inputs for the composite service and are thus a potential source of robustness problems. The last step of this phase consists of **gathering information on the response domains of any external web services used**. While executing the generated workload, all calls to external services are intercepted and logged (using AOP, as explained in the following section). The relevant information (i.e., operations, parameters, and data types) on those services responses is afterwards extracted from the corresponding WSDL descriptions. If this description already specifies the valid domains (using EDEL) then these valid domains are used. Otherwise, the developer is asked to provide the valid response domain for each component service, which is converted to EDEL and appended to a local copy of the XSD file.

5.3 Robustness tests execution

Our proposal for robustness problems identification consists of performing a set of tests on web services inputs, considering both call parameters and external web services responses. In terms of the complete procedure, besides generating and executing robustness tests it may be necessary to go back to previous phases (for instance, when a parameter's domain needs to be redefined).

The robustness tests are based on combinations of exceptional and acceptable input values that are generated and applied by our fault injection tool. Comparing to the setup presented in the previous chapter (see Figure 4.2) for client-side testing, a different setup is used (namely regarding the position of the fault injection location). This setup is illustrated in Figure 5.4.

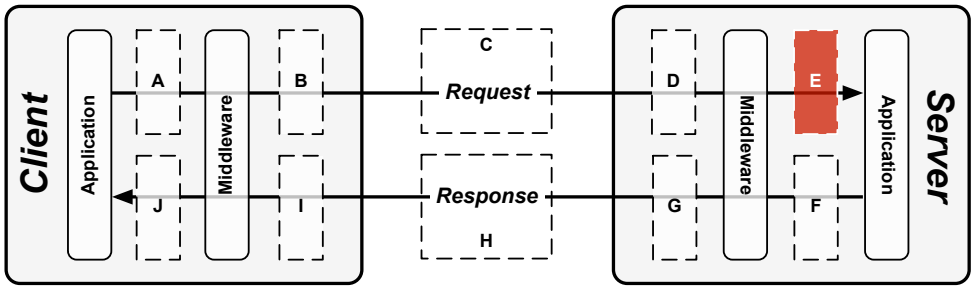


Figure 5.4 – Fault injection location used for the server-side web services robustness tests.

As discussed in Section 3.3.2 and because we now have access to the service and supporting infrastructure, we can inject faults immediately before the execution of the service code. In fact, as the goal is to test the service implementation (and not the supporting middleware) we can choose location 'E' as a fault injection location. Selecting this location also fits well with the use of Aspect Oriented Programming technology (see Section 2.3.1 and (Kiczales et al. 1997)), as AspectJ provides an easy way to intercept calls to web services.

Figure 5.5 presents a more detailed view of the fault injection process including the relation between a simple web service, a client, and the fault injection tool.

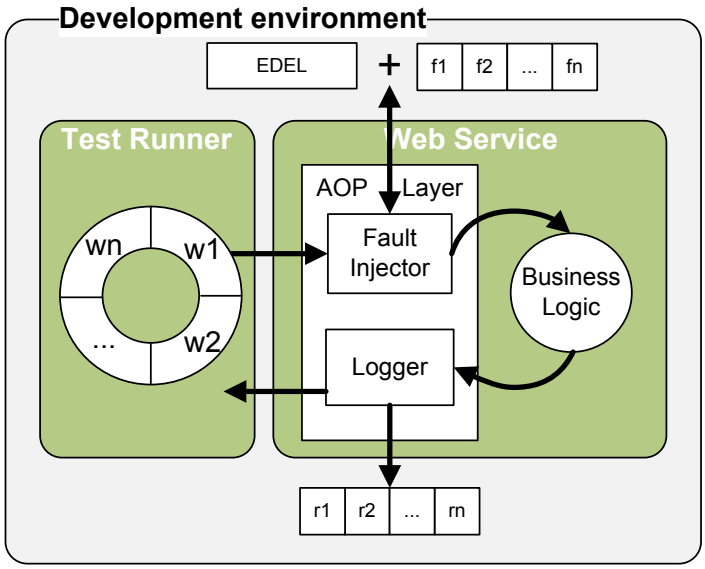


Figure 5.5 – The fault injection process.

AspectJ (Eclipse Foundation 2008) was used to create a fault injection tool (represented by the fault injector in the AOP layer in Figure 5.5) that is able to inject faults into any web service that uses the reference Java JAX-WS model (Sun Microsystems Inc. 2010). As introduced in Section 2.3.1, AOP is a programming paradigm that allows injecting crosscutting concerns into any application in a non-intrusive way (Kiczales et al. 1997). Essentially, fault injection capabilities are transparently integrated into the bytecode of the web service during the build process. Fault injection can hence be included or excluded from the process with a simple change in the build file at compile-time, or by changing of a value in a properties file at runtime, if necessary. Note that this technique extends to other programming languages and web service stacks, as the concepts used here are quite generic and are also present in other major languages.

In practice, the test runner invokes the service by cyclically using the generated workload (each workload member is w* in Figure 5.5). The fault injector looks at the EDEL definition (and any possible information about external service invocations) at the beginning of the injection campaign and generates a set of faults to be applied (*f1* to *fn* in Figure 5.5), and repeatedly applies them, as explained in the following paragraphs. The tests end when all faults have been applied, or when the developer decides to stop the process.

To **trigger robustness problems in the web service call parameters**, invalid input values are injected. This is done as described earlier in Section 3.3.4, again including several phases (pre-injection, injection, and post-injection). The rules used are essentially the same as the ones used for the public web services (see Section 4.1.3); however, due to the injection method selected (i.e., using aspect-oriented programming), some of the rules used in Table 4.II cannot be applied due to limits enforced by the programming language (e.g., setting an integer to its maximum value plus one). Although a different injection method could make use of the extended set of rules, as we are targeting implementations (and not supporting stacks) we can use only a reduced set. Note that the excluded rules would result in a invalid programming language representation, which essentially means that an object resulting from the application of one of those extra rules will be kept from being delivered to the service by the supporting middleware (refer to Section 3.3.1 for an explanation on the middleware role in service applications). The set of parameter mutation rules used in this context is described in Table 5.I.

To **trigger robustness problems in the responses of external web services** we again inject invalid input values, but we extend this concept to the injection of Exceptions, as defined earlier in Table 3.I. In this case, we explicitly throw Exceptions at particular joinpoints (invocations of external web services) with the goal of exercising any existing error handling code. Examples of the set of possible exceptions to be thrown in the injection period is defined in Table 5.II.

The exception injection rules include the injection of all exceptions declared by the external service (i.e., checked exceptions, that are a subclass of Exception) and a set of runtime exceptions. The latter set includes Java's RuntimeException (the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine) and all of its direct subclasses known to the Java environment. These exceptions are created by reflection (McCluskey 1998), which enables us to maintain the fault injection code independent of the service being tested.

Table 5.I – Parameter mutation rules.

Parameter type	Parameter mutation
String	Replace by null value
	Replace by empty string
	Replace by predefined string
	Replace by string with nonprintable characters
	Add nonprintable characters to the string
	Replace by alphanumeric string
	Set a number of characters below the minimum acceptable size
	Add characters to overflow max size
	Malicious predefined string (see examples in Table 4.III)
Number	Replace by null value
	Replace by -1
	Replace by 1
	Replace by 0
	Add one
	Subtract 1
	Replace by maximum number valid for the type
	Replace by minimum number valid for the type
	Replace by maximum number valid for the type plus one
	Replace by minimum number valid for the type minus one
	Replace by maximum value valid for the parameter
	Replace by minimum value valid for the parameter
List	Replace by null value
	Remove element from the list
	Add element to the list
	Duplicate elements of the list
	Remove all elements from the list except the first one
	Remove all elements from the list
Date	Replace by null value
	Replace by maximum date valid for the parameter
	Replace by minimum date valid for the parameter
	Replace by maximum date valid for the parameter plus one day
	Replace by minimum date valid for the parameter minus one day
	Add 100 years to the date
	Subtract 100 years to the date
	Replace by the last day of the previous millennium
	Replace by the first day of this millennium
Boolean	Replace by null value

Table 5.II – A subset of Java Exception mutation rules.

Exception superclass	Exception mutation
Exception	Any Exception declared in by the web service operation
RuntimeException	ArithmeticException
	BufferOverflowException
	BufferUnderflowException
	ClassCastException
	IndexOutOfBoundsException
	NullPointerException
	...
	WebServiceException

A detailed log is created during the execution of the robustness tests. Each entry in this log (represented by r* in Figure 5.5) corresponds to a single execution of a web service operation and includes the input values, the fault injected, and the web service response. **Identifying robustness problems** consists then in running a analyzer tool (Laranjeiro 2011), that, using the EDEL description, is able to detect responses that violate the valid output domain of the operation, including unexpected exceptions (i.e., any checked exception not declared at the service's interface or any runtime exception). The output is a set of invalid responses (if any) and all useful information to debug the problem (injected fault type, input value, parameter name and type, etc.), which can be used later to fix the detected robustness issues (see Chapter 6).

5.4 Web service characterization

As discussed in Chapter 3, a service that undergoes robustness testing can be characterized according to the following three dimensions:

1) Severity of the observed failures;
2) Detailed service behavior;
3) Compliance to a service specification.

When executing tests at the server-side, more information can be collected and there is the possibility of distinguishing failures in a better way. It is

now possible to use a more detailed failure mode scale as there is access to the service and its supporting infrastructure (there is also knowledge about the code and runtime environment). In this sense, the adaptation of the CRASH scale (P. Koopman and DeVale 1999) presented in Section 3.4, is more adequate than the CCE scale (Correct-Crash-Error) presented earlier in Section 4.1.4 for the characterization of public services. In these circumstances, it is possible to distinguish between the 5 failure modes in the CRASH scale (e.g., Catastrophic and Restart) and this constitutes in fact our proposal for the classification of the **severity of the observed failures** at the server side.

The characterization procedure requires also a **detailed description of the service behavior**, (not only in terms of severity). Based on our practical knowledge and experimental evaluation presented in Chapter 7, we found the instantiation of the service tags proposal for the public web services (see Section 4.1.4), also adequate for the server-side robustness tests.

Finally, when robustness tests are executed at the server-side, it is possible to understand to what degree a given service is compliant to a specification. As referred in Section 3.4, a **compliance or conformity** problem is observed when an acceptable input (as defined in a specification) results in a robustness problem (e.g., an out-of-domain or unexpected service response). The proposed three-degree severity scale (Severe, Medium, Light) can then be directly used to characterize the service behavior in this perspective.

5.5 Conclusion

This chapter discussed techniques for server-side robustness testing of web services. Access to code or service infrastructures enables the execution of a more detailed testing procedure. In particular, this chapter introduced a new domain expression language (EDEL) that allows developers to express domain dependencies between parameters and announce domains in a more complete way, allowing the creation of robustness tests in a more complete way.

The server-side execution of tests enables collecting useful information regarding aspects like workload coverage and the detection of external services invocations. We extend our technique to take advantage of such

aspects by proposing a workload generation method that can benefit from being informed about the code coverage. This is an indicator of the workload quality and can be used to stop, continue, or adjust the generation process until a satisfactory coverage level is obtained. Our robustness testing technique also detects if external services are being used, and can perform this detection without source code access. This is a useful feature, particularly in legacy services, or in services where it is not possible to obtain full code access. Considering that the responses of external services are also an input to a web service, we extend the robustness testing technique to include the injection of exceptions at these special points. In fact, a service can possess multiple entry points, in addition to the main (client-triggered) entry point.

The server-side robustness testing approach presented here sets the basis for a technique that can improve the robustness of web services (described in the following chapter). Both techniques (assessment and improvement) are interconnected by components originally designed for robustness assessment (namely, the domain definition technique using EDEL and the robustness tests), but that are actually the core of the approach for improving the robustness of web services.

Chapter 6
Mitigating Robustness Problems in Web Services

This chapter proposes a set of techniques for robustness improvement. The domain expression language proposed in the previous chapter is used as the support of a wrapper-based technique for robustness improvement. We take this improvement technique one step further and propose an approach focusing on securing services from attacks based on malicious inputs (although possibly domain valid), namely SQL/XPath Injection attacks. Furthermore, we also describe how to integrate the robustness testing technique in the software development process, as a way to prevent the deployment of web services with robustness problems. Thus, this chapter takes advantage of the robustness testing knowledge (available when tests are executed at the server-side), to focus on improvement, either to support automated tools and techniques, or by adapting a modern agile development process to include robustness testing.

This chapter is organized as follows. Section 6.1 describes a wrapping technique to improve the behavior of services in the presence of invalid

inputs. Section 6.2 proposes an approach for protecting services against malicious inputs. Section 6.3 discusses the integration of robustness testing in the Test-Driven Development methodology, as a means to develop more robust services. Finally, Section 6.4 concludes the chapter.

6.1 Automatic web services interface wrapping

The lack of proper input validation can result in a service that contains robustness problems. Wrappers have been used successfully in the past to provide increased robustness and security in many domains, ranging from C applications to Commercial off-the-shelf Components (COTS) (Fetzer and Zhen Xiao 2003; Fetzer and Zhen Xiao 2003; Susskraut and Fetzer 2007; Popov et al. 2001; Ghosh, Schmid, and Hill 1995). Our proposal for the automatic removal of robustness problems builds on our technique for assessing the robustness of services. In particular, we use the Extended Domain Expression Language (EDEL) proposed in Section 5.1, as a starting point to provide protection against invalid inputs. EDEL provides domain expression capabilities that can be used to represent existing dependencies among the service parameters. These capabilities are the support for automatically generating a wrapper that, at runtime, stops any incoming operation requests that fall out of the valid parameter domains.

There are multiple implementation techniques that can be used to deploy a wrapper around a given service. An easy option is to apply bytecode instrumentation (using for instance Aspect Oriented Programming – AOP (Kiczales et al. 1997)) to wrap validation logic over the target service (see Section 2.3.1 for an introduction to AOP). Thus, there is no need to directly modify the application source code, as the necessary modifications can be weaved in by the AOP compiler at compile-time.

AOP provides us with a way to inject a crosscutting validation concern into the multiple operations of a service. In this case, the goal is to obtain a way to intercept any starting executions of web service operations and precede each execution of the service business logic with input validation bytecode. An illustration of this runtime process is shown in Figure 6.1.

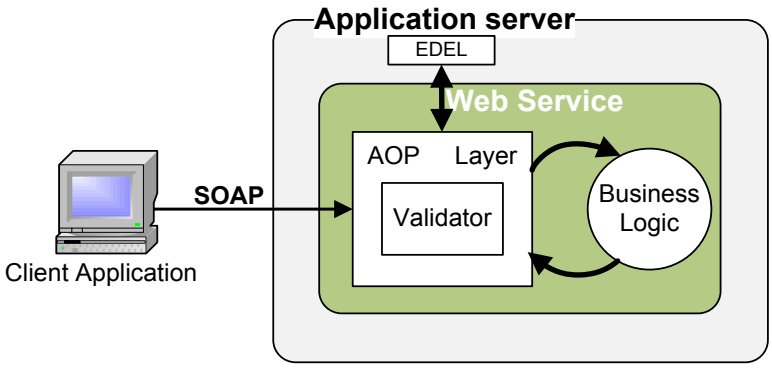


Figure 6.1 – The input validation process at runtime.

As shown, each arriving operation request undergoes a complete validation process, at runtime, before it is delivered for business logic processing. All incoming requests that are not valid, according to the domains defined in EDEL, are aborted and a custom exception is thrown. This custom exception exists both in checked and unchecked versions – *ValidationException* and *ValidationRuntimeException*, respectively. A developer creating a new service can leave the validation concerns to our framework, but can also declare that the service being developed can throw a *ValidationException*. This informs each client that the invocation of a given service may result in a well-known exceptional behavior. If the client-side programming language supports checked exceptions, the client itself will be forced to explicitly handle the exception. If there is no support for this kind of exceptions at the client-side, at least there is formal information about the possible behaviors resulting from the invocation of a web service operation. Having more information at the client side for sure helps creating a client that complies with a given specification.

An important aspect is that this validation technique applies not only to the starting executions of the local service operations, but also to any existing external service invocations. In particular, it provides protection against invalid responses (i.e., out of domain) resulting from the invocation of external services. From the point of view of the enhanced service these responses are nothing more than input data. In other words, besides instrumenting the main service operations, we also instrument any existing external service calls to perform domain validation over the received responses. Additionally, any exception thrown by the external

service is caught and analyzed. If it is one of the exceptions declared by the external service (i.e., it is an expected exception), then it is re-thrown. For all other cases, it is wrapped using the previously mentioned custom exceptions.

In some cases it may be necessary to fully protect a service. However, for other services (e.g., legacy services that already perform partial input validation), it may be unnecessary to double validate parameters. In this case, we assume that there is some source of information regarding which parameters should be protected (for instance, a developer may have robustness tests results that identify which parameters must be protected). With this information we perform a domain reduction, where all domain information that exclusively refers to non-problematic parameters is removed.

The robustness protection process is apparently completely effective but we must take into account the fact that the expression of the domains is still a human task, therefore error-prone. In fact, after applying the protective wrapper, the developer should verify if some deviation from the original service's business logic was accidentally introduced by our mechanism (e.g., due to a bug in the domain definition). So, ideally, our wrapping technique should be integrated with the robustness testing procedure, in such a way that makes it possible to verify the final correctness of the service (in terms of robustness). A possible integration of the wrapping technique and service behavior verification (robustness improvement) in robustness testing is shown in Figure 6.2.

Notice that phases 1 to 4 have already been described in the previous chapter. The integration is focused in phase 5, which consists essentially of wrapping and verification. After generating the protective wrapper, the next step is to repeat the robustness tests to check if the robustness problems identified before were fixed and if new problems appeared (fixing a problem may disclose another). This way, steps 3 to 5 should be repeated if new robustness problems are identified.

To confirm that the added functionality did not modify the behavior of the service and only corrected robustness problems, we need to execute an extra step (step 5.3 in Figure 6.2), which consists of re-executing the workload and checking the results looking for responses that are outside the expected domains or for responses that differ from the original workload execution step. In some cases, for a particular input, web service operations always provide the same response. In such cases, a workload

re-execution can be directly compared to a previous execution. If differences exist, then the wrapping process must have added some bug to the service (the developer should review the web service definitions (XSD and EDEL)). For web services that do not always provide the same response (i.e., are not deterministic) when in presence of a particular input, the tester can simply check for responses outside the expected domains. Again, if problems are identified, the tester should review the web service definitions (XSD and EDEL).

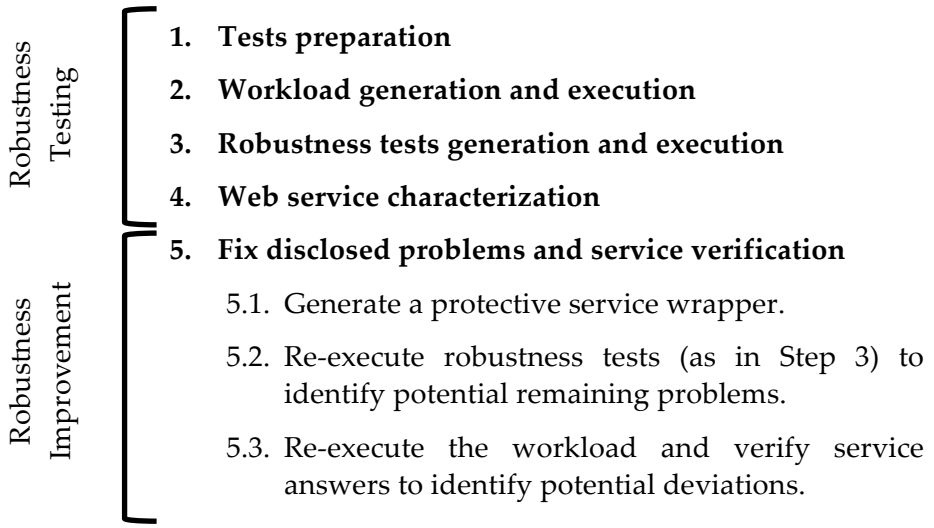


Figure 6.2 – Integration of the robustness improvement steps in the robustness testing technique.

At the end of the procedure, the developer is in possession of a highly protected service (in terms of the response to invalid domain parameters). Despite this, there are parameters that may fit in the valid domain and still be problematic to the system. This is, for example, the case of inputs that exploit security vulnerabilities present in the code. Such parameters should not be announced as invalid since they can be in fact valid from a domain point-of-view; however, there is a need to further protect services against the execution of these potentially hazardous requests. This is the goal of the technique presented in the next section.

6.2 Securing web services against command injection attacks

The wide use and exposure of web services results in any existing security vulnerability being most probably uncovered and exploited by hackers. As previously referred, command injection attacks (e.g., SQL or XPath injection) are particularly dangerous as they take advantage of improperly coded applications to change queries sent to a database, enabling, for instance, access to critical data (Stuttard and Pinto 2007).

Vulnerabilities allowing SQL Injection and XPath injection attacks are exceptionally relevant in web services (Vieira, Antunes, and Madeira 2009), as services exposure is high and they frequently use a data persistence solution (Transaction Processing Performance Council 2008) based either in a traditional relational database or in a XML database. Currently major database vendors and several open-source efforts provide XML databases (e.g., Oracle XML DB, SQL Server 2008, Apache Xindice, etc.) and, typically, the access to this type of databases uses XPath expressions. While the goal of XPath Injection is to maliciously explore any existing vulnerabilities in XPath expressions used by an application (for instance to access an XML database), SQL Injection tries to change the SQL statements in a similar manner (Stuttard and Pinto 2007).

To perform SQL Injection the attacker exploits an unchecked input in order to modify the structure of a SQL command (W. Halfond, Viegas, and Orso 2006). Usually, the attacker starts by trying to add an extra condition in the '*where*' clause of a SQL command to gain some form of privileged access. Then the attacker provides inputs that lead to the executing of a SQL command that returns valuable information (typically using a '*union*' clause with the malicious select), disrupting the database by performing inserts, deletes or updates. The same happens for XPath Injection (only the syntax differs).

In this section we propose an approach that is able to handle malicious inputs that can be part of the valid domain of a given service, but can however represent a threat to the service when security vulnerabilities are present. Our proposal to protect services from potential SQL and XPath injection attacks is based on detecting anomalies, which consists of searching for deviations from an historical (learned) profile of good commands (Valeur, Mutz, and Vigna 2005), and includes two major phases:

1. **Statement learning:** consists of using a real or generated workload (i.e., a set of calls) to exercise the web service. The service is instrumented to learn valid SQL statements and XPath expressions executed during the execution of that workload.
2. **Service protection:** consists of instrumenting the service to provide protection against SQL/XPath Injection attacks by performing the following operations:
   2.1. Matching incoming requests with the valid set of requests gathered in the learning phase;
   2.2. Applying a set of heuristics when unlearned data access statements appear (i.e., no match can be made with previously learned statements).

As we will see in the experimental evaluation presented in Section 7.4.2, the proposed approach is quite effective, has an extremely low overhead, and does not require any access to source code (the technique uses bytecode instrumentation). This work focuses on source code vulnerabilities and not on any specific security mechanisms, such as authentication or data encryption. Moreover, regardless of the fact that the mechanism presented is designed for SQL and XPath Injection vulnerabilities, the technique can be used with other kinds of injection vulnerabilities (e.g., CRLF Injection, Reflection Injection, among others (The Open Web Application Security Project (OWASP) 2011)). The requirements are that a monitoring point exists and can be set and that it is possible to learn the regular profile of requests.

6.2.1 SQL/XPath learning

The first step of the learning technique is to exercise the web service by using a workload and identify all locations in the service implementation where SQL and XPath commands are executed. An existing workload can be used, if available. If not, we can fallback to the workload generation process presented earlier in Section 5.2 and obtain, in this way, a set of distinct service calls that can be used to collect the required information (i.e., a set of genuine, non-malicious, data access calls). In order to collect this information we instrument the service and wrap all data access

operations in our custom code. Notice that this same interception technique is used to protect the service in a later phase that, however, executes a distinct wrapping code.

As before, we chose to use Aspect Oriented Programming to instrument the service as AOP provides us with an easy way to transparently intercept all calls to, in this case, data access operations. In order to be able to use this technique in the current context, the developer must define (before the instrumentation can take place) a set of method signatures that correspond to APIs for executing data access operations (i.e., the execution of SQL commands and the evaluation of XPath expressions). No source code access is required during this phase. Examples of such methods are Java's JDBC API, the Spring Framework JDBC API for SQL, and Java's JAXP API or Jaxen for XPath. Although these well-known signatures are already provided by default in our mechanism, the set of APIs is easily extensible; the only requirement is to know the full signature of the method to be intercepted, which is typically part of the developer knowledge and is available in the language or persistence API documentation. After defining these signatures, the instrumentation can take place and all data access statements will be intercepted at runtime. Figure 6.3 represents the basic architecture of the interception mechanism. The learning module is described in the following paragraphs, whereas the protection module is described in Section 6.2.2.

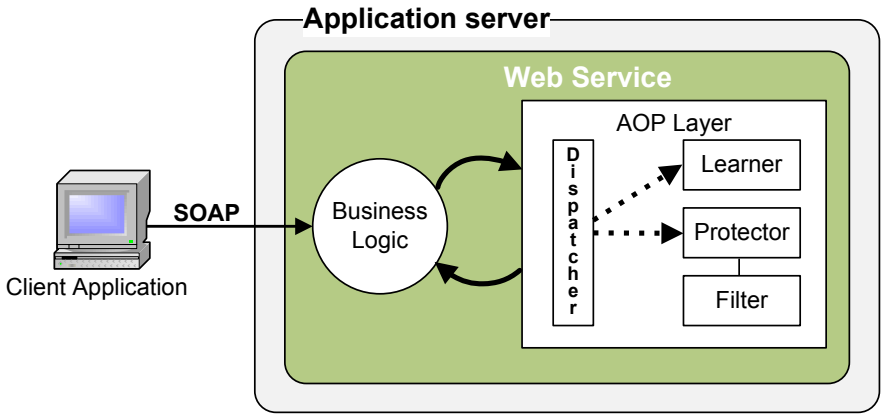


Figure 6.3 – Configuration for data access statements learning and service protection.

At runtime, each data access call is intercepted and delivered to a dispatcher. The decision here is simply to check if the application is in learning mode or in protection mode; in each case the request is delivered to an appropriate module (learner or protector module, respectively). During the learning phase, SQL and XPath commands are parsed in order to remove the data variant part (if any). In other words, the information used does not represent the exact command text, since commands may differ slightly in different executions, while keeping the same structure. For example, in the SQL command "*SELECT * from EMP where job like 'CLERK' and SAL > 1000*", the job and salary in the select criteria (job like ? and sal > ?) depend on the user's choices. This way, instead of considering the full command text, we just represent the invariant part of it by identifying delimited strings and numbers (like 'CLERK' and 1000) and replacing them with constant strings according to the data type. For example, the command above is transformed as follows: "*SELECT * from EMP where job like FOUND_A_STRING and SAL > FOUND_A_NUMBER*".

Each invariant command is then associated with a source code entry point (provided by the AOP framework) in a Map (key-value) structure. This does not mean that we need the original application's source code, but it rather means that we need bytecode compiled with source code line information. This is generally the case, even in production applications as it provides extra information on failure events (i.e., debugging information). In the Map structure, each key corresponds to a given source code point and has a set of associated valid/expected invariant commands (i.e., at a given point there might be more than one valid command, depending on the application). For example, as shown in Figure 6.4, the SQL command submitted to the database might be an insert or an update. This is why we need a list of valid SQL or XPath commands for each source code point.

When a generated workload is being used, the developer must assure that the workload is generated in such way that a minimum level of code coverage is guaranteed. As explained earlier in Section 5.2, our workload generation technique already considers this issue and uses a coverage analysis tool to measure the workload coverage (we used Cobertura (Doliner 2008), in particular). This allows getting information about the coverage space of the workload before deploying the system. Although this does not assure a complete learning of SQL commands and XPath expressions, it allows the developer to have a higher degree of confidence. Obviously, increasing the size of the workload, or choosing a more

representative one (e.g., a real or realistic workload), is a way of improving coverage and further guaranteeing a more complete learning.

```
...
if (isInsert())
{
  sql = "INSERT INTO CLIENT VALUES (seq.nextval, 'Jack')";
}
else
{
  sql = "UPDATE CLIENT SET NAME='John' WHERE ID=1";
}

statement.execute(sql);
...
```

Figure 6.4 – Example of SQL commands execution.

6.2.2 Service protection

Service protection at runtime (i.e., after deployment) consists of performing one security check per data access. Execution is allowed to proceed when that check concludes that the statement is secure, thus presenting no harm to the application or supporting infrastructure. For all other cases, the default behavior is to abort the execution, signaling an exception. Nonetheless, this behavior can be overridden by configuration and an additional check (based on heuristics) can then be executed by a Filter component. These behaviors are explained in the following paragraphs.

During the protection phase, all SQL and XPath commands are intercepted and parsed into invariant codes. The request flow is very similar to the learning phase; the difference is that each request is now delivered to the protector module (see Figure 6.3). Obviously, the calculated codes are not added to the learned command set. Instead, they are compared to the learned and valid invariant commands for the code point at which the command was executed.

This matching process consists of looking up the current source code origin in the previously referred Map structure and getting the list of

codes of the valid (learned) commands for that point. This list (generally small) is then searched for an element that exactly matches the invariant command that is being executed. Execution is allowed to proceed if a match is found. Otherwise, a security exception (the unqualified name for this exception is *SecurityRuntimeException*) is thrown and, in this way, code execution is kept from proceeding, preventing the potential attack. If the source code origin is not found in the Map lookup, execution is also kept from proceeding in a similar manner (in this case, a different exception is thrown – *CodePointNotTrainedRuntimeException*). This case strongly indicates that the learning phase is incomplete (coverage was not good enough) and that an extended workload is probably required. We also provide checked versions of these exceptions, allowing the developer to explicitly state that a web service may throw a particular exception.

As the developer may have no way of verifying the completeness of the training mode (e.g., when there is no access to source code), it may happen that, at runtime, some genuine (i.e., valid) statements are marked for abortion. In this case, and depending on the acceptable rate of false-positives (i.e., valid commands marked as invalid), a system administrator has the following options:

1) Switch back to training mode (when in a secure environment);
2) Verify the tool's logs and add any false positive to the list of valid commands;
3) Configure the heuristics-based filter for automatic false-positive handling.

The heuristics filter uses an attack dictionary to decide if a new (i.e., not yet learned) SQL/XPath statement is potentially malicious (or not). In the positive cases, an exception is thrown (*FilteringRuntimeException*) stopping any possible damage ahead of time. An excerpt of this dictionary, which can be found packaged with our security tool at (Laranjeiro 2011), is presented in Figure 6.5.

The dictionary used by the filter is essentially a set of regular expressions and was built based on the following sources:

- Current studies on security (Anley 2002; Mookhey and Burghate);
- A compilation of attacks generated by well known top commercial vulnerability scanners (Acunetix Web Vulnerability Scanner, HP

WebInspect, and IBM Rational AppScan (Acunetix 2011; Hewlett Packard 2011; IBM 2011));

- A set of malicious request patterns built by a security team that was challenged to attack a typical web services scenario (see Section 7.4.2 for a more detailed description of this scenario, team, and experiments).

```
# typical SQL Injection attack
/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix

# SQL Injection with the UNION keyword
/((\%27)|(\'))union/ix

# SQL Injection attack targeting Microsoft SQL Server
/exec(\s|\+)+(s|x)p\w+/ix
```

Figure 6.5 –Expressions for well-known SQL Injection attacks

The goal of using multiple sources as basis for the construction of the filter was to include as much diversity as possible in the definition of the regular expressions. With a richer filter it is possible to identify and stop more malicious requests before any damage is done to the actual service.

In addition to the set of expressions used by the filter it is also important to set the conditions that define when and how the filter is going to be used. Considering that an unknown SQL/XPath command can appear either at trained or untrained code points, we have designed the following configurable strategies for the filter and protection mechanism:

- **Level 1:** no unlearned SQL/XPath statements are allowed to execute. At runtime, if a data access statement converts into a previously unseen code, then execution is immediately blocked without further analysis (i.e., not delivered to the filter).
- **Level 2:** SQL/XPath statements originating from new (previously unseen) code points are allowed to execute. New SQL/XPath statements originating from old (previously seen) code points are delivered to the filter for further analysis. At the filter, if the data access statement matches with one of the known attack patterns, execution is blocked. If not, execution is allowed to proceed.

- **Level 3:** new SQL/XPath statements originating from old (previously seen) code points are allowed to execute. SQL/XPath statements originating from new (previously unseen) code points are delivered to the filter for further analysis.
- **Level 4:** all unlearned SQL/XPath statements are delivered to the filter for further analysis.
- **Level 5:** all unlearned SQL/XPath statements are accepted for execution. This is essentially an easy way to disable the protection at runtime. It can be used if a severe unexpected case appears (e.g., a serious learning problem is discovered).

Table 6.I summarizes the proposed five configurable strategies. Obviously, no strategy is able to fit all scenarios, and it is up to the provider to select the strategy that best fits his environment. As mentioned, filter strategies can be changed at runtime, thus, if the environment changes (e.g., when the service is delivered to a greater number of clients or to a potentially malicious set of clients), the provider can adapt its strategy according to the specificities of that environment.

Table 6.I. Configurable protective strategies

Level	Code point	Action
1	New/Old	Block
2	New	Allow
	Old	Filter
3	New	Filter
	Old	Allow
4	New/Old	Filter
5	New/Old	Allow

Before being deployed in a production environment, the protected web service should be tested in terms of security by using, for instance, vulnerability scanners or manually crafted malicious requests. The goal is to verify if the security mechanism is working properly and to check if it is able to stop all XPath/SQL injection attempts by raising the appropriate security exception. If vulnerabilities are detected it means that the workload coverage was not good enough and that the learning phase is

incomplete. In this case, the workload should be extended and the learning process repeated.

Finally, the developer may also want to re-execute the original workload (the one used for training) to verify that the service behavior remains correct. Problem indicators include responses outside the expected domains. For deterministic services, responses that are different from those obtained during the first workload execution are also problem indicators. These might indicate potential problems introduced by the security mechanism (e.g., due to an incomplete learning of SQL and XPath commands). The process should obviously be canceled if these problems are identified, and their source should be investigated. If this source is related to the learning phase, the developer should extend the workload in order to improve its completeness.

6.3 Extending test-driven development for robust web services

Test-Driven Development (TDD) (K. Beck 2003) is a software development process particularly suitable for web services, as these are based in well-defined interfaces that are quite appropriate for producing test cases. In TDD, the tests specify the requirements and contain assertions that can be true or false. Running the tests allows developers to quickly validate the expected behavior as code development evolves. A large number of unit testing frameworks are available for developers to create and automatically run sets of test cases (e.g., JUnit or NUnit (Kent Beck 2011; NUnit.org 2011)). However, in general, defining test cases that guarantee high coverage is quite difficult (Whittaker 2000) and developers tend to focus on positive test cases (i.e., tests that do not try to break the feature, but simply to demonstrate that it works on normal situations) and often disregard negative test cases, such as the ones targeting robustness validation (P. Koopman and DeVale 1999). This way, and considering that web services are usually complex software components developed against typically aggressive schedule constraints, they are frequently deployed without being properly tested and carrying residual software defects (as demonstrated in our experimental evaluation in Chapter 7). This may expose applications to severe problems, including critical vulnerabilities that can be exploited with serious consequences, such as denial-of-service (DoS), or data loss.

To allow developing more robust web services from the beginning, we propose an approach that extends the Test-Driven Development technique to include robustness testing. The approach consists of integrating robustness tests into the process allowing developers to execute the test cases that are necessary to also validate the web services robustness. As described in Chapter 2, Test-Driven Development is an iterative development process that, in summary, consists of the following steps (K. Beck 2003):

- **Step #1 – Add a test**: the process starts by writing or modifying a test (or set of tests) that validates the feature to be developed or modified.
- **Step #2 – Run all tests and verify if the new ones fail**: this step validates if new tests effectively fail (and do not incorrectly pass without requiring any new code). It can also show that the testing infrastructure is working correctly.
- **Step #3 – Write code**: the developer writes the code required for the tests to pass. This code may not be final and may pass the tests holding poor coding quality. The development process considers this hypothesis and there is space for improving the code in later steps. At this step, the written code should not include functionality that is not contemplated by the existent tests.
- **Step #4 – Run the automated tests**: if all test cases pass then the code meets all the tested requirements. When some tests fail the development process goes back to step 3. If the developer notes that the tests do not reflect the desired functionality the process falls back to step 1.
- **Step #5 – Refactor code**: the last step, once the code passes all tests, consists in code improvement (e.g., more readability, less duplication). Tests should be re-executed after this step to ensure the functionality is not altered.

A new test (or set of tests) sets the starting point of a new iteration. The total number of iterations depends on numerous factors, such as the size of the software being developed, the developer's experience, specific development tools being used, etc. In practice, to integrate robustness testing in the test-driven development process we need an additional step related to the web service interface specification:

- **Step #0 – Specifying the web service**: this step must be performed before anything else and consists of specifying the web service interface (i.e., operations, parameters types, input and output domains, parameter dependencies). This information should defined using EDEL and will be used later to generate robustness tests and detect problems.

Robustness tests are automatically generated during Step #1 using the definitions from Step #0 and robustness tests execution is performed during Steps #2 and #4 of TDD. The execution of both steps follows the regular execution profile presented in Chapter 5 and a pre-injection, injection, and post-injection period may apply as well as all configuration aspects associated with the robustness testing procedure (described earlier in Chapter 3).

6.4 Conclusion

This chapter proposed an approach to create automatic domain validation wrappers, which does not require additional development effort (other than defining the domains themselves). The technique is based on the use of EDEL to define the operations domains. After these domains are defined, validation wrappers are automatically created, protecting services from invalid inputs at runtime.

An additional step was taken to support the deployment of not only services that fail in the presence of invalid inputs, but also services that are vulnerable to malicious inputs that exploit top vulnerabilities in the web services domain. Since malicious inputs can take numerous forms, the approach uses a learning phase to capture the pattern of genuine requests. This phase serves as a basis to later decide if a given request is genuine or malicious.

The chapter ended with a proposal to adapt the Test-Driven Development technique to include robustness testing, a possible way to prevent the development of services with robustness issues. Tests are created and executed before the actual service functionality is written and are used to drive the development as well, as opposed to testing services only after development. Obviously, this technique can also extend to other software development processes that require a testing phase.

In summary, we presented techniques for improving the robustness of services in three distinct points of view. Despite being described separately, they can be used jointly. In fact, improving the robustness of services requires executing robustness tests and protecting services against malicious inputs. The techniques can be used, in a more formal way, integrated in a software development methodology; their joint use contributes to the deployment of robust software services.

Chapter 7
Case Studies on Robustness Testing and Improvement

This chapter illustrates the practical application and evaluation of the techniques presented in previous chapters. An experimental evaluation was conducted, exploring distinct major robustness testing and improving perspectives. These include client-side robustness testing; server-side robustness testing; and robustness and security improvement.

The robustness testing approach is applied in real scenarios, namely in a large sample of public web services implementations, popular JMS middleware implementations, and a set of TPC-App and open-source web services. All tested services are available at (Laranjeiro 2011) (with the exception of the public web services that are listed in Annex B). This variety of scenarios illustrates the adaptability of the approach to real service applications and results strongly indicate its utility not only for developers but also for service consumers and providers. The improvement approaches are conducted in server-side realistic development environments. The experiments conducted clearly show that the proposed techniques are quite effective not only in protecting services

from invalid inputs conditions but also from malicious values, while requiring a low developer effort.

The experimental evaluation followed the scientific method, which can be expressed as the test of an hypothesis by performing controlled experiments. According to the scientific method, the hypothesis must be testable and falsifiable (it can also produce a negative result); the experiments must be controlled by testing only one variable at a time, and must be reproducible so that the results are also repeatable (from a statistical perspective they lead to the same conclusions) (Peisert and Bishop 2007a; Peisert and Bishop 2007b).

All datasets have obviously their own specific characteristics and therefore cannot be easily generalized to a broad range of situations. Anyway, all results are presented, stating clearly how the experiments were conducted and their limitations. An effort was made to draw conclusions only within the scope of the experiments, avoiding "hard to prove" generalizations. Despite this, we believe that the set of scenarios selected (and experiments conducted) focus on very popular service technologies and serve the purpose of illustrating the applicability of our proposal.

This chapter is organized as follows. Section 7.1 describes the setup used for all experiments. Section 7.2 presents the experimental evaluation carried out to illustrate the application of the robustness testing procedure at client-side. Section 7.3 presents the server-side approach for testing the robustness of services and Section 7.4 analyses the results of the approach for improving services robustness, including protection against malicious inputs. Section 7.5 concludes this chapter.

7.1 Experimental scenarios

The following experimental scenarios, detailed in the next paragraphs, are used to illustrate the robustness testing and improvement concepts proposed earlier:

1) **Client-side robustness testing: a)** a large set of public web services implementations is used for robustness testing; **b)** distinct JMS middleware platforms are evaluated for robustness;

2) **Server-side robustness testing:** a set of different web services implementations (three versions of TPC-App services and open-source services) is used for server-side robustness testing;
3) **Robustness improvement:** the set of web services implementations referred in the previous point is used to illustrate our robustness improvement procedure;
4) **Security improvement:** the security improvement approach is used along with a set of open-source services and one implementation of the TPC-App services.

In **experimental scenario 1.a)**, using the client-side robustness testing approach proposed in Section 4.1, and supported by the *wsrbench* tool (see Annex A), we evaluated the robustness of **250 public web services**, comprising 1204 operations and 4085 parameters, deployed over 42 different country domains, and provided by 150 different parties. These parties include several well known companies like Microsoft, Volvo, Nissan, and Amazon; multiple governmental services; banking services; payment gateways; software development companies; internet providers; cable television and telephone providers, among many others. The complete list of tested services includes web services deployed on 17 distinct server platforms and 7 different web service stacks. A complete and detailed list can be found in Annex B.

In **experimental scenario 1.b)** we selected **three JMS middleware implementations** to demonstrate the testing approach proposed in Section 4.2. This kind of middleware components are usually used in conjunction with application servers, as they can provide other functions typically needed for enterprise applications (e.g., data persistence, or presentation layer support). As JBoss AS (Red Hat Middleware 2008b) is one of the most widely used application servers on the market, we decided to test JBossMQ (Red Hat Middleware 2008a) for robustness problems (including two different major versions – 4.2.1.GA and 3.2.8.SP1). We have also tested the Apache Software Foundation project ActiveMQ 4.1.1 (Apache Software Foundation 2008b) as it has a large popularity among the open source community.

Most of the JMS implementations are application server independent (i.e., a given JMS implementation can be used in any application server). However, in real scenarios the JMS provider used is typically the one that is built into the application server being employed (otherwise a serious

configuration effort is normally involved). In this way, we tested each JMS implementation in its most used container, which is JBoss AS for JBoss MQ (versions 4.2.1.GA and 3.2.8.SP1) and Apache Geronimo (version 2.0.2) (Apache Software Foundation 2008c) for ActiveMQ. A custom built tool, available at (Laranjeiro 2011), was used for testing the JMS middleware implementations.

In **experimental scenario 2)** we used different implementations of the services specified by the **TPC-App benchmark** (Transaction Processing Performance Council 2008) and a set of **4 open-source web services** adapted from code publicly available on the Internet (Exhedra Solutions, Inc. 2010) – consisting of a total of 32 service operations. TPC-App is a performance benchmark for web services and application servers widely accepted as representative of real environments. The open-source services used perform the following functions: manage student information; manage phone book addresses; and simulate bank operations (in 2 versions).

To verify if our robustness testing approach is applicable in multiple scenarios, we created 3 different versions (versions A, B, and C) of the services specified by the TPC-App benchmark (Change Payment Method, New Customer, New Product, and Product Detail). Distinct developers (with more than 2 years of experience in Java development) implemented these versions using N-Version programming (A. Avizienis 1995).

Table 7.I presents an overview of the size and average cyclomatic complexity (McConnell 2004) of the tested services, as reported by SourceMonitor (Campwood Software 2011). We can observe that the extension of these services varies between 71 and 527 lines of source code (an approximate total of 3093 lines), while the average cyclomatic complexity varies between 1.32 and 9.0.

Table 7.I – Services characterization.

Web Service		Lines of Code	Average Cyclomatic Complexity
TPC-App A	ChangePayment	108	5.0
	NewCustomer	229	5.60
	NewProducts	114	4.5
	ProductDetail	152	5.5
TPC-App B	ChangePayment	158	1.67
	NewCustomer	219	2.15
	NewProducts	71	1.75
	ProductDetail	68	1.32
TPC-App C	ChangePayment	269	1.94
	NewCustomer	257	2.0
	NewProducts	115	1.57
	ProductDetail	100	2.0
Public	JamesSmith	353	6.2
	PhoneDir	130	2.0
	Bank	223	3.4
	Bank3	527	9.0

The robustness testing scenarios **1.a), 1.b), and 2)** used the test configurations presented in Table 7.II (see Chapter 3 for an explanation about each configuration parameter).

Table 7.II – Configuration of the robustness testing experiments.

Tested Scenario	Random workload size	Phases duration (A, B, C)	Fault replication	State restoration	Reverse-replaying tests	Order of magnitude of the executed tests
1.a) Public services	1	1	No	Not possible	No	Half-million
1.b) JMS Middleware	1	1	Yes	Yes (JMS MessageID parameter)	No	Two hundred
2) TPC-App and open-source services	5	1	No	No	No	Sixty thousand

Experimental scenario 3) demonstrates the wrapping technique presented in Section 6.1 and uses the same set of services previously described for scenario 2)**.** To illustrate the security improvement approach proposed in Section 6.2, **Experimental scenario 4)** includes the **4 open-source services** referred in the previous paragraph and **version A of the TPC-App services**. As we will see in the next sections, the robustness tests strongly indicated the potential presence of security vulnerabilities in TPC-App Version A, making it an interesting candidate to be used with our security improvement mechanism (as it most likely contained vulnerabilities).

All server-side experiments using web services were supported by JBoss 4.2.1.GA and the reference implementation for the Java API for XML Web Services (JAX-WS 2) due to their relevance in industry (Red Hat Middleware 2008b; Sun Microsystems Inc. 2010). Oracle 10g was used to support the persistence requirements of the services. The experimental setup consisted of two main nodes (client and server) deployed on two machines on an isolated Fast Ethernet network.

7.2 Client-side robustness testing results

In this section we present the experimental evaluation carried out to demonstrate the effectiveness of the client-side robustness testing approach. In particular, we demonstrate the use of the approach in the two classes of services mentioned earlier: web services and JMS middleware. The experiments presented in this section try to give answer to the following questions:

- Can developers or testers use robustness testing to effectively test their services?
- Can robustness testing be itself an initial step for improving the robustness and security of web and messaging services?
- Can robustness testing be used to compare different implementations of a given service?
- Are services, in general, being deployed or made available with robustness or security issues?

The experiments executed delivered strong answers to the above questions. As we will see, even with tests generated at the client-side, the

results presented ahead show the importance of robustness testing in a clear way.

7.2.1 Public web services

Two hundred and fifty publicly available web services, including more than 1200 operations, were tested for robustness. These services were obtained using Seekda (http://webservices.seekda.com/) - a web service search engine. This is, to the best of our knowledge, the largest web service search engine currently available on the Internet. The service selection process consisted in introducing technology-related keywords (e.g., web, service, xml, etc.) in the search engine and randomly selecting some services from the search results. The services were then tested using *wsrbench* according to the previously described procedure. Each test result (a total of 420375 responses) was double-checked by two distinct researchers with more than two years of experience in developing and testing web services and web applications, to confirm the failure modes observed. The failure mode scale used in these experiments was CCE (Correct-Crash-Error), as proposed in Section 4.1.4.

The results obtained in this first experimental scenario indicate that a large number of services are currently being deployed and made available to the general public with robustness problems. In fact, 49% of the tested services presented some kind of robustness issue. This is a very large percentage of problematic services and the problem gains a larger dimension if we consider that a large part of these problems also represent security issues. Figure 7.1 presents the **global results for our public services evaluation in three different granularity perspectives**. The service execution granularity presents the analysis from the service perspective, being the service the unit of analysis. Similarly, the operation and parameter granularities consider these two items as being the analysis unit. In practice, and considering the 'service' granularity, this means that a service is marked with the 'Correct' failure mode if this mode has been observed at least once in all tests applied to this service. This applies also to the remaining failure modes. In the same manner, if we consider the 'operation' granularity, we mark a given service operation with the 'Correct' failure mode if at least one of the tests for that operation resulted in the identification of a 'Correct response'. The 'parameter' granularity refers to each individual operation parameter in a way analogous to the remaining levels of detail referred.

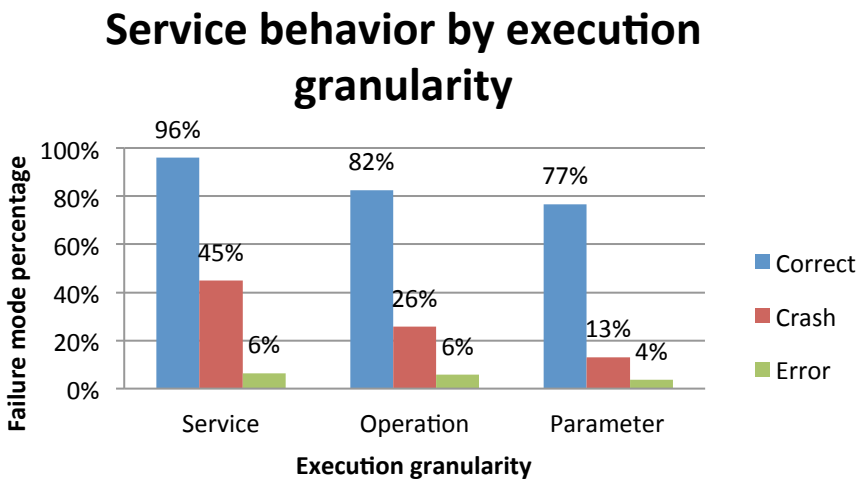


Figure 7.1 – Global robustness results.

If we consider the 'service' execution granularity, we can see that the 'Correct' failure mode was observed at least once for 96% of the services. In fact, the correct behavior is present in almost all services tested, which indicates that in some point services are able to display an adequate and expectable behavior. However, there are still relatively high percentages of the 'Crash' and 'Error' failure modes, respectively 45% and 6%, which indicates that services also display unexpected error conditions in relatively high percentages. These are obviously non-additive results as the same service can display multiple failure modes.

We can also observe that the percentages of the different failure modes generally decrease as we increase the analysis granularity. Note that each failure mode only needs to be observed once (in a given parameter), so that we mark the whole parameter, operation, and service with that failure mode. This justifies the fact that higher execution granularities generally display higher percentages for each failure mode. Despite this, the global image is maintained being the 'Correct' and 'Error' failure modes, respectively the most and least observed, whereas the 'Crash' failure mode consistently maintains its middle position.

Besides this global analysis, each response generally represents a rich resource that enables us to understand the source of failures and obtain a global view about the relative frequency of each observed behavior. The

response information obtained in each result was used to build the proposed tag-based behavior classification system (see Section 4.1.4). As the set of responses to be analyzed was very large, we analyzed and tagged each response in multiple iterations. Although we started from a base set of tags, during the analysis process more tags where created whenever a new (previously unseen) problem appeared and no existing tag could be used to accurately describe that problem. These iterations were also necessary to generalize a few tags (e.g., merge two tags into a more generic one). Generalization was applied whenever possible; however, we are aware of the intrinsic limitations of providing this kind of classification system. The final outcome of this process was the set of tags presented in Section 4.1.4.

Figure 7.2 presents the **distribution of the most frequently observed tags** (tags representing less than 1% of the total observed problems are not represented).

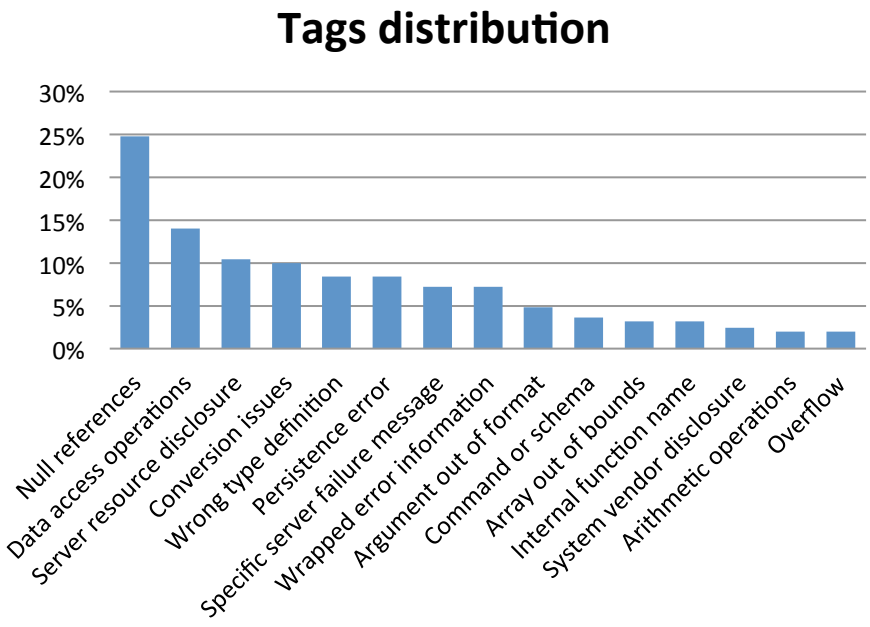


Figure 7.2 – Distribution of the most frequently observed tags.

As we can see, '*Null references*' was the issue most frequently seen in all services. This issue was present in 25% of the tested web services and is related with the fact that services typically assume that clients will invoke

their operations with non-null input parameters. Services tend to expect a correct, non-malicious client and thus provide themselves with no protection, resulting, at its best, in an unexpected exception at the client-side. Clients built on the assumption that the service is robust and that executes accordingly to some specification, can then easily collapse when in presence of an unexpected answer.

Among the most relevant issues are persistence related problems (observed in 14% of the services), which, in our experiments, were associated essentially with the use of SQL statements to access a database. This reveals more than a simple SQL construction error. In fact, it shows that the provider does not validate SQL inputs, which may offer an entry point for SQL injection attacks that can compromise the security of the web service (or of the whole service infrastructure).

'*Server resource disclosure'* was also a frequently observed issue (in 10% of the services). In fact, we frequently observed that some services, when in presence of an invalid input, disclose not only development information (e.g., a stack trace wrapped in an exception thrown at an unexpected point), but also more critical information (e.g., the partial directory structure of a hard-drive) that can represent security issues.

During our analysis we also observed that the '*Conversion issues*' and '*Wrong type definition*' tags are very frequently associated with each other. In fact, in about 96% of the observed conversion issues, the problem was caused by an incorrect definition of the parameter data type. This indicates that many robustness problems are related with the fact that developers often do not choose the adequate data types for the parameters of their services. For instance, several cases were observed in which a given service expected a Number, however the WSDL document announced that a String was required (that was later handled as a number). In these cases, besides not using the adequate data type, the service provides no adequate protection against an incorrect or possibly malicious client. Note also that the announcement of an incorrect data type in a WSDL document can result in severe interoperability issues.

Figure 7.3 presents an analysis of the **tags distribution with respect to the total tag count** (and considering '*service*' granularity). For readability, the figure presents the 8 most observed tags and aggregates the remaining in the '*Other*' group. As shown, the top 8 tags presented in Figure 7.2 (where results also represent a '*service*' granularity but with respect to the total

service count) are again displayed as top issues in Figure 7.3. Furthermore they also maintain their relative positions.

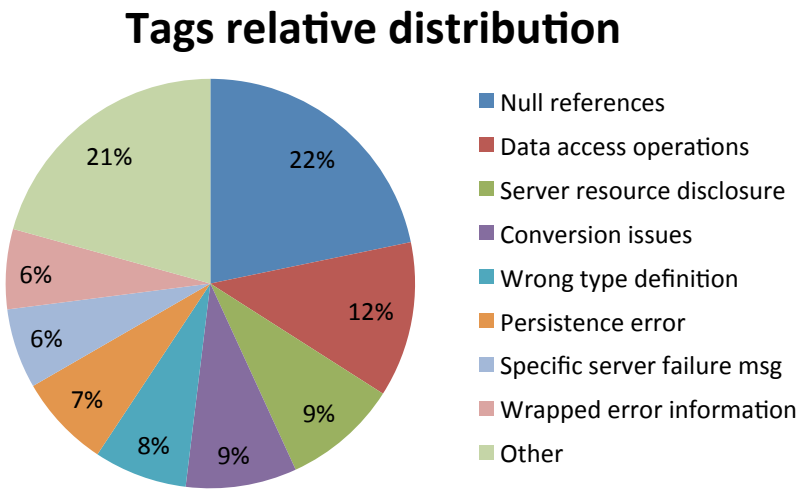


Figure 7.3 – Relative distribution of tags per total tag count.

The previous paragraphs presented the top issues disclosed in our set of experiments. However, other types of issues can be relevant even if they occur less frequently (e.g., when severe failures are involved). From the analysis of the results, it is clear that the following actions are urgently needed for robust web services development and deployment:

- Integrate robustness testing in the development process. Currently, any developer can freely use *wsrbench* to test its services for robustness. Additionally, our approach can be adapted and integrated into popular project build and management tools like Maven (Apache Software Foundation 2008d), which can then be used by developers in any Integrated Development Environment (IDE).
- System administrators should also use robustness testing, including when running legacy services. Testing will enable them to assess services in terms of robustness, and in many cases, security. Frequently service or server configuration is sufficient to hide or correct major issues.

- Better support for complete domain expression and announcement in WSDL documents needs to be included in the web services technology. Such support can help clients to execute services with adequate inputs, preventing accidental robustness problems.
- Easy support for domain validation must be available in web services development frameworks. Providing developers with easy ways to validate inputs would certainly reduce many of the observed issues.

7.2.2 Java Message Service

In this section we present the evaluation of the robustness of three different JMS providers (see details regarding experimental scenario 1.b in Section 7.1). Although the main goal of these experiments is to demonstrate the applicability of the approach to service middleware, we also want to understand if major JMS providers suffer from robustness problems, and in particular if our approach can also highlight other kinds of problems in service middleware (e.g., security and non-compliance issues).

Notice that, in this scenario we generate robustness tests at the client side, however we also have provider visibility as we are testing the full end-to-end middleware communication process in the presence of invalid inputs (and we obviously had to set up the server-side component of the tests). As such, and as stated previously, we can perform a more detailed classification of the results. Namely, we can use a detailed failure mode scale (i.e., adopt of the CRASH scale (P. Koopman and DeVale 1999)) and analyze the observed behavior in a better way, including any possible compliance disparities found.

In summary, the experimental evaluation exposed several robustness problems. The problems revealed are, in fact, of extremely high importance, as we will see ahead, and they also represent major security vulnerabilities. Several minor non-conformities were also found in all three providers tested (see our conformity scale in Section 3.4). All providers revealed Level 2 non-conformities and ActiveMQ added a Level 3 non-compliance issue to these. Table 7.III summarizes the results, which are discussed in detail in the next sections.

Table 7.III – Problems detected in the JMS implementations.

Provider	Robustness issues		Behavior tags		Compliance issues		Security issues
	Type	#	Type	#	Type	#	
JBoss MQ 4.2.1.GA	Catastrophic	1	Incorrect Object Handling	1	Level 2	4	DoS Attacks
			Invalid Input Format	4			
JBoss MQ 3.2.8.SP1	Catastrophic	1	Incorrect Object Handling	1	Level 2	4	DoS Attacks
			Invalid Input Format	4			
ActiveMQ 4.1.1	Silent	1	Invalid Input Format	4	Level 2	4	Message suppression
			Data type Usage Issues	1	Level 3	1	
			Other	1			

7.2.2.1 JBoss MQ 4.2.1.GA results

JBoss MQ was, at the time of testing, the latest production-ready messaging product available from JBoss. According to the developers, it was submitted to multiple internal tests and to the Sun Microsystems compliance tests (it is 100% Sun Compatibility Test Suite (CTS) JMS compliant) (Maron 2004). At the time of these experiments it was about to be superseded by JBoss Messaging (Red Hat Middleware 2007) and no new features were being added: the developers were at the time focusing on bug fixing.

Considering this scenario, we were expecting to find none or a very small number of robustness problems. In fact, JBoss MQ passed all the robustness tests related to message properties and body, and almost all the tests related to the header fields. However, it failed in the tests where JMSMessageID was set to null. At first, sending a message with this field set to null appeared to cause no harm to the JMS provider, as the message was correctly stored in the internal JBoss JDBC (Java Database Connectivity) database. However, when a receiver tried to read the message an unexpected null pointer exception was thrown and any subsequent reads of that message were unable to succeed (this behavior corresponds to the *Incorrect Object Handling* tag displayed in Table 7.III).

Furthermore, regular subsequent messaging operations were also affected. In fact, consumers were unable to retrieve any valid messages sent afterwards.

This **failure was classified as Catastrophic** as a corruption of the JMS server has occurred. In fact, a restart of the application server was unable to restore the normal behavior of the JMS provider. The only way we found to recover was to manually delete the invalid message information from the repository. This was possible because JBoss MQ uses a JDBC database that is easy to access and modify using SQL (Structured Query Language). If JBoss used any other proprietary repository, we would have had to delete the whole message repository. Nevertheless, both solutions are totally unacceptable and difficult to manage.

As this represents a severe failure, we then analyzed the JBoss MQ source code in order to find the root for this problem. We concluded that, although the JMSMessageID represents a unique identifier for the message, JBoss uses another variable to uniquely identify each message. This allows sending and storing messages with a null JMSMessageID. However, when a receiver fetches a message from the repository JBoss MQ tries to load messages into memory in a Map structure that uses a hash of the JMSMessageID as key (and the whole message as value). This of course produces a null pointer exception as hashing a null value is not possible.

It is important to emphasize that this serious robustness problem also exposes a severe security issue that can be exploited by hackers to cause Denial-of-Service. In fact, it is quite easy for a malicious user to generate a message with a null JMSMessageID and send it to the JMS provider and basically stop it from delivering service. This also shows the usefulness of the robustness testing approach in discovering possible security vulnerabilities.

Some minor compliance issues were also detected in JBoss MQ 4.2.2.GA. In fact, the overflow mutation (i.e., overwriting the parameter with an overflown string) applied to JMSMessageID, JMSCorrelationID and JMSMessageType fields caused a JMSException wrapping a UTFDataFormatException (corresponds to the Invalid Input Format tag in Table 7.III). This JMSException is in agreement with the standard, which specifies that this exception must be thrown if the provider fails to send the message due to some internal error. Nevertheless, the specification does not impose the restriction that the value for these header fields has to

be handled as a UTF string. This is the cause for the UTFDataFormatException as the chosen method to write the string imposes a limit on its length (Green 2007). Note that, choosing a different writing method would easily solve this issue. The same happens while setting a property name with an overflowed string (it does not happen while setting the value for the property). These are both **Level 2 compliance issues** as they represent non-severe problems related to what is defined by the JMS specification.

7.2.2.2 JBoss MQ 3.2.8.SP1 results

Legacy or older application servers are still used frequently in enterprises as upgrading a system to a newer version may have a high cost. In this sense we found interesting to test the previous JBoss MQ major version. With this experience we also wanted to verify if the serious robustness problem detected in the latest version also existed in the previous version. Again, the same **Catastrophic robustness failure** occurred in this version. In fact, the results observed (for all three characterization dimensions) were exactly the same for both JBoss MQ versions tested. Note that, even though this middleware is normally submitted to a large battery of tests (as stated by the developers), a major robustness problem and security vulnerability has passed silently between versions. This is something that could have been easily avoided by testing the middleware using our testing approach.

7.2.2.3 ActiveMQ 4.1.1 results

Apache ActiveMQ is a widely used open source JMS implementation, for which we were also expecting to discover a few or no robustness problems at all. Testing the message properties and body revealed no robustness problems. However, a failure was triggered while setting the JMSMessageID to a predefined value. In practice, mutating the JMSMessageID of a single message posed no problem. However, tampering a second message (before delivering the first to the consumer and, consequently, removing it from the repository) caused the first message to be overwritten (tagged as '*Other*' in Table 7.III, as it does not produce a directly observable behavior). Note that the JMS specification states that the JMSMessageID field contains a value that uniquely identifies each message. Overwriting messages represents an

unacceptable behavior as it may lead to messages corruption and losses. After analyzing the source code, we concluded that this implementation uses this field as a unique identifier for the messages. Although this is an acceptable and natural approach, exceptional cases like duplicated values should be verified. This failure was **classified as Silent** since an operation was incorrectly completed and the JMS server indicated no error.

The above robustness failure can also represent a security issue. In fact, security attacks that try to generate duplicate message IDs are quite easy. Original messages will silently disappear from the provider's repository and will never be delivered to consumers. Similarly to JBoss MQ, some minor compliance issues were detected. JMSMessageID, JMSCorrelationID and JMSType are susceptible to the overflow mutation. In fact, this implementation imposes a limit on the size of these fields (2^15 – 1 characters) that is different from the one imposed by JBoss MQ (and both are incorrect as there is no limit defined in the standards). This same limitation is found in the size of the properties keys that, when violated, invalidate message delivery. These were classified as **Level 2 non-conformity issues** (and are marked with four '*Invalid Input Format*' tag instances in Table 7.III), as they correspond to non-severe restrictions to the standards.

Another compliance issue was detected while testing the message properties. The JMS specification states that (Sun Microsystems, Inc. 2002):

"The setObjectProperty method accepts values of class Boolean, Byte, Short, Integer, Long, Float, Double, and String. An attempt to use any other class must throw a JMSException."

This was not the case with this implementation. In fact, it accepts a few other data types (e.g., Map, List, and Date). This represents a **Level 3 non-conformity issue** (and corresponds to a '*Data Type Usage Issue*' tag in Table 7.III) as it is clearly related to a specification extension. Note that a generic JMS based application may have problems if it is built upon the assumption that these data types are not allowed.

7.2.2.4 Additional comments on the results

Although only catastrophic and silent failure modes were observed, we believe that the remaining failure modes defined by the CRASH scale can still be useful. Our results are obviously dependent on the JMS providers tested and cannot be generalized. Additionally, we believe that there are

many JMS implementations where restart and abort failures are triggered when invalid parameters are provided. Indeed, some preliminary tests on the latest Java EE 5 (Sun Microsystems Inc. 2007b) JMS reference implementation (OpenMQ 4.1 (Sun Microsystems Inc. 2007a)) exposed an **Abort failure**. In this version of OpenMQ, setting an Object property to null exposes a robustness problem. This is caused by the fact that this implementation checks if the object type is one of those allowed by the specification. If it is not an allowed type (as in the case of null that has no type), then it tries to build a String (for logging purposes) with information of the object itself. This produces a null pointer exception, as the null object does not contain any information.

The information provided by our robustness testing approach is of high utility if one wishes to choose a robust JMS provider. It can also provide valuable information for providers to improve the quality of their solutions, enabling the correction of issues that lead to robustness and security problems. In addition, we observed several compliance issues and this is extremely important, particularly in compound services (where interoperability issues are vital). In this sense, the approach shows to be highly valuable not only for client developers (it enables the selection of the implementation that shows less compliance disparities), but also for middleware developers (it enables the improvement of the level of conformity to the standards). In addition, robustness testing approaches that run at the server-side may be even more helpful in disclosing robustness issues, providing also a basis for improvement in a more integrated way. This is the main general topic presented in the next section.

7.3 Server-side robustness testing results

In this section we apply our robustness testing methodology at the server-side, as proposed in Chapter 5. The set of TPC-App services and open-source services described in Section 7.1 was used, as basis for the tests. The initial step was to analyze the WSDL document and XML Schema file (XSD) of each web service. Each input and output parameter in the XSD file was manually extended to include domain restrictions, using the standard XSD *restriction* element referred earlier (refer to Section 5.1 for details). It is important to indicate that the TPC-App domains were defined based on the TPC-App specification document (Transaction

Processing Performance Council 2008). On the other hand, the open-source Internet services did not include any kind of specification, so their domains had to be inferred by a programmer external to the experiments (this process involved a code analysis and a detailed view of the database contents of these services). To reduce the possibility of human error, another developer verified the inferred domains (both developers had more than 2 years of experience in developing and testing service applications). The final step of the interface definition was to use EDEL to express the final domains. The full schemas can be found at (Laranjeiro 2011).

In these experiments we used wlgenerator (Laranjeiro 2011) and a random value generation strategy to create a service workload. Before proceeding to the tests we analyzed the coverage of the workload using Cobertura (Doliner 2008). As we can see in Table 7.IV, the coverage is in general above 75% (except in one case), a value frequently placed in an acceptable range (Kaner, Falk, and Nguyen 1999). However, we decided to analyze the source code of all versions to understand what code was not being covered. In all cases it corresponded to unused exception catch blocks. Our workload was able to exercise the useful source code, but was not expected to trigger any error-handling blocks (this is the goal of the invalid inputs set by the robustness tests and not of the workload itself). Thus, we considered the workload adequate for all services.

Table 7.IV – Workload coverage.

Web Service		Version		
		A	B	C
TPC-App	ChangePaymentMethod	92%	87%	91%
	NewCustomer	80%	92%	93%
	NewProducts	74%	80%	87%
	ProductDetail	94%	80%	87%
Public	JamesSmith	90%		
	PhoneDir	76%		
	Bank	83%		
	Bank3	75%		

After running the workload, the web services responses were analyzed in order to identify deviations from the expected output domain that could indicate software defects, which was not the case. Therefore, we proceeded with the execution of the robustness tests and collected the results.

Table 7.V summarizes the results of the experiments. The first two columns identify the service and operation tested. Under the CRASH column header we can find the count of observed failure modes for each tested operation, classified using the CRASH scale as reference. Only 2 failure modes were observed, so only two sub-columns are present (A and S correspond to 'Abort' and 'Silent'). For each operation, this value counts each unique observed behavior only once, except if it is present in multiple parameters. In this latter case, the observed behavior is multiplied by the number of times it was observed in different parameters (but at most once per parameter).

The 'Observed Behavior' column in Table 7.V presents a detailed view of the issues found in each tested operation, using the service behavior tags shown in Table 4.IV. The number of problematic parameters per tag (or per set of tags) is also indicated between parentheses in this column. The last table column indicates the number of problematic parameters (P) with respect to the total number of parameters for the operation (T). Each robustness problem identified represents either a regular result that falls outside of the specified valid response domain, or an exceptional result that is not one of the declared web service exceptions.

Table 7.V – Robustness problems observed for the TPC-App and open-source web services.

Service	Operation	CRASH		Observed Behavior	P/T
		A	S		
TPC-App A	changePayment Method	3	1	Null references (1); Out-of-domain responses (1); Data access operations - Persistence error (2)	3/4
	newCustomer	18		Data access operations - Persistence error (16); Specific server failure message (1) out-of-domain responses (1)	16/16
	newProducts	1		Data access operations - Persistence Error (1)	1/3
	productDetail			-	0/1

TPC-App B	changePayment Method	4	1	Null references (2); Out-of-domain responses (1); Data access operations - Persistence Error (2)	2/4
	newCustomer	17		Data access operations - Persistence Error (14); Null references (1); Argument out of format (1); Overflow (1)	15/16
	newProducts	1		Null references (1)	1/3
	productDetail	1		Data access operations - Persistence Error (1)	1/1
TPC-App C	changePayment Method			-	0/4
	newCustomer			-	0/16
	newProducts			-	0/3
	productDetail			-	0/1
JamesSmith	login			-	0/2
	add			-	0/11
	update			-	0/12
	delete			-	0/1
	search	4		Null references (1); Data access operations (3)	3/9
Bank3	deleteAcc	2		Null references (1); Conversion issues - Wrong type definition (1)	1/1
	deposit	5		Null references (4); Conversion issues - Wrong type definition (1)	4/4
	displayDeposit	1		Data access operations - System Vendor Disclosure (1)	1/2
	displayInfo	1		Null references (1)	1/2
	newAccount			-	0/4
	Withdrawal			-	0/4
Bank	balance			-	0/1
	create			-	0/2
	deposit	2		Null references (1); Conversion issues - Wrong type definition (1)	2/2
	sign			-	0/1
	withdraw	2		Null references (1); Conversion issues -Wrong type definition (1)	2/2
PhoneDir	addNewRecord			-	0/2
	deleteInput			-	0/1
	firstNameWithIt			-	0/1
	modify	2		Wrapped Error Info (2) - Data access operations - Persistence error (2) - System Vendor Disclosure (2)	2/2

As we can see, several robustness problems were found in the tested services. Concerning the observed failure modes, we detected the **'Abort'** failure mode in 15 out of the 32 tested operations. On the other hand, we were able to observe the **'Silent'** failure mode in 2 operations where the service replied with an out-of-domain (as defined in the TPC-App specification) response. These two cases do not correspond to compliance issues as they were triggered with invalid inputs; these would only be a compliance issue if the triggering inputs were valid. Figure 7.4 presents an overview of the behavior tag distribution considering the '*service*' granularity and with respect to the total count of tags found (using this granularity).

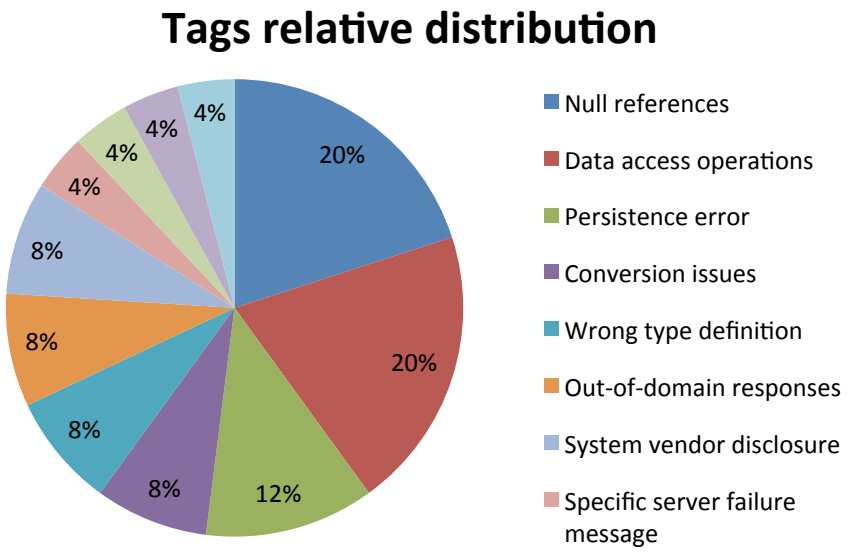


Figure 7.4 – Relative distribution of tags per total tag count (service granularity).

We can observe that the '*Persistence error*' tag has now more weight than what was observed for the public services. It changed from being the sixth most observed tag to the third most frequently observed tag (see Figure 7.2). Furthermore, the '*System vendor disclosure*' tag was not present in the top tags analyzed in Figure 7.2. On the other hand, the '*Server resource disclosure*' tag, the third most observed in the public services, was not detected in the home-implemented ones. Obviously, these changes simply reflect the specificities of this smaller set of services and cannot be

generalized. However, and in general, the most frequently observed behaviors also fit in the top behaviors observed during the public services tests. '*Null references*'; '*Data access operations*'; '*Persistence error*'; '*Conversion issues*'; and '*Wrong type definition*' were again top detected issues. The following paragraphs present a more detailed analysis of the results in both sets tested: TPC-App and open-source web services.

The results displayed in Table 7.V show a lot of diversity and, regarding the TPC-App implementations, we found no robustness issues at all in **TPC-App Version C**. In fact, a later analysis of this version's source code confirmed that the programmer took into account expectable erroneous input data, and for all tested cases a check was in place to verify if the incoming data was valid. On the other hand, several robustness problems were observed for versions A and B. Seven out of eight service operations were marked with the Abort failure mode (3 and 4 for implementations A and B, respectively). Two out of eight operations displayed the Silent failure mode, as referred earlier.

TPC-App Version A presented problems in 3 of the operations. The vast majority of the 22 observed issues manifested as unexpected exceptions due to erroneous construction of SQL commands used in database access operations (19 detected issues). This is explained by the fact that these services directly use client input to dynamically build SQL statements. More than robustness problems, this type of problems usually represent security issues, as typically we observe this behavior in services that do not use parameterized data access queries (which was, in fact, the case of this version). This is a major concern as it can be an entry point for SQL or XPath Injection attacks, two of the most frequent attack types in the web environment (Christey and Martin 2007). The remaining issues included out-of-domain responses, null references, and specific server failure messages. A great number of issues was uncovered in the NewCustomer service. In fact, this service has a large number of parameters and no appropriate input validation was in place.

Twenty-three issues where observed in **TPC-App Version B**, including problematic database access operations (17 detected issues), null references (4 issues), exceptions indicating an out of format argument, a stack overflow exception, and responses falling out of the expected domain (1 instance observed per each of these three issue types). We did not disclose any robustness issues related to external service invocations (in both versions).

Although we found several problems in the TPC-App services, a relevant robustness problem was observed for the *newCustomer* service in implementation A. In fact, although the code targeting the validation of a *contactEmail* parameter was in place, too large email addresses caused the web service to throw a *StackOverflowException*. After some analysis of the code we concluded that the problem resided in the external API that was being used to validate email addresses (Jakarta Commons Validator 1.3.0 (Apache Software Foundation 2010b)). This shows that robustness problems may occur even when programmers pay a great attention to the code correctness. In fact, the use of third party software (as is the case in this example) may raise problems that are not obvious for developers. Furthermore, this type of errors can easily appear or disappear when an apparently harmless update is done to the external libraries commonly required by projects. However, they can be easily detected with the help of our robustness testing approach.

Regarding the **open-source services**, 40% of the operations tested presented some kind of robustness problem (8 out of 20 operations). Again, null references and database access operations were predominant; however we also detected several data type conversion problems during the experiments. After a source code examination we were able to conclude that the developer of the services presenting this issue misrepresented the data type in some operations. As an example, we can refer the case where a particular operation expected a numeric value but the operation interface was coded as requiring a String. In these cases, besides not choosing the adequate data type, the service provides no adequate protection against an incorrect or possibly malicious client. It is important to emphasize that the data type of the operation is exported along with the WSDL document, and the announcement of an incorrect data type in a WSDL document can result in severe interoperability issues. This had already been a frequently observed issue during the public services tests and a major source of robustness problems. We also observed that in two operations the service response included information about the database vendor, engine, and version. Besides being a robustness problem, it again translates into a severe security issue that can be maliciously exploited, justifying the need for robustness testing.

This experimental evaluation shows the effectiveness of the proposed methodology when used in typical web service development scenarios. From the provider point-of-view, the results show that some software improvements are needed to solve the robustness problems detected. In

fact, after performing the robustness tests we forwarded the results to the programmers in order to get the implementations improved. Through a detailed analysis of the results the programmers were able to identify solutions for the existing software faults and new versions of the services were developed. New robustness tests were then executed for these new versions and no robustness failures were observed. This shows that our methodology can be extremely useful, as it can help web service developers to assess (and later improve) the quality of their code (in terms of robustness). It can also serve as basis to compare different implementations of a given service or operation. For instance, when a service client needs to use the TPC-App *newProducts* service, and has the possibility to choose from versions A and B presented earlier, it is clear that he should select version A where no robustness problems were found (1 issue was found in version B). Despite the advantages referred, manual improvement may not be practical due to aspects like software complexity, limited access to source-code, etc. Thus, after collecting these results, we studied the applicability of our automatable technique for robustness improvement to these sets of services. The next section presents the obtained results.

7.4 Robustness improvement results

This section presents the results obtained by applying the robustness improvement technique explained in Section 6.1, in particular the use of robustness/security wrappers to protect services against inputs that can expose robustness problems. Section 7.4.1 presents the results obtained with our robustness improvement approach created to protect services against invalid inputs, whereas Section 7.4.2 focuses on the experiments conducted to illustrate the mechanism for protecting services against malicious inputs.

7.4.1 Protection against invalid inputs

The first step required to improve a service's robustness is wrapping it with a protective application layer that is responsible for performing input validation. However this is not the only step required. In fact, after applying the wrapper, it is recommended to execute new robustness tests (to verify if there are still problems that need correction) and execute a

workload (to verify if there are any deviations regarding the expected service behavior).

In this section we wrap the TPC-App versions and open-source services used in the robustness tests presented in Section 7.3, with the goal of protecting them against invalid inputs. As we had already defined the domains for all service operations using EDEL, it was fairly easy to create the validation wrapper using the technique described in Section 6.1. We recompiled the services using AspectJ's compiler, which introduced our validation procedure in all required points (operation entry points and calls to external services). We applied full validation in all tested services (as opposed to partially protect the services with our wrapper, delegating part of the validation to the original service).

After creating the wrapper, we deployed the services and generated **new robustness tests**. Notice that the tests are not generated in a deterministic way and variations can exist, depending on the workload used or on the specific test itself. Moreover, services may also be non deterministic, as this characteristic depends on the service being tested. In addition, as the domain definition procedure is a manual step (and therefore prone to errors), residual issues may remain. These aspects essentially mean that we need to execute robustness tests after wrapping to have a higher degree of confidence on the robustness of the services that are being deployed.

After executing the robustness tests, we analyzed the results, which did not indicate the presence of robustness issues. This confirms the utility of the improvement procedure not only in removing the problems but also in the amount of development effort required. In fact, after having the domains defined, the effort involved is reduced to a service recompilation using AspectJ and our crosscutting validation concern.

Despite the fact that the tests showed no issue, there is a possibility that, accidentally, our mechanism could introduce a deviation in what is the expected service behavior. Therefore, we re-ran the original **workload** for all services used in the experiments (see Table 7.II). The web services responses were automatically analyzed by a component of our robustness improvement tool to identify potential deviations from the valid output domains. As expected, no problem was identified (we also double-checked each result manually), providing a strong indicator that our framework did not change the application's normal behavior. Note that, for deterministic operations, this run's output must be exactly equal to the

output of the first workload run (as discussed in Section 6.1). However, for services that do not deliver equal answers for distinct invocations (with the same input), the framework's only option is to check if each response fits the specified domain or not.

As a final experiment, we executed a test to assess the **performance impact** related to the introduction of the robustness improvement mechanism. We executed the original workload, with the validation mechanism in place, and found that the best executing time was of 7,338 ms (± 0,746), being the worst execution time of 16,093 ms (± 0,961) for a total of 10000 single operation executions for each service. To obtain these measurements we used a Java method that provides nanosecond precision (but however does not guarantee nanosecond accuracy). We initially ran baseline performance tests for each service (30 minutes each test and using the original workload) and were able to verify that these values represent 6% (best case) and 48% (worst case) of the average execution time of the original unmodified services. This is an acceptable value, particularly considering that it fully protects the services (in terms of robustness) and that in these environments high execution times are frequent (Bovy et al. 2002).

The results presented above show that the robustness improvement technique is able to remove robustness problems and prevent the execution of a service with invalid inputs. Despite this, there are still valid inputs (from a domain point-of-view) that may be very problematic to the web service, in particular inputs that exploit service vulnerabilities. In the next section we present the application of the technique to prevent the execution of malicious inputs over vulnerable services.

7.4.2 Protection against malicious inputs

This section presents and discusses the experimental evaluation performed to illustrate our approach for protection against command injection (SQL and XPath Injection) attacks, as proposed in Section 6.2. We initially perform a security assessment over the set of services being tested, we then apply our approach for learning and protecting services against SQL/XPath Injection attacks and verify the services behavior.

7.4.2.1 Services assessment

The first step of this experimental evaluation consisted of **identifying potential vulnerabilities** in TPC-App version A and in the open-source web services described in Section 7.1. This information was used later to verify the effectiveness of the proposed protection scheme by re-assessing the protected services. As referred earlier, we are testing TPC-App version A, the original version without any wrapping validation code (discussed in Section 7.4.1) in place, as the goal is to verify the effectiveness of this mechanism on its own. Ideally, both mechanisms should be used to provide full protection against invalid and malicious inputs.

Three well-known commercial vulnerability scanners were used to test the services for vulnerabilities: Acunetix Web Vulnerability Scanner; IBM Rational AppScan; and HP WebInspect (Acunetix 2011; IBM 2011; Hewlett Packard 2011). The results were collected exactly as indicated by each scanner and double-checked manually (to discard potential false-positives). Due to scanner usage restrictions, we will refer to these scanners, from this point onwards, as VS1, VS2, and VS3, in no particular order.

The use of vulnerability scanners in this phase enabled us with an initial characterization of the web services. However, it is well known that even the best top current scanners are unable to present accurate results (Vieira, Antunes, and Madeira 2009), so we asked a team of security experts to perform a code inspection and execute penetration tests to detect additional vulnerabilities. Each individual test was stored by SoapUI as we had the goal of using it later to assess the effectiveness of our protection mechanism. The security analysis team was composed of 5 elements. Three of these elements are experienced developers on developing database centric business critical web applications in Java (about 3 years of experience). The remaining two are security researchers, one junior (one year of experience) and one senior (four years working on security related topics). The team's results represent the union of the vulnerabilities detected by each team member. In this case, one vulnerability was counted for each web service input parameter used in a statement in a vulnerable way. For instance, if a given operation accepts a single parameter, and this parameter is used in a vulnerable way in ten distinct code locations, we count a total of ten vulnerabilities. It is important to mention that, as before, we double-checked the

vulnerabilities pointed out by each participant (under the form of an example service request) to discard potential false-positives.

Table 7.VI summarizes the results obtained. False positives are indicated between parentheses. These detected problems correspond entirely to SQL injection issues, as both the TPC-App specification and the public services do not include any XPath usage. However, the approach is essentially the same, the main difference is the language's syntax.

Table 7.VI – Vulnerabilities detected.

	Web Service	VS1	VS2	VS3	Experts
TPC-App A	ChangePayment	2	2	0	2 (2fp)
	NewCustomer	15	15	2	19 (1fp)
	NewProducts	1	1	0	1 (1fp)
	ProductDetail	0	0	0	0
Public	JamesSmith	3 (5fp)	(1fp)	0	20
	PhoneDir	3	3	3	6
	Bank	2	0	0	4
	Bank3	3	4	0	13

Concerning the scanners results, it is possible to verify that all services, with the exception of one (ProductDetail), presented some kind of vulnerability (the newCustomer service was the service with more disclosed vulnerabilities). Considering the code inspection results, we can see that 3 of the TPC-App services were reported as vulnerable and for one in particular 19 security flaws were detected. This value is essentially due to a large number of user input parameters, being used in more than one SQL statement throughout the code. A large count of vulnerabilities was also obtained for the JamesSmith and Bank3 services. These services use a great number of input parameters, include multiple operations, and some of their operations execute more than one SQL statement in a vulnerable way, which justifies the large number of vulnerabilities found in these two cases.

7.4.2.2 Statement learning

In these tests we opted to re-use the workload generated for the robustness tests (see Section 7.3), as we already had the domains defined.

The workload coverage results (in general above 75%) are shown in Table 7.IV. We performed a second verification of all code not being covered, and concluded that the not covered code had no influence on the total number of SQL statements that could be reached (the code not being covered had no data access statements and it's execution would not change the data access statements being covered). As such, we considered the generated workload adequate for all services being tested.

The workload was then applied to exercise each service operation in order to learn the expected SQL commands. After the learning process, we manually checked whether all possible SQL commands executed by the service application were correctly learned by our mechanism, and that was effectively the case. This learning process is quite important in our approach and is directly influenced by the coverage of the workload used. If there were commands not learned in this phase we would have to increase the size (and coverage) of the workload. The learning process was quite fast taking only a couple of minutes and resulting in a total of 16 and 24 invariant data access expressions for the TPC-App and open-source web services, respectively. Although the process is quite fast, it can obviously vary depending on the size of the workload, the amount of coverage desired, or the type of service being executed.

7.4.2.3 Improving security

After the learning phase, we configured our mechanism to enter the protective state and detect malicious commands. The 3 vulnerability scanners were then used to re-test all services for security vulnerabilities. A total of 146 attacks were executed (VS1 executed 59 attacks, whereas VS2 and VS3 executed 77 and 10 attacks, respectively) and the results were a total of zero disclosed SQL/XPath injection vulnerabilities for all services. For example, in the first assessment, the '*Change Payment Method*' service presented a vulnerability when one of the scanners replaced a particular parameter with (') resulting in a '*quoted string not properly terminated*' database error message. With our protection mechanism in place, this type of request corresponds to the generation of a new checksum not detected in the learning phase. This and all new malicious requests were indeed stopped, preventing any further service execution and possible security consequences.

We then replayed all malicious requests crafted by our code inspection participants using the attacks stored by SoapUI for each individual team

member (a total of 65 requests, see Table 7.VI for detailed values per service). All attempts to inject SQL code were aborted by our mechanism by throwing the SecurityRuntimeException exception. There were no false-positives (a genuine data access statement execution identified as malicious) at this point, as all requests used were malicious and would effectively result in the execution of a data access statement in a vulnerable way. As such, the only remaining possibility of obtaining an incorrect behavior (from the security mechanism) would be the presence of false-negatives (a malicious data access statement being silently executed, i.e., not detected by our mechanism), which also did not occur. Note that our mechanism was not specifically tuned to stop any particular attack. The success of this mechanism is completely related with the process of converting multiple genuine requests executed during the learning phase into invariant commands, thus being fully prepared to detect any abnormal request after the learning phase.

To evaluate the performance of the filtering mechanism, we ran an additional set of experiments. We disabled the invariant determination mechanism and maintained only the regular expression filter. We then re-executed the security scanners over the services. Table 7.VII presents the number of attacks (including multiple variants for a single service parameter) defended by the filter (column 'defended attacks') with respect to the total requests originally marked as successful by each scanner (in column 'tested attacks').

We found that the filter was able to block 76% of the attacks originally marked as successful by each scanner (i.e., when testing the services without any protection mechanism). This represents an extremely solid behavior for a generic filter that is only to be used as a second barrier.

Table 7.VII – Filter performance by scanner.

Scanner	Tested Attacks	Defended Attacks	Defense Success %
VS1	59	49	83%
VS2	77	52	68%
VS3	10	10	100%
Total	**146**	**111**	**76%**

Although our mechanism showed a good performance, we still wanted to check if it could accidentally trigger a security alarm (i.e., a false-positive) when in presence of genuine accesses. So, we deployed a trained version of the TPC-App services and ran a client emulator (this client is part of the TPC-App specification and was not used in any other part of the experiments) during 60 minutes to see if the security mechanism would trigger a false-positive (the invariant determination or filter components reporting an attack when in presence of a genuine statement). During this process 26092 different data access statements were executed and no false-positives were detected (i.e., no security exception was thrown by the protection mechanism). This is due to the completeness of the learning phase, which was able to learn all possible invariants, being then prepared not only to detect malicious statements, but also to allow the execution of genuine statements. This later aspect is obviously important for users that do not want an application's functionality to break due to any kind of security procedure.

7.4.2.4 Behavior verification and comments

To verify if the protection mechanism changed the services functionality, we re-submitted the workload to the protected application. The services responses were analyzed to identify deviations from the valid output domains. As expected, no problem was identified, providing a strong indicator that our framework did not change the application's normal behavior.

A final test was conducted to assess the performance impact of the invariant conversion and filtering processes. As we were expecting small values, we tested the worst case scenario found in the TPC-App services – the service with more invariant statements to check and the data access expression matching only the last element of the learned statements. We executed 500000 invocations using that worst-case scenario, and the mechanism took on average 0,190 ms (± 0,075) to execute, less than 0,11% of the total time for the fastest executing service.

To summarize, the approach showed to be totally effective in securing a set of open-source services and TPC-App web services against SQL Injection attacks. The process is similar if the goal is to protect a given service against XPath Injection attacks and the mechanism can be used to protect against both types of attacks at the same time. The developer just needs to add the appropriate method signatures that require interception

(and protection). This technique does not require source code access, and as such, during the experimental phase, no complexity is added to the services code. This fact is very important for protecting legacy services, where no source code is available, and results in a helpful tool for developers and service administrators.

7.5 Conclusion

This chapter presented an experimental evaluation carried out to illustrate the effectiveness of robustness testing and the techniques to improve services in presence of invalid and malicious inputs. Concerning the testing approach, we considered several scenarios: tests executed over a large sample of publicly available web services; tests performed to assess the robustness of messaging middleware; and tests carried out in a development environment using in-house implementations of the TPC-App web services, but also using open-source services (third-party implementations). In all cases, the approach clearly showed its value by disclosing multiple issues.

The public web services tests showed the predominance of persistence related failures, conversion issues associated with the incorrect definition of data types, server information disclosure, and also null references issues, among others. These problems can impact the security of a service and the interoperability with other systems, however they can be identified with robustness testing.

Tests executed with major JMS middleware providers again revealed the presence of key robustness and security problems, but also the presence of specification compliance issues. These can have severe effects in service environments, ranging from potential exposure to DoS attacks to interoperability issues, which are especially relevant in compound services. Finally, tests executed at the server side over the TPC-App and open-source services confirmed the usefulness of the robustness testing approach disclosing several issues in the tested systems, ranging from unexpected exceptions to information disclosure, indications of presence of SQL Injection vulnerabilities, among others. The services characterization performed also emphasized the utility of robustness testing when used for comparative purposes. In fact, we had already observed that this technique could be used to compare different systems (for instance, for selecting a more robust service middleware). Robustness

testing applied at the server-side again confirmed that the results can be used to differentiate and select the most robust services.

At the server-side, EDEL was presented as a basis for robustness testing and improvement. The robustness improvement technique prevents the execution of services with invalid inputs, with the added advantage of considering domain dependencies between inputs. The use of the technique is quite simple, and showed to effectively protect a set of TPC-App and open-source services. Although the input validation mechanism performance can obviously vary with the amount of validations to execute, in general the global impact detected on these services was quite low. Similarly, valid but malicious inputs can be effectively handled by our approach for securing services against SQL/XPath injection attacks. We selected a set of services (TPC-App and open-source) to assess the approach's effectiveness. After exposing the services and mechanism to a training period, the mechanism was able to learn all invariants from specific data access statements, showing to be 100% effective in stopping all tested attacks. The configurable filter also showed a good performance, preventing 76% of attacks generated with well-known commercial security scanners. A solid performance for a component designed to be used as a second attack barrier.

The results presented in this section demonstrate the importance of conducting robustness tests before service deployment and show the gains obtained with our improvement solutions. Developers can make use of these techniques to assess and improve the robustness and security of their service applications. Additionally, service providers now also have an easy way of fixing robustness/security problems in already deployed services.

Chapter 8
Deploying Fault Tolerant Web Services

This chapter discusses the design and application of fault-tolerant mechanisms in the web services context. In particular, we propose the use of design diversity to improve dependability-related properties (e.g., availability, response correctness, and response time) that have particular relevance in service applications. Besides proposing a mechanism that can in fact use design diversity, we also make use of historical behavior data to perform informed runtime decisions that can have a positive impact on the overall behavior of the service (or sets of services). The ultimate goal is to provide services that can deliver high availability, high correctness, and low response times, in a way that is adequate to the users requirements.

Distributed applications over unreliable transport channels such as the Internet usually suffer from long response delays or temporary unavailability. For example, in a web service composition when a component service fails or shows bad performance the entire service composition may be affected. In fact, as a composition consists of a sequence of different web services invocations, where the results from the execution of one web service can be passed to the next in an atomic manner, if a web service fails the entire execution must be aborted. Obviously, as previously referred, providers can try to guarantee high

dependability for the infrastructures of their responsibility but may not be able to guarantee particular quality attributes for services that are provided by external entities.

Redundancy is used in several contexts to tolerate failures and attain high availability (Marcus and Stern 2003). Typically, the solution to handle design faults is design diversity, where distinct implementations of a component or service are used to detect and tolerate faults (A. Avizienis and Kelly 1984). Despite this, current support for developing web services compositions does not provide the features needed by programmers to easily create services with fault-tolerance characteristics.

When fault tolerant services are required, developers have to select the component web services and have to include, in the composition, all the code required for redundant invocation and responses validation and selection. This is a very difficult, high time consuming, and error prone task for several reasons: concurrent programming (e.g., using threads) can be complex; there are no practical evaluation mechanisms available; voting mechanisms must be embedded in the composition; the comparison of results is difficult as different services may respond using different data formats (the same is true for input parameters); voting impasses are difficult to resolve; deadlines for execution are difficult to implement and detect; etc.

To overcome these difficulties, in this chapter we propose a mechanism that automatically deals with all the aspects related to the redundant web services invocation and responses voting. The mechanism, named FTWS (Fault Tolerant Web Services), allows programmers to specify alternative web services for each operation and offers a set of artifacts that simplify the software design and coding process. This way, developers need only to perform a small set of configurations and provide (if necessary) some input/output adapters.

FTWS includes an evaluation functionality that helps assessing and comparing alternative services starting from the development phase. At runtime, the evaluation mechanism performs a continuous assessment of the services based on their behavior during operation. This data is used by a voting mechanism to solve impasses. Three core metrics are considered: response time, availability during operation, and correctness of the results. A composed metric based on these three can also be defined. FTWS can be currently used with stateless web services. Stateful services

imply tackling an additional set of challenges that are not covered here and remain open for future research.

This chapter is organized as follows. Next section presents an overview of the fault tolerance mechanism designed to overcome the previously mentioned difficulties. Section 8.2 presents the core metrics used by the mechanism and Section 8.3 describes the mechanism's different operation modes. Section 8.4 describes the technique used to adapt distinct services responses and Section 0 presents the implemented voting mechanisms included in FTWS. Section 8.6 illustrates the use of FTWS and describes the experiments carried out to demonstrate its effectiveness. Finally, Section 8.7 concludes the chapter.

8.1 Fault tolerance mechanism overview

The use of FTWS requires the modification of the typical web service compositions environment. Figure 8.1 presents a web services composition scenario that has been modified to include the component that is able to provide fault tolerance. This scenario includes regular service clients, the service composition and its supporting infrastructure (an application server and a business process engine), and external services that are used, at particular points, by the service composition. As shown in Figure 8.1, instead of directly invoking component web services, a composition should invoke a proxy web service (a new element in the environment). The proxy web service then executes the alternative component services in a redundant manner and deals with all the voting and evaluation issues.

The implementation of this mechanism raises several relevant problems, namely:

- **Evaluation of alternative web services:** as mentioned before, we believe that it is important for developers to have a practical mechanism to evaluate and compare alternative web services. Additionally, evaluation metrics can also be used to overcome voting impasses and achieve low-cost redundancy (see details below). This way, the mechanism must be able to perform offline (before deployment) and online (during utilization) evaluation of the component web services.

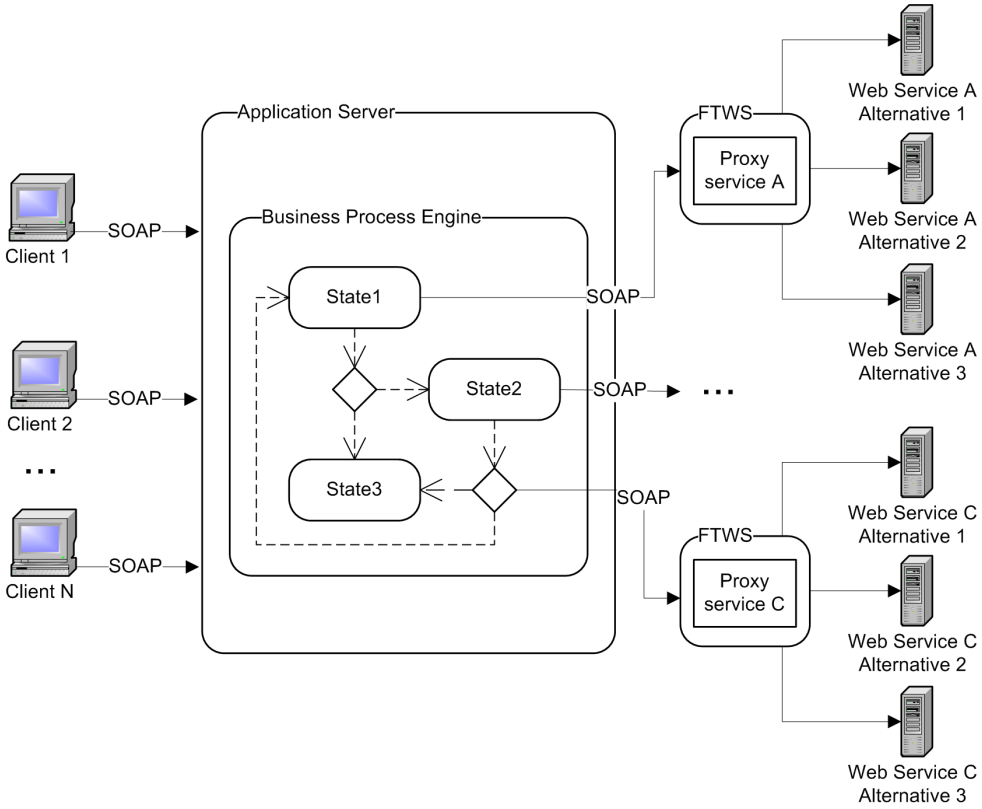


Figure 8.1 – Modified environment for fault-tolerant web service compositions

- **Implementation of redundant invocation:** low performance overhead is obviously a major issue. Additionally, as compositions use alternative component web services that may provide slightly different interfaces and response formats we need to define an easy way for the programmers to reconcile input and output data (preferably some standard adapters should be defined). Finally, concurrent execution must be implemented and the definition and detection of deadlines must be possible (deadline violation may indicate that a given web service is unavailable).
- **Voting:** voting algorithms are well understood in the literature (Parhami 1994). A major issue in this context is to resolve impasses. In fact, as typically there is a small number of alternative web services available (sometimes it is difficult to find more than two or three alternative services) impasses may become a major issue. For example, when only two web services are available and

both provide different results, there is an impasse and no result can be obtained. However, in some cases it is well known that one of the services provides correct results with a higher rate than the other (this can be observed during testing or based on previous utilization history). A key goal is then to endow the voting mechanism with an impasse-solving feature.

- **Low-cost:** the use of commercial web services is typically paid. This way, an infrastructure based in redundancy implies additional costs for the provider of a web service composition. For example, if a given composition uses five alternative web services to perform a given operation, the cost of that operation is, in average, five times higher than if a single web service is used. Obviously, if the goal is high-dependability this may not be a major issue. However, in many cases the provider may want to reduce costs while providing a robust service. Our goal is to endow the mechanism with the capability of switching between an execution mode based on the simultaneous execution of all the alternative services and a basic recovery blocks (Horning et al. 1974) execution mode (sequential web services execution). This consists in the execution of the alternative web services in a sequential manner until a response is obtained. Obviously, the identification of the execution order is a problem that must be addressed.
- **Consistency in stateful web services:** consistency is a major issue in web services when the execution changes the internal state. During redundant execution, if one of the web services does not provide the result, the execution of the composition may continue as some component web services have completed the request. However, the state of the failed web service is no longer consistent with the state of the services that completed the execution successfully. This way, it is necessary to study new algorithms and to implement features that guarantee the recovery of the component web services after failures.

This section focuses on the definition of a mechanism that overcomes these problems with exception of the last one. In fact, there are several challenges to overcome when considering the use of stateful services in

systems like FTWS. This class of services is not handled here and this topic remains an open issue for future research work.

The FTWS system uses a simple architecture based on the latest standards and available technologies to build and provide web services. Figure 8.2 presents a detailed view of its internal architecture.

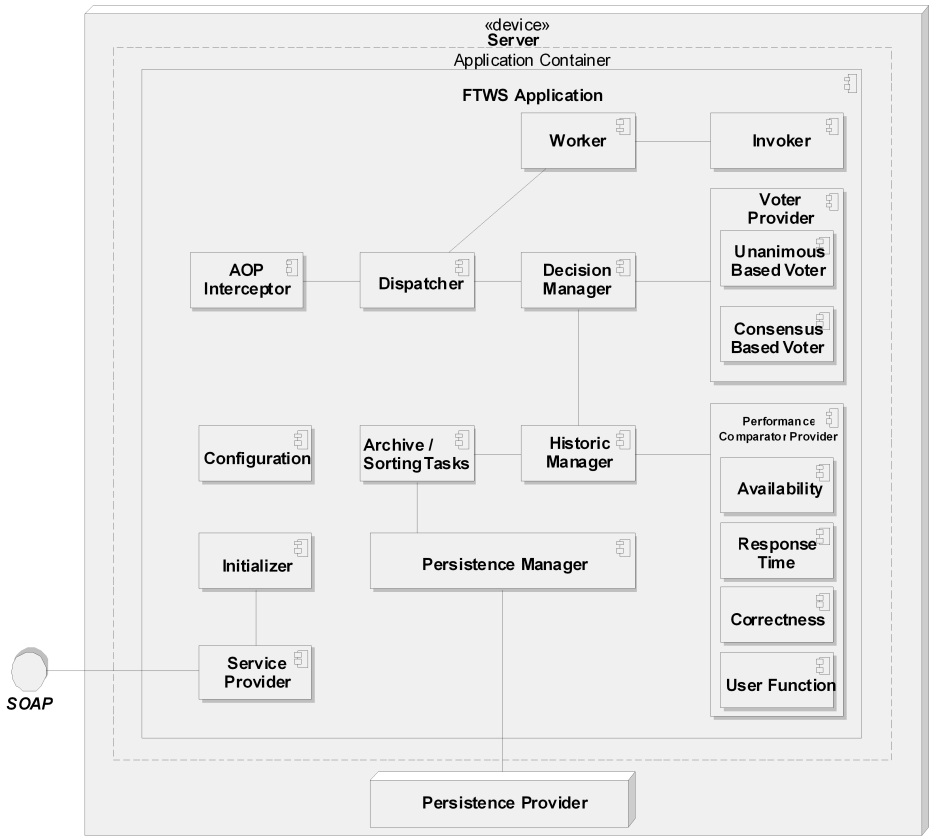


Figure 8.2 – Graphical representation of the FTWS architecture.

FTWS is supported by the following core components:

- **Dispatcher:** Receives and handles a web service request. Detects the operation mode for that particular service request and controls all operational flow;

- **Decision Manager:** Orders one or more configured voters to decide over obtained results and collects the result;
- **Historic Manager:** Collects behavior metrics and maintains results (via persistence tasks) in persistent storage. Provides a link for the Decision Manager to decide upon multiple criteria (e.g., availability, correctness, etc.).

FTWS uses AOP (Aspect Oriented Programming) to, in a transparent way, intercept each client request, and do the necessary tasks to calculate a response. The intercepted method signature and its arguments are delivered to a *Dispatcher* (see Figure 8.2). This is the core component of the system and is responsible for resolving each request and coordinating the whole replication/adaptation process. The intercepted information consists of a method signature and its arguments, which correspond to our generic proxy service (this is the web service exposed by FTWS that will serve as basis for the interception of service invocation, and which will then trigger all subsequent FTWS procedures). At the dispatcher, the configuration is consulted to see what target services are related to this generic service and what invocation method applies (sequential or parallel). This target information consists of **two method signatures** (per each target service configured) that matches proxy classes previously generated with the *wsimport* tool (see a small description in Section 8.6). After this information has been collected, and for each service to be executed, the generated class that represents our service provider is instantiated by reflection (McCluskey 1998) (using the class name included in the first signature referred in the configuration file), and the service port is obtained by calling the configured method (included in that first signature). Finally, the method defined in the second signature is invoked, which executes, in this way, the desired operation on the target service. This reflection-based mechanism enables the user with a transparent way of invoking services only by providing a few method signatures. The user does not need to write client code, the only important task is configuring FTWS, as the framework handles almost all remaining technical details.

The FTWS solution consists of a lightweight application that can be used without a standard application server. This is an adequate solution if low memory footprint is required, or scarce computational resources are available. Nevertheless, for other cases, the application can also be deployed in an application server in a traditional way. In addition, FTWS

enables the use of services that have different interfaces through the use of convenient adapters and several configurable voting strategies are offered with the possibility of easy extension. As in some cases voting may result in a draw, historical data are kept and used to solve these impasses, which can also be used to define a higher call priority for services that presented better behaviors in the past (in terms of performance, response availability, response correctness, or a composed metric of these three).

Notice however that, in some specific cases transparent fault tolerance may not be the best solution. For instance, in legacy or complex applications, exceptional service behavior may represent useful information for the developer to make some design or optimization decisions. FTWS does not currently support the propagation of target services exceptions to the end client. However, these are logged by the application in order to provide useful information for developers.

8.2 Evaluating alternative web services

The evaluation and selection of alternative web services to improve performance and fault tolerance is a key aspect. In fact, as the web services used in a given composition may be supplied by several different providers and have different characteristics in terms of performance, availability, etc., making the right choices is important to improve the effectiveness of the composition. This way, the FTWS evaluation mechanism allows characterizing each web service using three core metrics and a composed adaptable metric:

- **Response time (RT):** characterizes the average time (in milliseconds) that each component web service needs to execute.
- **Response correctness (RC):** characterizes the accuracy of the responses returned by a web service. Correctness is given as a ratio between the number of times the web service returns a potentially correct answer and the total number of times the web service is executed. Potentially correct responses are identified during the voting process.
- **Response availability (RA):** characterizes the web service availability during operation. The web service is considered available when it is able to respond. For example, a given web service is not available if it is invoked and returns no answer or

returns an error. In this context, service availability is given as a ratio between the number of times the web service returns an answer and the total number of times the web service is executed. This may differ from the real service availability, however it is a client-side view that can be used to characterize the service in a practical terms.

- **Composed metric (CM):** it is defined based on the three core metrics presented above. It consists of the following formula: (CM = RTw * RT + CRw * CR + RAw * RA), where RTw, CRw, and RAw, represent the weight of the respective core metric, as defined by the developer. This metric allows programmers to associate a different importance to each core metric.

The metrics can be collected before (offline) or after (online) deploying FTWS. The offline evaluation is typically performed while developing a new composition. The results can be applied to select the best web services and are used later as the basis for the online evaluation. Obviously offline evaluation is not possible when the evaluation tests change the state of the service (e.g., change the state of the data used) and there is no parallel infrastructure that can be used only for testing purposes. In this case, the programmer must define for all web services the default values for each metric, which in turn will be updated during the service lifetime.

The online evaluation is automatically performed during the utilization of the web services. When a web service is invoked, the FTWS mechanism collects information about the behavior observed (i.e., information regarding availability, correctness, response time). This information is used to keep the characterization metrics updated.

Although our proxy-based approach can give a biased perception of the real Quality of Service (QoS) for each target service (e.g., it does not represent an absolute and possibly accurate measure of the service's real performance at a given moment), this perception represents the quality of the service from the client point-of-view. Obviously, since clients typically do not have access to the target services infrastructure (except when they are also providers), this technique is many times the only possible way to obtain such information.

8.3 Operation modes

FTWS provides the following two modes of runtime operation:

- **Simultaneous invocation of the alternative component web services**: for each given operation all the available web services are executed in simultaneous. The obtained responses are then voted in order to select the final result. Two voting mechanisms are provided: a unanimous voter and a consensus voter. The consensus voter can be complemented with the metrics information mentioned before to overcome impasses.
- **Sequential invocation of the alternative component web services**: available web services are executed sequentially as needed. The mechanism starts by executing one service and waits for the response. If there is no response then another service is executed. Invocations stop when a service returns a result or when all services available have been used (an exception is raised if none of the services returns an answer). Since the mechanism returns at the first available answer, no voting is done. The first response is considered the best and will be the one to return to the client.

In the simultaneous invocation method, several worker threads are created to invoke the corresponding target services. The dispatcher (see Figure 8.2) waits for worker thread termination or timeout before proceeding to response analysis and delivery. If the developer does not define a timeout, the application assumes a default value for all services. If history on response times is available, the application can be configured to use it to calculate the timeout value, which is typically equal to the maximum response time observed excluding the potential outliers. For this we consider that, for a given service, response times that diverge from the average more than N times the standard deviations are outliers (N is defined by the FTWS user). A keep value can be added to the maximum response time observed (excluding outliers) in order to reduce timeouts.

In the sequential invocation mode, the web services execution sequence is defined based on the historical information available for each service. The services are ranked based on the evaluation metrics (see Section 8.2) as selected by the programmer. For example, the programmer may specify that the web services should be ranked using the response correctness

metric, the response time metric, the availability metric, or the composed metric (which allows a more elaborated ranking approach).

The sequential invocation mode is quite useful when the cost of the services is an issue. It allows a cost reduction by executing the minimum number of services needed to obtain a response. Obviously it must be applied only when the services being used provide high correctness. When a service does not provide high correctness values and is the first to be consulted, there may be a high probability of returning an incorrect answer. This highlights the importance of choosing an appropriate sorting method. Sorting services, for instance, by response time can be a good practice if the fastest services also present high correctness values (or when wrong answers are acceptable). In other cases, the sorting method will render our composition useless. All the available metrics can be initially collected during offline evaluation, and this preliminary evaluation can be used as basis for choosing the adequate sorting method.

In practical terms, during execution a list of target services is consulted. This list is sorted from the best to the worst service and is updated continuously. This means that each time a service is executed and its score changes the corresponding list may be reordered. As this may be quite expensive in computational terms, list sorting may be performed periodically in configurable time intervals. To reduce input/output operations, service score information is stored in memory during the application's life cycle. This information is written to secondary memory on demand or when service scores change (configured by modifying an attribute in the configuration file).

Our mechanism allows periodic switching between the two modes. This switching feature is important to maintain the metrics up-to-date. An aspect that needs to be considered is that, in the sequential operation mode there are services that are only used if other services fail. This may lead to the aging of the metrics for the services that are not used frequently. Thus, providers must periodically move to simultaneous invocation mode in order to keep the metrics up-to-date.

As we can see, these operation modes complicate the use of such a system, when having stateful target web services. In fact, it may not be useful to use such a system when considering the sequential invocation mode. All services not used in a given invocation would have to be updated some time later, i.e., we would be using, in fact, all of the available services, and this would no longer be a low cost operation mode. However, it would

remain an interesting option when one is interested in obtaining an answer from the service that is historically better (regardless of the chosen criteria). Considering the simultaneous invocation of stateful services, such a system would have to guarantee that all services are updated, even in the presence of faults, which is an issue that has no trivial answer. As mentioned before, stateful service usage is out of the scope of this mechanism and remains an open issue for future research.

8.4 Adapting web services invocation

An important characteristic of the FTWS mechanism is that it can use component web services that have different interfaces, providing developers with more options to build their solutions.

The mechanism uses an Adapter Design Pattern inspired implementation (Gamma et al. 1994) to provide easy uniform access to different target services. It does so by transparently providing a generic and standards compliant web service as an interface to multiple target services. These target web services may have been built for distinct purposes, to a certain extent, but can still be used in an aggregated generic solution (provided the semantic meanings of their input and output messages are similar). An example of this can be seen in the generic Zip Service (used in Section 8.6.2), created to return geographic details for a provided zip code. In fact, one of the target web services used is a weather condition reporting service, but it includes in its answer the required information for the Zip Service.

The mechanism does automatic parameter copy, based in a parameter map provided in the configuration file (see Figure 8.3), and allows the use of more complex transformation logic using custom adapters. For example, a converter may be used simply to convert a word to a Boolean value or an adapter may be included to extract a date from a string. An important aspect is that web services responses may also have different formats. Thus, adapters can also be configured to conciliate responses into a standard format.

FTWS provides a set of standard converters (available at (Laranjeiro 2011)) and allows programmers to define adapters that perform more complex logic that may be required to convert specific responses. The current version of FTWS supports primitive data types and complex data

types in input parameters and response parameters in all possible combinations, but nested complex types are not supported by the current version.

To illustrate the adaptation process, consider the case where we have 4 services that accept a temperature value as input. Three of these accept values in Celsius and the remaining service accepts values in Fahrenheit. If we wanted to model this with FTWS it would be necessary to create a generic service that accepted a temperature value, but it would also be necessary to define one Celsius to Fahrenheit converter (the other way around would be of a computationally higher cost). So, the tasks are: 1) map possible non matching parameter names (between our generic service and each target service) in the configuration file; 2) code the converter; and 3) reference the converter in the configuration file.

Figure 8.3 presents a partial example of a configuration file. The parameter *celsiusTemp* in our generic web service maps to the *FahrenheitTemp* parameter in the target service (one map must be built per target service if any kind of conversion is needed). Furthermore, we are indicating that class *C2FConverter* is the class to be used for parameter conversion in every invocation (FTWS knows which converter method to invoke as the class implements *IConverter*, a well known interface).

```
...
<genericServiceInputMap>
    <entry>
        <string>celsiusTemp</string>
        <fieldAndConverter>
            <fieldName>fahrenheitTemp</fieldName>
            <converterName>pt.uc.dei.ftws.
                 converter.C2FConverter</converterName>
        </fieldAndConverter>
    </entry>
  ...
</genericServiceInputMap>
...
```

Figure 8.3 – An example of adapter configuration in FTWS.

Reflection is used to create the converter object and to invoke its well-known conversion method at runtime. The temperature value will be

transparently converted from Celsius to Fahrenheit by FTWS and this is the value that will be used as *fahrenheitTemp* to invoke the target web service. Obviously, a similar approach may be necessary to map and/or convert the target service's response.

8.5 Selecting web services responses

When a request dispatcher determines that all (or the maximum possible) answers were collected, it transfers them to a decision manager (see Figure 8.2). This will be responsible for coordinating a voting process that will determine the best of the answers obtained.

The decision manager has access to a voter list (represented by the *Voter Provider* component in Figure 8.2). This list is defined and sorted by the application user and includes two predefined voters. New voters can be added. The standard voters include a unanimous based voter and a consensus based voter. The former votes each answer as correct if all of the collected answers are consistent. The latter marks a response as correct if a majority of consistent answers exists. The concrete process for this voter includes the evaluation of a result list of converted target responses.

In a first step, each response in the list is associated with a value that represents the number of answers that are equal to it. A relevant fact is that this 'message' comparison is not done directly over the SOAP messages, as that would be an extremely difficult and error-prone process. Instead, the comparison is done at object level (i.e., programming language level). For example, in Java this implies that complex object types implement the 'equals' method or, in alternative, the 'Comparable' Java interface (otherwise the comparison process would become more difficult). Since we are dealing with our own generic answers (i.e., we are not comparing each of the target service's specific answers, but our already converted answers), there is no need to apply difficult comparison logic over different objects; this comparison logic is embedded once in our generic response object. After this first step, the maximum value obtained is stored. Then, all of the answers that have a cardinality value equal to the maximum are marked as correct. At the end of the process the voter returns a value that enables the decision manager to know that no correct response was found, a single correct response was found, or multiple correct responses were found. Figure 8.4 illustrates the basic algorithm (in source code form) of the unanimous based voter provided with FTWS.

```
public class UnanimousBasedVoter implements IVoter
{
    public VotingDecision vote(List<WsResultWrapper> resultList)
    {
        VotingDecision result = VotingDecision.ALL_FALSE_ANSWERS;
        int correctElements = 0;

        for (WsResultWrapper wrapper : resultList)
        {
            int cardinality = 0;

            for (WsResultWrapper toCompare : resultList)
            {
                if (wrapper.equals(toCompare))
                {
                    cardinality++;
                }
            }

            if (cardinality == resultList.size())
            {
                wrapper.getServiceScore().setCorrect(true);
                correctElements++;
            }
        }

        if (correctElements == resultList.size())
        {
            result = VotingDecision.UNIQUE_CORRECT_ANSWER;
        }

        return result;
    }
}
```

Figure 8.4 – The main algorithm used by the unanimous based voter of FTWS.

At the decision manager, if only one response was marked as correct, this is the response to return to the dispatcher and subsequently to the client. For the other two cases we have a tie and if there are no more voters to break the tie, it is necessary to consult historical information.

If multiple different answers were marked as correct, these are separated from the wrong answers and the historic information is searched using the service identifier of each of the answers. As the service list is, in principle, ordered using the configured criteria, the best services will be first in the list. This means that we can find the best service (for the chosen

criteria) searching the service list from the beginning; the best will be the first to be found also in the list of potentially correct answers (the answer object is a wrapper of a service response that includes a unique service identifier). If no potentially correct response was found, history is searched in an analogous way, but in this case, the first service identifier found in the service list that is also contained in the list of all obtained answers, will be selected (opposing to considering only the correct answers).

Besides specifying one preferred criteria, the programmer may also specify that impasses should be solved, for example, using first the response correctness metric, then the availability metric, and finally the response time metric. This way, if an impasse occurs, the response selected is the one provided by the set of web services that present the highest correctness ratio. If the impasse still holds, then the response selected is the one provided by the set of web services that present the highest correctness and availability. And so on.

Note that this represents a best effort strategy whose effectiveness is dependent on the metric used. Obviously it should be applied only when the services being used provide high correctness (which can be observed during offline evaluation).

8.6 FTWS in practice

This section describes a practical usage of the FTWS prototype developed in Java. An experimental evaluation is conducted to demonstrate the capabilities of the mechanism using a set of services specified by the TPC-App benchmark and a set of web services publicly available on the Internet.

8.6.1 Develop fault tolerant web services

As mentioned before, one of the goals of FTWS is to reduce the developer programming effort to simple configuration tasks. In this section we briefly show how web services programmers can use our prototype of FTWS. The basic system requirements to build a FTWS project (using the current version) are Java 6 SE (Oracle 2011c) and Maven 2 (Apache Software Foundation 2008d). The following steps must be followed:

1) Create a Java project in any Integrated Development Environment .

2) Generate the component web services artifacts, including the local proxy classes for the remote services. A potential tool to support this task is *wsimport*. This is a tool that is included in the latest Java Development Kits and is regularly used for client code generation from each service's WSDL. Keep in mind that this is a task that is always executed by web service developers, in normal circumstances, whenever the generation of a web service client is necessary.

3) Create an FTWS configuration file. This is an XML file whose configuration includes performing the following tasks:

a) Define the name of each proxy service signature. This is the method signature of the new service that will accept client requests and manage the redundant invocations and voting (code for this service is built automatically).

b) For each component service define the method signature of the service's provider (obtained from the generated artifacts classes) and a method signature for the component web service invoker (also in the generated classes).

c) Define a mapping between the proxy service's input parameters and the component service's input parameters. Optionally define the name of any specific converters needed. Figure 8.3 exemplifies a mapping between two different services (only one parameter is considered). The parameter 'celsiusTemp' on our generic service maps to the 'fahrenheitTemp' on our target service. 'pt.uc.dei.ftws.converter.C2FConverter' is the name of the class that will be used to convert values between these services.

d) Define a field mapping between each component service's output fields and the proxy service's output fields. Again a parameter converter can be defined as in the previous step.

4) Define some properties for the FTWS application in Maven's pom.xml. Table 8.I presents the most relevant parameters.

5) Build the project using Maven's default packaging method.

Table 8.I – Main parameters for the FTWS configuration.

Parameter	Description
delegate	The name of the package where the proxy services classes are included
comparator	Specifies the metrics that should be used to solve voting impasses (i.e., response time, availability, correctness, or other function specified by the programmer). To use a function specified by the programmer the relative weights of each base metric should be defined (see Section 8.2)
invokeAll	Specifies the operation mode. If true, the alternative component web services are invoked simultaneously. If false, then sequential invocation is used
archiveOnDemand	Boolean property that specifies if historical data should be stored in persistent memory. For higher performance it should be defined as false. For higher reliability it should be defined as true (data is changed when scores change)
historyWindowSize	Number of items from the recent historical data to be used in the metrics calculation. Zero means that all available data should be used

8.6.2 Experimental evaluation

This section presents a set of experiments that was conducted to demonstrate FTWS and analyze two important aspects: the average impact on web services performance and the fault tolerance gain in a typical usage scenario.

For the performance impact analysis, a 30 minutes load test was executed using a composition (not using FTWS) based on a serial invocation of a set of web services specified by the TPC-App performance benchmark (Transaction Processing Performance Council 2008). In particular, this composition was built using the services NewCustomer, ChangePayment, NewProducts, and ProductDetail (in this order), and was created with the help of soapUI (Eviware 2011). There was no use of specific business process engines; instead, the simple testing facilities provided by soapUI were sufficient to create the desired testing environment. After this first test, a second 30 minutes test was done, this time with our application as a proxy for the TPC-App services.

The average impact on the overall performance for the FTWS application under high stress conditions was of 1090.63 ms with a standard deviation of 117.50 ms. Although this is not a low impact, it is important to note these environments are characterized by large execution times which mitigate the relative impact in the performance (Bovy et al. 2002). Nevertheless, as it is simply a technological aspect, an effort can be undertaken to minimize this impact in the future, via some code refactoring and design optimizations.

The fault tolerance test scenario included the creation of three compositions using FTWS. Seven component web services were used: three to obtain a country list service, two to validate the format of e-mail addresses, and two to obtain Zip code geographic details, all referenced in (XMethods.net 2008) and publicly available on the Internet. For experimental purposes a service was considered unavailable whenever it took more than 60 seconds to respond to a client request. This is directly related to the availability results presented.

Table 8.II presents the initial evaluation of the web services in terms of the core metrics considered in this work. Afterwards, both FTWS operation modes were tested (Table 8.III). In this way 4 sets of tests were executed: one for the simultaneous invocation operation mode (referenced as the 'All' criteria in Table 8.III) and the remaining three for the sequential invocation operation mode. A different criteria was used (response time, availability and correctness) for each of these three tests. Results for the correctness metric are not presented, as we did not have information regarding wrong answers returned by the web services during the test phase. However, this metric may prove its usefulness in services not included in the tested set. Additionally, these particular results do not weaken the utility of the simultaneous invocation mode. This mode continues to be useful even if all services are correct. For instance, when using Internet based services, it is frequent to have large fluctuations in response time or availability. In these cases, if the user is interested in the fastest services, this will be the method that assures that the client always gets the fastest answer (comparing to sequential invocation and assuming that there is a large variation in historic response time values). Similarly, if the user is more interested in a highly available service, he will have an answer without having to potentially wait for a sequential invocation of N services.

Table 8.II and Table 8.III present the result of the tests; these had a duration of 2 hours and the web services were invoked with a periodicity

of 60 seconds (with a uniform variation of 30 seconds). Each test was executed three times in order to increase the results representativeness.

Table 8.II – Baseline performance results for the used services.

Web Service	Response Time (ms)	Standard Deviation	Availability
Country (1)	1509.71	1901.77	95.94%
Country (2)	1303.60	1461.76	97.48%
Country (3)	826.62	994.66	93.90%
E-Mail (1)	2483.42	46.77	98.20%
E-Mail (2)	2033.23	3387.68	92.37%
Zip (1)	858.63	27.48	99.30%
Zip (2)	402.24	754.17	99.66%

Table 8.III – FTWS services results.

Criteria	Web Service	Response Time (ms)	Standard deviation	Detected failures[3]	Availability
All	Country	3762.27	4668.97	2.21%	100.00%
	E-mail	3211.32	4900.63	4.09%	100.00%
	Zip	1295.06	1303.02	0.70%	100.00%
Response Time	Country	1929.27	2950.43	1.60%	100.00%
	E-mail	1879.57	1644.48	0.00%	100.00%
	Zip	1325.28	4371.35	0,53%	100.00%
Avail-ability	Country	1852.88	2240.58	0.00%	100.00%
	E-mail	2154.21	5076.96	1.42%	100.00%
	Zip	1039.57	3564.50	0.35%	100.00%

As we can see from the analysis of Table 8.II and Table 8.III, choosing the FTWS simultaneous invocation operation mode increases the average response time, as the application is always dependent on the worst performer. Note that, regardless of the high response times, the availability also increases to 100% even in presence of detected remote service failures, emphasizing in this way the usefulness of the FTWS

[3] Represents the individual component services that failed and not FTWS failures.

mechanism. The high standard deviation values observed are related to the fact that, although the majority of the requests had a response time that was close to the average, occasionally a few requests took much longer than the average (a typical behavior when using services over the Internet), largely increasing the standard deviation values. Additionally, as mentioned before, these results were extracted from preliminary testing that does not have a high enough duration to be statistically tested. However, they completely serve the purpose of demonstrating the utility and performance of FTWS in more qualitative terms.

Considering the sequential invocation mode (shown in the rows Response Time and Availability criteria in Table 8.III), we can see that the response times have decreased (compared to the simultaneous invocation) for all existing criteria (we empirically noticed a relation between the fastest and most available services). For this operation mode we can also see that the percentage of failures is lower, when compared to the simultaneous invocation method. This is a potential indicator that the correct metric has been used to rank the services.

8.7 Conclusion

A mechanism (FTWS) that allows programmers to easily develop fault tolerant compositions using diverse web services was presented in this chapter. It provides a set of artifacts that simplify the coding process, deals with all the aspects related to the redundant web services invocation and responses voting and is able to perform a continuous evaluation of the services.

FTWS uses well-known techniques (redundancy, voting, etc.) to solve a set of problems typically faced by developers that need to implement fault tolerant services. It consists of a lightweight application that can be used without a standard application server. This is an excellent solution if low memory footprint is required, or scarce computational resources are available; however, for other cases, the application can still be deployed in a full application server in a traditional way.

Results indicate that the use of FTWS is beneficial when the goal is to provide highly available services. The features provided by FTWS also make it an excellent option when the goal is to obtain correct responses, or to select the fastest service. Despite this, the mechanism does not yet

provide ways to maintain consistency in stateful web services. This is in fact an issue in web services whose execution changes the internal state. Indeed, if one of the alternative web services does not provide a result, the state of the failed web service is no longer consistent with the state of services that completed the execution successfully. This way, we need to study new approaches that guarantee the recovery of the component web services after failures.

Despite the utility of FTWS, we can understand that the use of multiple software services to obtain a single response may certainly not be the best option in some scenarios. In fact, a multiple invocation scheme may not be useful when the use of extra services is limited (for instance, when there are bandwidth limitations or payment is required to use the services). In addition, in business critical environments time is an issue, so there is no point of invoking a service (or a set of services) if the client can know beforehand that a response will not arrive on due time. The next chapter discusses the creation of a mechanism that handles precisely this issue, and provides timing failures detection and prediction features to services, enabling clients to use alternative services and preventing the unnecessary execution of an operation (or allowing its execution in a lower priority).

Chapter 9
Timing Failures Detection and Prediction in Web Services

This chapter addresses the problem of handling timing requirements in web services and discusses the design of a mechanism that can endow services with timing failures detection and prediction capabilities. Such mechanism can be extremely useful for service clients to express timing requirements and also to choose a service that can comply with those requirements. Furthermore, it can also be beneficial for developers as creating services that are able to deal with timing constraints can be a difficult and time consuming task.

The web services technology, programming models, and development tools do not provide easy support for assuring timeliness properties during web services execution. Although some transactional models provide basic support for detecting the cases when operations take longer than the expected/desired time (Elmagarmid 1992), this usually requires a

high development effort. In fact, developers have to select the most adequate middleware (including a transaction manager that must fit the deployment environment requirements), produce additional code to manage the transactions, specify their properties, and implement the basic support for timing requirements. Transactions are actually well suited for supporting typical transactional behavior, but they are inadequate for deploying simple time-aware services.

Despite the lack of mechanisms and tools for building time-aware web services, the number of real applications that have to support this kind of requirements is getting more frequent, as time is a critical issue in business/services environments (Menasce 2002). Typically, developers deal with these requirements by implementing ad-hoc solutions to support timing failures (this is, obviously, expensive and prone to fail). The concept of time has been, in fact, completely absent from the standard web services programming environment. Important features such as timing failure detection and forecasting have been overlooked, although these are particularly important if we consider that web services are typically deployed over wide-area or open environments that exhibit poor baseline synchrony and reliability properties.

In this chapter we discuss the problem of handling timing requirements in web services. Besides defining the concepts, we present an extensible framework that offers detection and prediction functionalities (wsTFDP). This framework provides a ready to use timing failure detection mechanism that is based on developer-transparent code instrumentation and is also able to collect historical data that can be used for prediction. Thus, the framework provides a prediction component that implements the Dijkstra's shortest path computation algorithm (Dijkstra 1959), which uses historical execution time values to predict if execution can terminate on time. This framework can be easily extended with multiple components that implement different prediction algorithms. By using wsTFDP, developers are able to easily plug in their preferred prediction algorithm and fine-tune it in the way that best fits the deployment environment. Services that use this model enable clients to express their timeliness requirements for each service invocation by defining a timeout value and an associated confidence value for prediction. When timing requirements are not possible to satisfy (e.g., because the deadline was exceeded or because it will predictably be exceeded) the server responds with a well-known and consistent exceptional behavior.

The outline of this chapter is as follows. Section 9.1 presents an overview of the requirements and architecture of wsTFDP. Section 9.2 focuses on the detection mechanism, whereas Section 9.3 details the prediction mechanism. Section 9.4 describes the mechanism from a practical point-of-view and presents an experimental evaluation executed to assess its effectiveness. Section 9.5 concludes this chapter.

9.1 Detection and prediction mechanism overview

As mentioned previously, in web service environments, it is normal to observe services that exhibit high or highly variable execution times. High execution times are usually associated with the serialization process involved in each invocation, coupled with a great amount of protocol information that has to be transmitted per each payload byte (e.g., the SOAP protocol requires a large amount of data to encapsulate the useful data to be transmitted). This serialization process is particularly important since a given web service can also behave as a client of another service, thus duplicating the end-to-end serialization effort. Variable execution times are essentially related to the use of unreliable, sometimes slow, transport channels (i.e., the Internet) for client-server and inter web services communication. These characteristics make it difficult for developers to deal with timing requirements.

Two outcomes are possible when considering timing requirements during a web service execution: either the server is able to produce an answer on due time, or not. The problem is that in both cases the client application has to wait for the execution to complete or for the deadline to be violated (in this case a timing failure detection mechanism must be implemented). However, in many situations it is possible to predict the occurrence of timing failures in advance. In fact, execution history can typically be used to confidently forecast if a timely response will be possible to obtain. Note that, this is extremely important for client applications, that can retry or use alternative services, but also for servers, that can use this information to conveniently manage the resources allocated to each operation (e.g., an operation that is predictably useless can be canceled or proceed executing under a degraded mode). In addition, such mechanism can be integrated in the FTWS mechanism presented in the previous chapter in a general solution that also provides tolerance to timing failures.

To provide client applications with the possibility of invoking services in a timely manner we need a mechanism that is able to abort, not execute, or gracefully degrade the execution of operations that are unable to match the client's requirements. Besides failure detection, wsTFDP enables us to collect accurate runtime failure data that is later used for predicting failures. To be useful in real scenarios, a timing failures detection and prediction mechanism must achieve a key set of quality attributes. Thus, the mechanism must fulfill the following objectives:

- **Effective:** it must provide low detection latency (i.e., it must be able to detect failures on due time) and must achieve a very low number of false-positives (i.e., the mechanism should be able to detect all failures that are indeed temporal failures). Our implementation goal, defined based in empirical knowledge of web service-based environments, was to keep the detection latency below 100ms and the detection false-positives rate under 5%.
- **High prediction accuracy:** the number of false-positives must be very low. A low false-positive rate means that, in few cases, the mechanism predicts a timing failure that, in reality, will not occur. On the other hand, the number of false-negatives (i.e., failures that are not predicted) must also be kept low. A system that fails less in identifying or predicting failures is obviously preferable. Based on our experimental knowledge, we aimed to provide an integrated mechanism able to maintain at least one of the rates under an average of 5%. We opted to define a single limit as lowering one of these two measures typically results in increasing the other.
- **Low performance overhead:** a time-aware service obviously performs more computational work than a basic service. For our prototype, we aimed to achieve a maximum overhead of 100ms in terms of response time (when compared to the equivalent service without timing characteristics).
- **Easy usage:** code, tools, or deployment complexity and non-portability are among some of the characteristics developers try to avoid nowadays. It is important that developers, either consumers or providers, do not have to make significant changes to existing code in order to use the mechanism. In the same way, when developing new services, the development model must be maintained as similar as possible to the common practice.

- **Generic:** the mechanism must be generic so that it can be reused, as much as possible, in different environments. Our goal is to provide extension features so that wsTFPD can be used outside the scope of web service applications, e.g., in Remote Method Invocation methods (RMI).
- **Extensible:** a major goal is to provide extension features so that a developer can plug in additional prediction modules that implement more accurate prediction algorithms.

The wsTFDP framework (see Figure 9.1) is a server-side mechanism that, at runtime (and without extra coding effort) transparently detects timing failures, collects performance data, and predicts at specific points if a service invocation can conclude on time. A prototype of the wsTFDP mechanism is currently available in Java (a different language could have also been used) and uses Aspect Oriented Programming (AOP) (Kiczales et al. 1997) to instrument web services' bytecode in a completely transparent way to the developer. All logic aspects related to the timing failure detection and prediction are in fact an isolated package that can be merged into any application by using AspectJ. This process consists in compiling the candidate application and the wsTFDP component (using the AspectJ weaver) into a single Java application that is wsTFDP-ready. All wsTFDP logic is injected at special points (described further ahead) in the target application. Using this setup it is possible to define a single detection mechanism to be used in multiple different web service operations in the context of a given application.

wsTFDP uses an around advice (so that we have control before and after our join point) and a pointcut that matches all methods that are annotated with *@WebMethod* and take a *TimeRestriction* object as parameter. This object is defined in the context of the wsTFDP mechanism and is the one used by clients to specify timeout values (see Section 9.4 for details on how to use it). With this configuration we are able to intercept the web service calls for which we want to perform timing failure detection. At the moment of interception several actions are taken by our framework in order to determine if the execution is on time or if a timing violation has or will predictably occur. The whole process is summarized in Figure 9.1 and described in the following paragraphs.

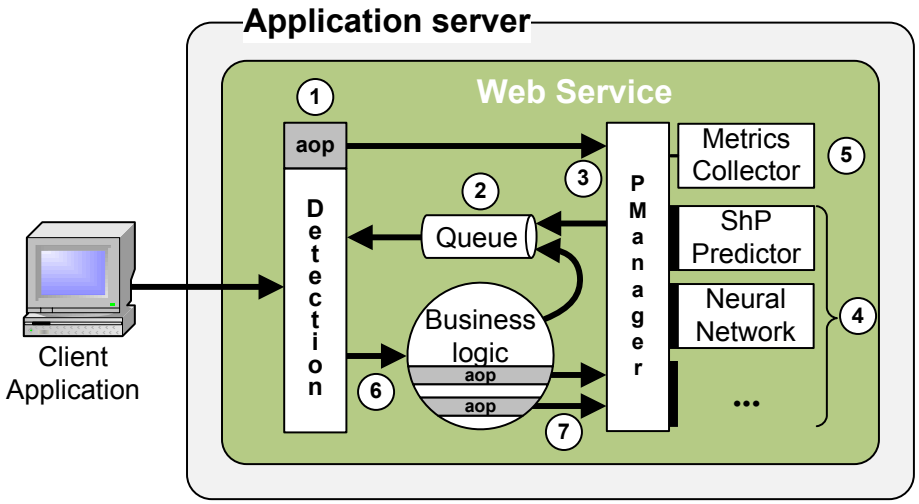


Figure 9.1 - wsTFDP extensible architecture.

Incoming client requests must include a desired maximum service execution time and a confidence value for prediction. Such requests are handled as follows: the service's entry point is instrumented to start a detection component **(1)** that waits a response over a blocking queue **(2)**. Control is then transferred to a prediction manager component **(3),** which executes the configured prediction module **(4)** over a set of historical metrics (metrics are collected and managed by a metrics collector **(5)**) and returns a predictable execution time for the client request. Based on this information, the prediction manager decides if the application's business logic should be executed, or if, considering the client's expected deadline it is probable that a timing violation will occur (according to the historical data and to the confidence value defined by the client). This decision essentially results in two outcomes: if a timing violation is predicted then a well-known exception is placed in the blocking queue, otherwise the execution of the service's business logic is started. In both cases an object will be placed in the blocking queue, and a signal will be delivered to the detection component (which is blocked over the queue). At that point, the detection component retrieves the object placed in the queue and delivers the result to the client application (see Section 9.2 for details).

Service applications may include requests to other time-demanding services, such as external web services (e.g., payment gateways, business partners, etc.). So, when service execution is allowed **(6),** our framework

transparently intersects those types of calls **(7)** to predict, based on the client-set values and the available remaining time (and before allowing execution to proceed), the probability of occurrence of a timing failure during the external service execution. This process is equal to the one applied at the service's entry point. Keep in mind that, at any time, the detection module may deliver a 'TimeExceeded' exception to the client if it detects that the desired time has expired.

An important aspect is that the framework's architecture enables connecting more effective, or faster, predicting modules as desired by the developers. For instance, it is possible to plug in one of Weka's (Frank et al. 2010) multiple data analysis algorithms. Currently, the general procedure consists of adding the predictor's jar file to the project and configuring the predictor module in wsTFDP project's descriptor. At each prediction point the available historical data is used in a similar manner, as in the case of the default Dijkstra's shortest path (ShP) module implemented by the ShP predictor. The following subsections detail the detection and prediction modules.

9.2 Detecting timing failures

Figure 9.2 illustrates the internal design of wsTFD and the sequence of events that occur at the time of interception of a web service call. The horizontal solid lines represent a thread that is in a runnable state and is performing actual work. Dashed lines represent a thread that is waiting for some event.

At interception time each container thread (parent), that is responsible for serving a particular client request, spawns a new thread (child). This spawned thread is now responsible for doing the actual web service work and placing the final result in its blocking queue. Immediately after the child thread is started, the parent tries to retrieve the result from the child's blocking queue. As at that starting point no result is available, the parent thread waits during a given time frame (timeout specified by the client application) for an element to become available on the queue. During this period there is no periodic polling of the blocking queue, which reduces the impact of the mechanism. Instead, the parent waits for the occurrence of one of the following events:

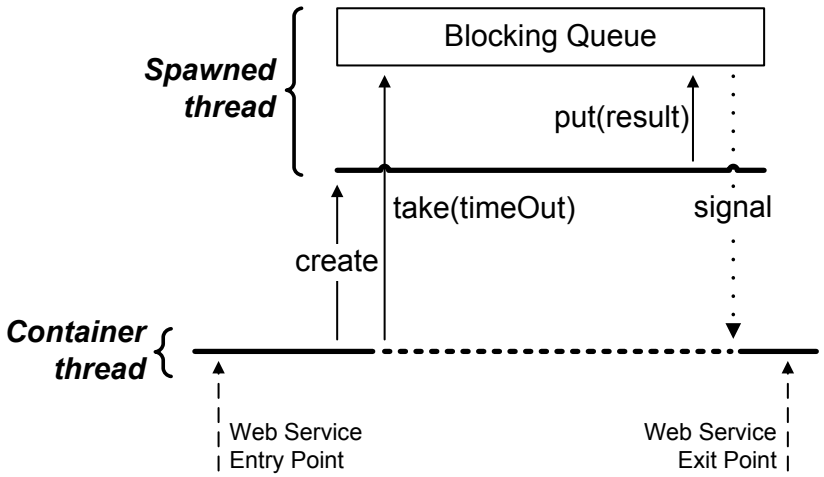


Figure 9.2 – Temporal failure detection mechanism.

- A signal to proceed with object removal, which occurs when a *put* operation is executed over the blocking queue. This operation signals any waiting thread (on that queue) to immediately stop waiting and proceed with object removal (this object corresponds to the result of the web service execution).
- The waiting time is exhausted. After the time has expired, the parent thread continues its execution (i.e., leaves the waiting state), ignoring any possible later results placed in the queue. In this case, an exception signaling the occurrence of the timing failure will be thrown to the client application.

There are two types of results that can be expected from the child thread's execution. The first is a regular result (i.e., the one that is defined as the return parameter in the method signature) and the second an exception being thrown (it can be a checked or unchecked exception). In both cases, an object is placed in the waiting queue (exceptions are caught by the child thread and also put in the queue as regular objects). When the parent retrieves an object from the queue, type verification is performed. For a regular object, execution proceeds and the result will be later delivered to the client. On the other hand, if the retrieved object is an instance of Exception, the parent thread re-throws it, which enables us to maintain program correctness (that would be lost if the child thread did throw the exception itself).

For notifying deadline violations, wsTFD offers two types of exceptions (it is the responsibility of the provider to choose the exception type that better fits his business model):

- **TimeExceededException**: this is a checked exception, i.e., if the provider decides to mark a given public method with a 'throws' clause, then the client will be forced to handle a possible exception at compile time (it will have to enclose the service invocation with a try/catch block).
- **TimeExceededRuntimeException**: this is an unchecked exception, i.e., the client does not need to explicitly handle a possible exception that may result from invoking a particular web service operation.

9.3 Prediction mechanism design

In order to predict timing failures, wsTFDP executes the following set of tasks (detailed in the next subsections):

- Analyzes the service and builds a graph to represent its logical structure;
- Gathers time-related performance metrics in a transparent way during runtime;
- Uses historical data to predict, with a degree of confidence chosen by the client, if a given execution will or will not conclude on due time.

9.3.1 Describing the web services structure

wsTFDP analyzes the logical structure of web services and automatically **organizes a graph structure** for each operation provided to clients. A graph consists of vertexes (or nodes) and edges that connect nodes. Each edge represents a connection between two nodes (i.e., an edge provides a path between two nodes, which in our case is unidirectional) that has an associated cost (i.e., travelling between two nodes involves a predefined cost) (Dijkstra 1959). This data structure perfectly fits the requirements of

our mechanism. To build the graph we have to define what information will constitute the nodes, edges, and costs:

- **Nodes**: specific instants in a service's execution where timing failures should be predicted. Natural candidates are the invocation of the target service itself and all nested service invocations (if any) performed by the service. Additionally, it is important to add support for user-identified critical points that allow the developer to instruct wsTFDP to add specific code parts to the graph.
- **Edges:** these naturally represent the available connections between nodes, and are automatically defined by wsFTP based on a runtime analysis of each service.
- **Cost:** for our goals, the cost involved in travelling between two nodes is the execution time in milliseconds.

Figure 9.3 presents an example of a web service augmented to be time-aware and the respective graph organization (source code in bold is specific to our framework). The arrows connecting the nodes indicate unidirectional relations between those nodes. That is, it is possible to go from the 'Service C' node to the 'DB query' (database query) node but not the other way around, which of course accurately represents what the programmer intended when writing this particular piece of source code. This service uses two wsTFDP methods: *check* and *ignore*. The *check* method indicates that that specific point in the code is critical and must be included as a node on the graph and a point for timing failures prediction. The *ignore* method indicates that the following web service call should not be considered as a node when building the graph and also when predicting timing failures. It is the responsibility of the programmer to identify critical execution points that should take part of the runtime graph. This task can be easily achieved with a regular profiling tool like VisualVM or JRockit (Oracle 2012; Hirt 2008). For example, the invocation of 'Service D' (represented by the *invokeServiceD();* statement) is not included in the graph as it was explicitly ignored by the programmer.

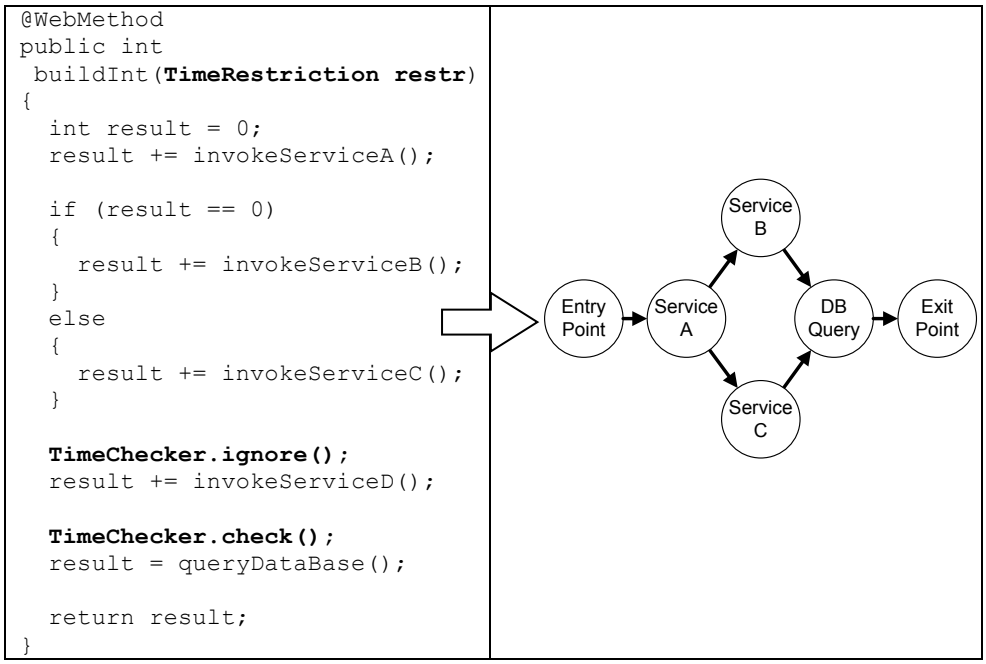


Figure 9.3 – Transformation from source code into a graph structure by wsTFDP.

The graph structure is organized during regular service execution. Instead of implementing a fairly complicated approach for compile-time path analysis, we use AspectJ (Eclipse Foundation 2008) during service execution. For example, in Java a web service can be invoked by executing a generated method that is marked with the *@WebMethod* JAX-WS annotation (Sun Microsystems Inc. 2010). In order to build the graph, wsTFDP assumes that the provider is using any implementation form of JAX-WS, which is currently the standard (e.g., JBoss provides a JAX-WS compliant implementation, such as most major middleware vendors). In our case, AspectJ is instructed to intercept calls to *@WebMethod* annotated methods and uses this interception information to build the service graph. As mentioned before, wsTFDP intercepts all external service calls (i.e., nested calls) by default. However, the mechanism can also be configured to intercept only specific (i.e., marked by the developer) external service invocations, ignoring the remaining.

9.3.2 Metrics Management

Time-related metrics are collected by the detection component and stored in the graph's edges. In each run, wsTFDP stores information that expresses how long it takes to travel from a preceding node to the current node (in terms of execution time). In practice, each edge maintains a list of historical values sorted from the shortest to the longest time. This list is updated in each web service call with the observed value. Obviously, as the maximum size of this list must be limited, the provider has to configure it according to the specificities of the web services being monitored. In environments that experience small variations in execution time a small list may be sufficient, while other type of environments may require larger lists that can capture the natural execution time variation of the web service. When considering the mechanism's performance smaller lists are preferable, since the sorting algorithm, required by our pessimistic approach for failure prediction, will obviously perform faster in smaller than in larger lists (see next section for details on the prediction process).

An important aspect is that the graph will be quite incomplete during the first executions of the web service. Typically, web services can be rather complex and several calls may be needed to explore all possible paths, and hence build a complete graph. The more paths are explored, the more accurate prediction becomes.

9.3.3 Prediction Process

At each node wsTFDP subtracts the elapsed time (counted from the moment execution started) from the maximum amount of time specified by the client application (which represents the total available time for execution). It then uses the Dijkstra's shortest path computation algorithm (Dijkstra 1959), which makes use of the best (fastest) time values stored on the graph's edges, to determine if there is any chance of the execution to terminate on time. If history indicates that the amount of time left in a given service invocation is not enough to complete execution in a timely fashion (independently of the code execution path followed), then it is fairly safe to return a well-known prediction exception to the client.

When a timing failure is predicted, two outcomes are possible: the server aborts the operation or the server continues the operation under a

degraded mode. The former should be used in the cases where it is safe to artificially abort a particular operation. The latter mode is useful for those cases where consistency is needed on the server side (i.e., an initiated operation must not be artificially aborted, as the application may not be prepared to deal with this situation). These two modes are particularly important to providers since they both ease the load experienced by servers when timing failures occur. Obviously, it is up to the provider to decide which mode better fits the application being deployed.

An important aspect is that there is always some degree of confidence involved in a prediction process. In wsTFDP, this degree of confidence can be manipulated by the provider or by the consumer in the following ways:

- When the list that stores execution times achieves its maximum in a given edge, it is necessary perform elements removal to accommodate new historical data. The **provider** can configure two different removal strategies. If accurate prediction is critical, then the provider should opt by dropping the longest duration values. This will make the prediction much more accurate since it uses a pessimistic approach (the algorithm always uses the fastest values on each edge). The other option is to do a random removal. In this case the mechanism may lose some accuracy (which in many cases is not highly relevant to clients), but it is able to capture the web service typical behavior in a better way.
- At runtime the **consumer** may discard a given percentage of the lowest execution times that exist in the collected history (i.e., discard the fastest values). This is quite useful to express an amount of confidence over the collected values. For example, if the consumer considers that 5% of the fastest executions happened in very particular conditions that do not capture the normal behavior of the web service and its related resources, then those 5% of values should be discarded. This represents a 95% confidence degree for timing failures prediction (see Section 9.4.1 for more details on how to express this confidence value at the client-side).

As in the detection mechanism, both checked and unchecked prediction exceptions are available. These are, respectively, *FailurePredictionException* and *FailurePredictionRuntimeException*. An important aspect is that these exceptions are hierarchically organized and extend a superclass

(*TimeFailureException*) also available in unchecked and checked versions. This enables clients to handle both detected and predicted timing failures in a single catch block.

The complete procedure for detecting and predicting timing failures is relatively straightforward. However, it is important to notice that, despite being a simple approach, it is able to produce very good results that can be used for service selection and resource management, with clear advantages for clients and servers respectively (see Section 9.4.2 for details on the experimental results).

9.4 wsTFDP in practice

This section presents an introduction to the usage of wsTFDP in a development environment. Both server-side and client-side views are discussed. The section concludes with an experimental evaluation performed to illustrate the capabilities of the mechanism in all the relevant topics presented at the beginning of this chapter (refer to Section 9.1 for details on these topics).

9.4.1 Develop web services with timing features

Developing a web service with timing failure detection and prediction characteristics using wsTFDP is a straightforward task. In fact, a developer wanting to provide time-aware web services just needs to execute the following steps:

- Add the wsTFDP library (or source code) to the project. The library and source code are available at (Laranjeiro 2011).
- Add a *TimeRestriction* parameter to the web service operations for which timing failure detection and prediction is wanted. This object holds a numeric value that is set by clients to specify the desired service duration (in milliseconds).
- Compile the project using an AOP compiler (AspectJ, in the case of our prototype). For example, when using Maven (Apache Software Foundation 2008d) as a building tool, compilation and packaging

can be done by executing the following command from the command line: 'mvn package'.

As shown in Figure 9.3, the wsTFDP-related changes (presented in bold) are quite simple. In fact, the code needed to include timing failure prediction in a web service is quite simple and easy to understand. In its basic form, the provider only needs to add an extra *TimeRestriction* parameter to the web service and our framework will perform all remaining tasks automatically. An interesting aspect is that, although wsTFDP targets web services technology, it is also able to intercept other methods (fulfilling this way the 'to be generic' requirement presented in Section 9.1), like for instance Remote Method Invocation methods (RMI). To achieve this, the programmer should mark these methods with a *@Interceptable* annotation (an annotation provided by the wsTFDP library). Obviously, these methods will also have to accept a *TimeRestriction* object as parameter.

No special measures are needed to invoke a time-aware service. In fact, the only difference between a regular service and a time-aware service, from the client point-of-view, is the use of a regular extra object – the *TimeRestriction* parameter (and optionally an exception). This has to be created and set with a desired time value and a prediction confidence factor.

Figure 9.4 shows the client code needed to **invoke a time-aware service**. The service in question is an implementation of the 'New Customer' web service specified by the TPC-App performance benchmark (Transaction Processing Performance Council 2008) and augmented to be time-aware. In this example the client is setting a maximum service execution time of 1000ms, and choosing a 95% confidence factor for historical data. The wsTFDP-related changes are presented in bold.

```
NewCustomerInput myServiceInput = new NewCustomerInput();
TimeRestriction timeRestriction = new TimeRestriction();
timeRestriction.setTimeInMillis(1000L);
timeRestriction.setConfidenceValue(0.95D);
NewCustomer proxy = new
     NewCustomerService().getNewCustomerPort();
proxy.newCustomer(timeRestriction, myServiceInput);
```

Figure 9.4 – Client code for invoking a time-aware service.

9.4.2 Experimental evaluation

To assess the effectiveness of wsTFDP we conducted an experimental evaluation that also tries to give answer to the following questions:

- Does the mechanism introduce a noticeable delay in services?
- How fast can the mechanism detect failures and how does this detection latency evolve under higher loads?
- Is it able to provide low false-positive and false-negative rates during the detection and prediction process?
- Can developers easily use the mechanism?

The TPC-App benchmark was used as test case (Transaction Processing Performance Council 2008). We applied the proposed mechanism to a subset of the services defined by this benchmark (Change Payment Method, New Customer, New Product, and Product Detail). The Payment Method and the New Customer services include an external service call that simulates a payment gateway. In order to have a more realistic representation of what usually happens when invoking services over the Internet, and based in our empirical knowledge of common service execution times, we introduced a random variable delay of 1000 to 2000 ms whenever the payment gateway was contacted.

The setup for the experiments consisted in deploying the three main test components into three separate machines connected by a Fast Ethernet network (see Table 9.I). These components are: 1) a web service provider application that provides the set of web services used in the experimental evaluation; 2) a database server on top of the Oracle 10g Database Management System (DBMS) used by the TPC-App benchmark; and 3) a workload emulator that simulates the concurrent execution of business transactions by multiple clients (i.e., that performs web services invocations).

To analyze the effectiveness of our mechanism, we executed 36 tests with different configurations. In all tests, wsTFDP was configured to predict at all service invocation points (i.e., the target service entry point and any existing nested service invocations). Each test had a duration of 20 minutes and was executed three times to increase the results representativeness. We ran tests considering different service loads (i.e., 2, 4, 8, and 16 simultaneous clients) and the system state was restored

between each run, so that tests could be performed based on the same starting conditions. Time measurement was done using nanosecond precision, provided by Java 6.

Table 9.I – Systems used for the experiments.

Node	Software	Hardware
Server	Windows server 2003 R2 Enterprise x64 Edition service pack 2 & JBoss 4.2.2.GA	Dual Core Pentium 4 3Ghz 1.46GB RAM
DBMS	Windows server 2003 R2 Enterprise x64 Edition service pack 2 & Oracle 10g	Quad Core Intel Xeon 5300 Series 7.84 GB RAM
Client	Windows XP pro SP2	Dual Core Pentium 4, 3GHz, 1.96GB RAM

A first set of tests was conducted to assess the **performance overhead.** We ran 2 experiments, one without wsTFDP and another using wsTFDP (baseline experiment A and experiment B, respectively). Each of these experiments included 4 sets (correspond to a client-side configuration of 2, 4, 8, and 16 emulated business clients) of 3 tests (3 executions of a given configuration). Our goal was to test the system using different service loads to get meaningful measurements. For experiment B we defined a high deadline for the execution so that we would not have any service being aborted due to timing failures (we just wanted to assess the performance overhead of the mechanism, when compared to the normal conditions of experiment A). The results obtained are presented in Figure 9.5.a).

For each of the four web services, we calculated the average execution time (of 3 runs) in each set of 4 tests (2, 4, 8 and 16 clients) for experiments A and B. We then extracted the minimum, maximum and average differences between experiments in each set. We finally averaged the extracted differences of each of the four web services in a single value per set. As expected, as the number of clients increases, the overhead of the detection and prediction process also increases. However, the overhead remains quite low, with an observed maximum of about 56 ms, which perfectly fits our initial goal of keeping it under 100ms for moderately loaded environments. Note that the introduced overhead is tightly associated with the complexity of running Dijkstra's algorithm over the

graph. It is well known that when the graph is not fully connected, which is frequently the case in this type of applications (i.e., each node is not connected to all remaining nodes), the time complexity is O ((m + n) log n) where 'm' is the number of edges and 'n' is the number of vertices in the graph (for algorithms that make use of a Priority Queue as in our case) (Dijkstra 1959).

The second experimental goal was to assess the **detection latency** (i.e., the time between the failure occurrence and its detection). As we were expecting very low values for these experiments, we decided to add extraneous load to the server by creating two threads in the server application, running in a continuous loop. This pushed the CPU usage of our dual core machine to 100%.

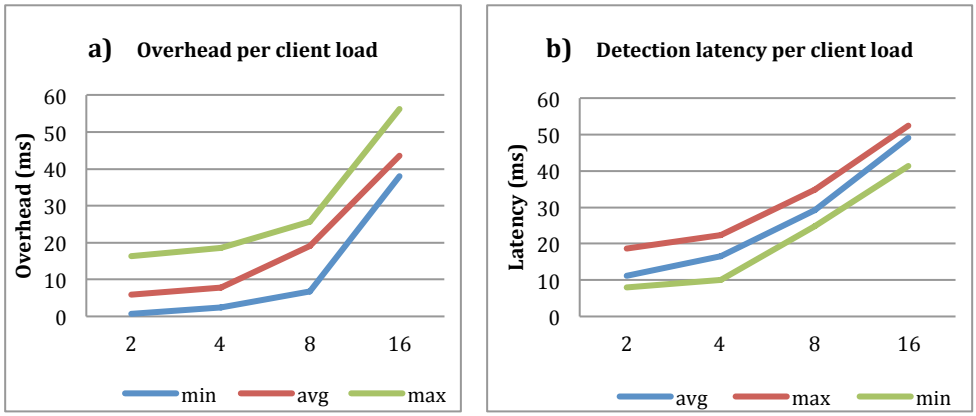


Figure 9.5 – wsTFDP mechanism overhead a) and detection latency b) per client load.

Based on the baseline reference values, we configured clients to request random service deadlines ranging from the average execution times and twice this value. Our goal was to mix services that finish on time with services that violate the client-set deadline and are terminated by wsTFDP, hence enabling us to measure the detection latency. As shown in Figure 9.5.b) four sets of tests (2, 4, 8, and 16 clients) were executed (experiment C) under the referred stress conditions. As expected, we observed increasing latency values as we added load to the server. The maximum observed latency was of 52 ms, which fulfills our initial objective. The extracted results are, in general, very good, particularly when considering the demanding environment used in this experiment (i.e., the extraneous load in the server).

In the third experiment we tried to assess the **false-positive** (i.e., the number of times wsTFDP predicted a failure that did not occur) and **false-negative** (i.e., the number of times a failure was detected but not predicted) rates. An important aspect is that, these two measures can be considered as conflicting, in the sense that, when we tune the mechanism to minimize the number of wrong predictions we increase the number of failures that may occur without being predicted. Balancing both measures can be a hard task and it is up to the user to choose if a balanced approach is preferable, or if one particular measure is the most important. In our particular case, we executed a single experiment (experiment D) and tried to minimize the false-positives, assuming that a wrong prediction has a higher cost than a failure that is not predicted. Note that, even if a timing failure is not predicted, it will be for sure detected (the detection coverage observed in all experiments performed was of 100%).

We performed some preliminary tests to check which configuration parameters would provide the lowest false-positive rates. Note that, we did not intend to execute an exhaustive search over all the available configurations, but merely aimed to choose a good enough configuration that enabled a solid demonstration of our mechanism. In real situations, users may perform a more thorough experiment according to their specific needs. We maintained the same client-side selection of deadline values (ranging from the average observed values to twice the average) and empirically we observed that, for our goals, an 80% of client-set confidence value provided the best results (we tested all lower and higher values in 5% increments, with worse results). On the provider side, the main configuration was related to the graphs edges management. We opted for a maximum list size of 100 elements and also for a random element removal strategy (see Section 9.3.3, for details on the available removal strategies). Figure 9.6 presents the obtained false-positive rates for the executed tests.

As we can see, with this moderate configuration tuning we managed to maintain the average false-positive rate under 5%, which fulfills our initial goals. This is an excellent result if we consider that we did not perform exhaustive configuration tweaking. An interesting aspect is that, the false-positive rate generally decreases as we increase the number of clients. This is related with the fact that, under heavy loads more operations are executed and, in consequence, history comprehends a time-frame that is smaller and closer to the current environment, representing it more accurately.

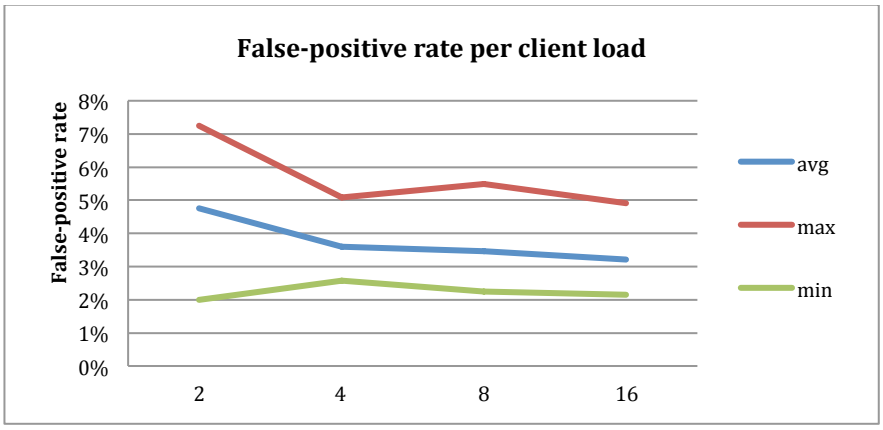


Figure 9.6 – The observed false-positive rate under different client loads.

Concerning the false-negatives, we observed a total of 26% considering all tests performed in experiment D. This is quite acceptable, as our main goal was to have a low false-positive rate (which for sure increases the false-negatives rate). Users that prefer more balanced results can fine-tune the wsTFDP mechanism, for instance by increasing the history size, decreasing the confidence value at the client-side or testing multiple combinations of the different available parameters.

To check the easiness of use of the wsTFTD mechanism we asked an external developer to implement time-awareness into our reference TPC-App implementation. This developer had about two years of experience in developing web applications. Since the application was already using Maven as building tool, the programmer's tasks were reduced to merging the application's project descriptor (a Maven configuration file) with the one provided by wsTFDP, and adding a *TimeRestrictionParameter* to each web service. The main task executed in the merging process was the replacement of the standard Java compiler with AspectJ's compiler. The whole process was concluded in less than 10 minutes, indicating that it is a fairly easy procedure.

9.5 Conclusion

This chapter proposed a programming approach to help developers to create time-aware web services. The approach proposed implements

timing failures detection and prediction in a generic and transparent way and does not add complexity to services (which typically occurs when this kind of mechanisms are created for specific services). In fact, most temporal failure detection/prediction solutions available are coupled to particular applications or to specific sets of services (i.e., these solutions are part of the application itself), which obviously has a huge impact on specific quality attributes.

The proposed framework provides a timing failure detection mechanism is able to collect historical data that can be used for prediction. The prediction mechanism is based on a component that implements Dijkstra's shortest path computation algorithm (Dijkstra 1959), which uses historical execution time values to predict at specific code points if execution can terminate on time. The prediction component can also be extended to use different prediction algorithms.

Clients can express their timing requirements by defining timeout values (and an associated confidence value for prediction). When those requirements are not possible to satisfy (e.g., because the timeout value was exceeded or because it will predictably be exceeded) the service consistently replies with a well-known exceptional behavior.

The results from several experiments show that our mechanism can be easily used and introduces a low overhead on the target system. The mechanism is also able to perform fast failure detection and can accurately predict timing failures. This sets the basis for a prediction mechanism that targets web services and is well tailored for compound services.

Chapter 10
Conclusion and Future Work

This thesis proposes methodologies and tools to assess and improve the robustness of software services. We present a generic robustness testing approach, which can be used to test the different classes of services typically present in service-oriented environments, such as web and messaging services. The approach is presented along with a discussion of relevant aspects that can influence the tests outcome. These include the analysis of the service interface, the generation of a workload (i.e., valid parameters), the definition of the tests (i.e., invalid and malicious parameters), and the characterization of results in multiple perspectives (failure severity, service behavior, and specification compliance).

The definition and execution of robustness tests is discussed in detail as it is of vital importance to the overall process. In particular, a proper selection of the fault injection strategy is important in order to target the component under test (e.g., middleware or the actual service implementation) more adequately. The fault model to apply during the tests is also of utmost importance. We present a fault model that can be adapted to different types of services and discuss the general profile for the execution of the tests. Depending on the tester's goals, several aspects can be configured when using the approach (e.g., the duration of the fault

injection phase or the replication of faults), which provides flexibility for using the technique in different scenarios.

Comparing to previous work, the proposal presented in this thesis innovates in several points. It views services' inputs in a broader way, encompassing not only typical user or client input but also incoming input from external services, using both as a fault injection points. The fact that the fault model comprises malicious values is particularly suitable in software services that usually have high exposure and can easily be the target of security attacks. Finally, the detailed three-dimensional classification scheme provides various views that can be helpful when comparing services from multiple perspectives.

The proposed approach was instantiated to two classes of services frequently used in service-oriented environments: web and messaging services. We illustrated these two cases by executing the approach with tests generated at the client-side on web services implementations and Java Message Service middleware. A tool that implements the proposed approach (*wsrbench*) was developed and used to test a large sample of publicly available web services. Even with the intrinsic limitations associated with the client-side partial view of the system under test (e.g., partial or no code access, less available points for fault injection, etc.) the approach revealed powerful enough to disclose numerous robustness problems, critical security issues in both environments, and specification compliance issues in the messaging middleware tested.

We observed that many services are being created and deployed on the web with robustness problems. The use of the proposed methodology could have prevented the deployment of services in such conditions. However, in addition to indicating clear problems, the results also provided hints for service development improvement. Observed issues like the misrepresentation of data types or the existence of domain dependencies between distinct parameters (an aspect that can be difficult to verify with regular testing) open a path for tools, that can express domains more accurately and that can perform automatic validation using those domains as basis.

This thesis also presented the application of the robustness testing approach at the server-side. In this context, more information can be used (e.g., information about the services' domains and internal structure, or access to the code or infrastructure) and we propose using it whenever it is available in order to improve the outcome of the robustness testing

process. In particular, we introduce EDEL (Extended Domain Expression Language) to provide developers with the possibility of expressing domains in a more complete manner. Besides being a starting point for the server-side robustness tests, EDEL also sets the basis for robustness improvement. In summary and starting from an EDEL description, input validation code can be automatically generated and executed, thus not requiring developer effort (in alternative, a developer should implement specific validation code every time a service is created). The experimental evaluation performed showed the application of EDEL in a typical web services scenario as the basis for executing robustness tests, which resulted in the disclosure of numerous issues. EDEL was then used as basis for the wrapper-based robustness improvement procedure presented, showing its effectiveness in the same typical scenario.

A service can be resistant to invalid inputs; however, the range of valid inputs can still pose several problems to services, in particular security issues. To target this issue, we proposed an approach that can help protecting services against SQL/XPath attacks, forms of attacks that are frequent in the web environment. The approach consists of learning the profile of genuine queries and transforming them into invariant statements that are used to prevent the execution of malicious data access requests. In a typical scenario, the approach showed to be lightweight, while still effectively protecting services from malicious requests. In addition to these techniques, we discussed the integration of robustness testing (as a means for creating more robust services) in a current agile software development process – Test-Driven Development.

In this thesis we also propose the use of fault tolerance mechanisms in the web services context, in two dimensions. We first presented and discussed a mechanism (FTWS) that is able to use diverse services as a means to increase dependability related properties like availability, response correctness, and response time. Results suggest that the FTWS can be very useful in increasing the availability of services. Its capabilities indicate its utility in improving correctness and response time. We later discussed services timing requirements and presented a mechanism (wsTFDP) to detect and predict timing failures in web services environments. wsTFDP features low intrusiveness and fast timing failure detection capabilities. However, it can be most useful in predicting timing failures. In fact, the experiments conducted showed that a simple algorithm for timing failures prediction was able to predict the occurrence of timing failures, with great success. Besides giving the opportunity to clients to try other services that

reply to demands in a timely fashion, servers can also use this information to better manage resources, by lowering the priority of requests that will not complete in a timely manner.

Future work

Several topics are currently under research as a continuation of the work performed in this thesis.

- **Study the applicability of machine-learning algorithms in the classification of web services robustness:** testing web services for robustness can be a lengthy and arduous process. After testing a set of services, there is typically a very large quantity and variety of test results to be analyzed, which poses a challenge to the developer that has to manually process all results and identify the outputs that indicate the presence of bugs in the code. This complex classification process can easily lead to errors resulting from the human intervention in such a laborious task. Text classification algorithms have been applied successfully in many contexts (e.g., spam identification, text categorization, etc.) and are considered a powerful tool for the successful automation of several classification-based tasks. In (Laranjeiro, Oliveira, and Vieira 2010) we study the applicability of five widely used text classification algorithms (Support Vector Machines, Naïve Bayes, Large Linear Classification, K-nearest neighbor (Ibk), and Hyperpipes) in the context of web services robustness testing. Results indicate that these algorithms can be effectively used to automate the identification of robustness issues while reducing human intervention. However, in all mechanisms there are cases of misclassified responses, indicating space for improvement.
- **Research new algorithms for classifying web services robustness:** well-known automatic classification algorithms can help classifying the robustness of web services in an automatic way. However, the applicability of such algorithms is limited, as they are frequently unable to deal with the large diversity of outputs present in typical web services scenarios, thus producing incorrect results. In (Oliveira, Laranjeiro, and Vieira 2011) we propose an approach that combines rule-based classification (including domain rules) and conventional machine-learning

algorithms trained using generic data to automatically classify web services robustness. Results show the effectiveness of the technique, indicating that it can be integrated in the robustness testing procedure, enabling the delivery of a full end-to-end automatic approach for web services robustness testing.

- **Specializing tests for identifying security vulnerabilities:** as we have seen earlier, valid inputs can also become a security problem for services. Tests can be further specialized with the goal of detecting security vulnerabilities. The workload can again be extremely important since we must exercise all existing data access statements (and ideally each data access statement in all possible valid ways). In (Antunes et al. 2009) we present a preliminary approach that uses the SQL/XPath Injection protection technique presented in this thesis with the goal of detecting vulnerabilities in web services. Vulnerability detection is performed by comparing the structure of the SQL/XPath commands issued in the presence of attacks to the ones previously learned when running the workload in the absence of attacks. In this study we show that our approach can perform much better than known tools (including commercial ones), achieving extremely high detection coverage while maintaining the false positives rate very low.

The work presented in this thesis has largely contributed to gain a broad experience on services robustness testing and improvement. In addition, this work provided us an excellent standard environment for the evaluation and comparison of alternative services based on their robustness and security characteristics. This way, several topics can be foreseen as a continuation of the present work:

- **Creating an approach to test complex service environments for robustness:** in services environments it is frequent to have services behaving as clients of other services, which in turn can, themselves, be services compositions. This draws a highly complex scenario, which can be challenging to test if we consider that some of the components can be exchanged or updated at runtime. In such scenario, it can be difficult to guarantee that the service that being provided for end-users maintains a given level of robustness. Furthermore, if there is no complete knowledge of the services involved in a particular service environment (i.e., some of the known services use external services that are virtually

unknown to the main provider) we can understand that a simple change can affect the robustness of the whole composition. Future research should focus on several aspects in these environments: the generation of representative workloads can be more complex; the discovery of unknown services can be extremely useful for testing and workload generation; the application of isolation techniques to components can be useful for performing tests only on particular components.

- **Generating high coverage workloads:** a workload that is able to explore more parts of a service code is more helpful in creating robustness tests that are can be more effective (i.e., can disclose more software bugs). Future research should focus on studying ways of producing high coverage workloads, which combined with robustness tests, as proposed in this thesis, may compose a strong testing tool that can help reducing the overall developer effort in testing phases. An important aspect is that, when producing a workload, the number of possible values can be much larger than what is admissible to use in a testing phase. In fact, frequently, the values to be used in a workload must be selected (so that the number of tests can be reduced, for instance, by eliminating redundant tests). This means that there must be a value elimination criterion that ideally should be executed in an automatic way. Obviously, this criterion can vary from service to service.
- **Extending robustness tests to focus on web services protocols extensions and other resources:** the web services protocols are many times used with extensions that provide extra functionalities (e.g., reliable message delivery extensions, support for transactions, etc.). Robustness tests can be used to test how the web services that use these extensions (or the extensions themselves) behave in the presence of invalid and malicious input conditions. Similarly, elements frequently present in web services environment, like discovery services, can also be the target of testing. Regard that the instantiation of the robustness testing approach to these targets may require adaptations to its components or generic procedure in order to fit the specificities of the services under testing.
- **Extending robustness tests to focus on other resources involved in messaging environments:** the instantiation of the robustness

testing procedure presented in this thesis focuses on the JMS message for introducing faulty elements. Future research can include the extension of the robustness tests to other important resources in a JMS environment such as the connection, session, producers or consumers. These are all participating resources in JMS based applications, so in this sense it can be useful to study the robustness properties of each of these components (for example, applying mutations to the metadata of a connection object) and verify the response of the JMS provider.

- **Creating instances of the robustness testing approach for other types of services or applications:** in typical client server interaction in service oriented environments there can be more types of services and participating resources than the ones explored in this thesis. Enterprise messaging buses, databases, mobile clients middleware, among many others, are all distinct technologies that can pose new research challenges and can be the target of robustness testing. As presented in this thesis, an adapted approach can be helpful in disclosing software bugs that pass silently even in highly tested software.
- **Providing additional security protection schemes for web services:** in many cases, it is not possible to protect web services against SQL/XPath attacks. There may not be access to the service bytecode, it may not be possible to change the service that is running (and servicing clients), it may not be feasible to introduce a learning phase in a production system, or it may be difficult to evaluate the completeness of a learning phase. Research should focus on solutions for these cases including the study of non-intrusive service wrappers, providing attack knowledge to the service without using a learning phase (solutions can involve automated ways of changing the vulnerable code, when possible, or creating a compiler extension that produces non-vulnerable code, whenever data access statements are involved). Additionally, other security issues, that may require different protection schemes, can be taken in consideration in a more global protection mechanism (e.g., Insecure Direct Object References) (The Open Web Application Security Project (OWASP) 2010).
- **Providing ways for developers to create fault tolerant stateful services compositions:** as previously referred, when using multiple services to provide a fault tolerant service, it is important

to keep consistency whenever those services maintain an internal state. In fact, when invoking a set of redundant stateful services, if a replica fails, the internal states will not be consistent (among the various replicas). This way, it is important to detect such failure and devise ways to recover and update the failed component's state.

- **Studying the applicability of machine-learning algorithms for timing failures prediction:** our proposal to predict timing failures uses a simple prediction algorithm. Although it showed to be effective, it is important to study the usefulness and general applicability of machine-learning algorithms. These algorithms have already been used in prediction contexts with success; however, their utility in predicting timing failures in web services contexts remains unexplored and thus remains as an opportunity for research.

References

Acunetix. 2011. Acunetix Web Vulnerability Scanner. *Acunetix Web Vulnerability Scanner*. http://www.acunetix.com/vulnerability-scanner/.

Amazon. 2010. Amazon Web Services Solutions Catalog. http://aws.amazon.com/products/.

———. 2011. Amazon Web Services. http://aws.amazon.com/.

Ammann, P.E., and J.C. Knight. 1988. "Data diversity: an approach to software fault tolerance." *IEEE Transactions on Computers* 37 (4): 418-425.

Anley, Chris. 2002. "Advanced SQL Injection In SQL Server Applications." *White paper, Next Generation Security Software Ltd*.

Antunes, Nuno, Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. 2009. Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services. In *IEEE International Conference on Services Computing (SCC 2009)*, 260-267. Bangalore, India: IEEE Computer Society, September 21. doi:10.1109/SCC.2009.23.

Apache Software Foundation. 2008a. Apache Axis. http://ws.apache.org/axis/.

———. 2008b. Apache ActiveMQ. http://activemq.apache.org/.

———. 2008c. Apache Geronimo. http://geronimo.apache.org/.

———. 2008d. Apache Maven Project. http://maven.apache.org/.

———. 2010a. Apache Commons Math. http://commons.apache.org/math/.

———. 2010b. Apache Commons Validator. http://commons.apache.org/validator/.

Avizienis, A. 1995. "The Methodology of N-Version Programming." *Software Fault Tolerance*: 23–46.

Avizienis, A., and J.P.J. Kelly. 1984. "Fault Tolerance by Design Diversity: Concepts and Experiments." *Computer* 17 (8): 67-80.

Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing* 1 (1): 11-33.

Barros, Marcelo De, Jing Shiau, Chen Shang, Kenton Gidewall, Hui Shi, and Joe Forsmann. 2007. Web Services Wind Tunnel: On Performance Testing Large-Scale Stateful Web Services. In *Dependable Systems and Networks, International Conference on*, 0:612-617. Los Alamitos, CA, USA: IEEE Computer Society. doi:http://doi.ieeecomputersociety.org/10.1109/DSN.2007.102.

Beck, K. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

Beck, Kent. 2011. JUnit. http://kentbeck.github.com/junit/.

Bergmann, Volker. 2008. databene benerator. http://databene.org/databene-benerator.

Bernard, Emmanuel. 2010. jsr 303: Bean Validation. http://jcp.org/en/jsr/detail?id=303.

Bittner, K., and I. Spence. 2003. *Use case modeling*. Addison-Wesley Professional.

Bo, Yang, and Li Xiang. 2007. A study on software reliability prediction based on support vector machines. In *IEEE International Conference on Industrial Engineering and Engineering Management*, 1176-1180. doi:10.1109/IEEM.2007.4419377.

Bouwers, Eric. 2011. PHP-sat: PHP static analysis tool. http://www.program-transformation.org/PHP/PhpSat.

Bovy, CJ, HT Mertodimedjo, G. Hooghiemstra, H. Uijterwaal, and P. Van Mieghem. 2002. Analysis of end-to-end delay measurements in Internet. In *Proc. of the Passive and Active Measurement Workshop-PAM'2002*.

Bravenboer, Martin, Eelco Dolstra, and Eelco Visser. 2007. Preventing injection attacks with syntax embeddings. In *Proceedings of the 6th international conference on Generative programming and component engineering*, 3-12. Salzburg, Austria: ACM. doi:10.1145/1289971.1289975.

Bray, T., J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. 2000. "Extensible Markup Language (XML) 1.0." *W3C Recommendation* 6. http://www.w3.org/XML/.

Budd, T.A. 1981. "Mutation analysis: Ideas, examples, problems and prospects." Ed. B. Chandrasekaran and S. Radicchi. *Computer Program Testing*: 129–148.

Campwood Software. 2011. *SourceMonitor*. http://www.campwoodsw.com/sourcemonitor.html.

Carreira, J., H. Madeira, and J. G Silva. 1998. "Xception: a technique for the experimental evaluation ofdependability in modern computers." *IEEE Transactions on Software Engineering* 24 (2) (February): 125-136. doi:10.1109/32.666826.

Carrette, G. J. 1996. "CRASHME: Random Input Testing." http://people. delphi. com/gjc/crashme. html.

Casati, F., and V. Machiraju. 2003. “Business Visibility with Web services: Making sense of your IT operations and of what they mean to you.” *Proceedings of UMICS*: 123–135.

Chen, Shiping, and Paul Greenfield. 2004. QoS Evaluation of JMS: An Empirical Approach. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*, 90276.2. IEEE Computer Society.

Chillarege, Ram, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. 1992. “Orthogonal Defect Classification-A Concept for In-Process Measurements.” *IEEE Trans. Softw. Eng.* 18 (11) (November): 943–956. doi:10.1109/32.177364.

Christey, Steve, and Robert Martin. 2007. Vulnerability type distributions in CVE. *V1.1*. http://cwe.mitre.org/documents/vuln-trends/index.html.

Christmansson, J., and R. Chillarege. 1996. Generation of an error set that emulates software faults based onfield data. In , *Proceedings of Annual Symposium on Fault Tolerant Computing, 1996*, 304-313. IEEE, June 25. doi:10.1109/FTCS.1996.534615.

Christmansson, J., and P. Santhanam. 1996. Error injection aimed at fault removal in fault tolerance mechanisms-criteria for error selection using field data on softwarefaults. In , *Seventh International Symposium on Software Reliability Engineering, 1996. Proceedings*, 175-184. IEEE, November 30. doi:10.1109/ISSRE.1996.558785.

Chromatic. 2003. *Extreme Programming Pocket Guide*. 1st ed. O'Reilly Media, June.

Coelho, Roberta, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. 2006. Unit testing in multi-agent systems using mock agents and aspects. In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems - SELMAS '06*, 83. Shanghai, China. doi:10.1145/1138063.1138079.

Cohn, M. 2004. *User Stories Applied*. Addison-Wesley Boston.

Collabnet. 2008. Subversion. http://subversion.tigris.org/.

Copty, Shady, and Shmuel Ur. 2005. Multi-threaded Testing with AOP Is Easy, and It Finds Bugs! In *Euro-Par 2005 Parallel Processing*, ed. José C. Cunha and Pedro D. Medeiros, 3648:740-749. Berlin, Heidelberg: Springer Berlin Heidelberg.

Coulouris, George F., Jean Dollimore, and Tim Kindberg. 2005. *Distributed systems: concepts and design*. Pearson Education.

Crimson Consulting Group. 2003. *High Performance JMS Messaging: A Benchmark Comparison of Sun Java System Message Queue and IBM WebSphere MQ*. White Paper. Sun Microsystems.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. 2002. "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI." *Internet Computing, IEEE* 6 (2): 86-93. doi:10.1109/4236.991449.

Daniel, Florian, Fabio Casati, Vincenzo D'Andrea, Emmanuel Mulo, Uwe Zdun, Schahram Dustdar, Steve Strauch, et al. 2009. Business Compliance Governance in Service-Oriented Architectures. In *Advanced Information Networking and Applications, International Conference on*, 0:113-120. Los Alamitos, CA, USA: IEEE Computer Society. doi:http://doi.ieeecomputersociety.org/10.1109/AINA.2009.112.

Daran, Murial, and Pascale Thévenod-Fosse. 1996. Software error analysis: a real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, 158–171. ISSTA '96. New York, NY, USA: ACM. doi:10.1145/229000.226313. http://doi.acm.org/10.1145/229000.226313.

De, Pradipta, Anindya Neogi, and Tzi-cker Chiueh. 2003. VirtualWire: A Fault Injection and Analysis Tool for Network Protocols. In *Distributed Computing Systems, International Conference on*, 0:214. Los Alamitos, CA, USA: IEEE Computer Society. doi:http://doi.ieeecomputersociety.org/10.1109/ICDCS.2003.1203468.

DeMillo, R. A, D. S Guindi, W. M McCracken, A. J Offutt, and K. N King. 1988. An extended overview of the Mothra software testing environment. In , *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988*, 142-151. IEEE, July 19. doi:10.1109/WST.1988.5369.

Dijkstra, E. W. 1959. "A note on two problems in connexion with graphs." *Numerische Mathematik* 1 (1): 269-271.

Doliner, Mark. 2008. Cobertura. http://cobertura.sourceforge.net/.

Duraes, João, and Henrique Madeira. 2002. Emulation of software faults by educated mutations at machine-code level. In *13th International Symposium on Software Reliability Engineering (ISSRE 2003)*, 329- 340. IEEE. doi:10.1109/ISSRE.2002.1173283.

———. 2003. Definition of software fault emulation operators: a field data study. In *International Conference on Dependable Systems and Networks (DSN 2003)*, 105- 114. IEEE, June 22. doi:10.1109/DSN.2003.1209922.

———. 2004. Generic faultloads based on software faults for dependability benchmarking. In *International Conference on Dependable Systems and Networks (DSN 2004)*, 285- 294. IEEE, July 28. doi:10.1109/DSN.2004.1311898.

Durães, João, and Henriques Madeira. 2006. "Emulation of Software Faults: A Field Data Study and a Practical Approach." *IEEE Transactions on Software Engineering* 32 (11) (November): 849-867. doi:10.1109/TSE.2006.113.

Duvall, Paul M., Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1st ed. Addison-Wesley Professional, July 9.

Eclipse Foundation. 2008. The AspectJ Project. http://www.eclipse.org/aspectj/.

Edvardsson, J. 1999. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, 21–28.

Elmagarmid, A. K. 1992. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann.

Engelmann, C., and S. L. Scott. 2005. "Concepts for high availability in scientific high-end computing." *In Proceedings of High Availability and Performance Workshop (HAPCW 2005)* 11: 2005. doi:10.1.1.60.7186.

Erl, Thomas. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall Professional Technical Reference.

Eviware. 2011. soapUI. http://www.soapui.org/.

Exhedra Solutions, Inc. 2010. Planet Source Code. http://www.planet-source-code.com/.

Fabre, J.-C., F. Salles, M. Rodríguez Moreno, and J. Arlat. 1999. Assessment of COTS Microkernels by Fault Injection. In *Proceedings of the conference on Dependable Computing for Critical Applications*, 25. IEEE Computer Society.

Fagan, Michael. 2002. Design and code inspections to reduce errors in program development. In *Software pioneers: contributions to software engineering*, 575-607. Springer-Verlag New York, Inc. http://portal.acm.org/citation.cfm?id=944331.944367&coll=Portal&dl=GUIDE&CFID=26149866&CFTOKEN=67700479.

Feathers, Michael. 2004. *Working effectively with legacy code*. Prentice Hall PTR.

Ferris, Christopher, Anish Karmarkar, Prasad Yendluri, Keith Ballinger, David Ehnebuske, Martin Gudgin, Canyang Liu, and Mark Nottingham. 2007. WS-I Basic Profile - Version 1.2. October 24. http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html.

Fetzer, C., and Zhen Xiao. 2003. HEALERS: a toolkit for enhancing the robustness and security of existing applications. In *International Conference on Dependable Systems and Networks (DSN)*, 317-322. San Francisco, California, USA, June 22.

Fielding, F., UC Irvine, J. Gettys, Compac/W3C, H. Frystyk, W3C/MIT, L. Masinter, et al. 1999. *Hypertext Transfer Protocol -- HTTP/1.1*. Request For Comments (RFC). http://www.ietf.org/rfc/rfc2616.txt.

Fortify Software. 2011. Fortify 360. https://www.fortify.com/.

Frank, Eibe, Geoff Holmes, Mike Mayo, Bernhard Pfahringer, Tony Smith, and Ian Witten. 2010. Weka 3 - Data Mining with Open Source Machine Learning Software in Java. http://www.cs.waikato.ac.nz/ml/weka/.

Friedl, Jeffrey. 2006. *Mastering Regular Expressions*. O'Reilly Media, Inc.

Fu, Chen, Barbara G Ryder, Ana Milanova, and David Wonnacott. 2004. "Testing of java web services for robustness." *ACM SIGSOFT Software Engineering Notes*. ISSTA '04: 23–34. Boston, Massachusetts, USA. doi:10.1145/1007512.1007516.

Fugini, Maria Grazia, Barbara Pernici, and Filippo Ramoni. 2009. "Quality analysis of composed services through fault injection." *Information Systems Frontiers* 11 (3) (July): 227–239. doi:10.1007/s10796-008-9086-3.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 10.

Ghosh, A.K., M. Schmid, and F. Hill. 1995. Wrapping windows NT software for robustness. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing. Digest of Papers.*, 344-347. Pasadena, California, EUA, June 27.

Goodenough, John B. 1975. "Exception handling: issues and a proposed notation." *Communications of the ACM* 18 (December): 683–696. doi:10.1145/361227.361230.

Gorbenko, A., V. Kharchenko, and A. Romanovsky. 2007. Vertical and Horizontal Composition in Service-Oriented Architecture. In *Workshop on Methods, Models and Tools for Fault Tolerance at the Integrated Formal Methods (IFM) Conference*. Oxford, UK.

Gray, J. 1990. "A census of Tandem system availability between 1985 and 1990." *IEEE Transactions on Reliability* 39 (4) (October): 409-418. doi:10.1109/24.58719.

Gray, Jim. 1985. *Why do computers stop and what can be done about it*. Techical Report. Tandem Computers, June.

———. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Green, Roedy. 2007. UTF: Java Glossary. *Roedy Green's Java & Internet Glossary*. http://mindprod.com/jgloss/utf.html.

Grottke, M., and K. S. Trivedi. 2007. "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate." *COMPUTER*: 107-109.

Halang, Wolfgang A., Roman Gumzej, Matjaz Colnaric, and Marjan Druzovec. 2000. "Measuring the Performance of Real-Time Systems." *The International Journal of Time-Critical Computing Systems* 18 (1) (January 1): 59-68. doi:10.1023/A:1008102611034.

Halfond, William G. J., and Alessandro Orso. 2006. Preventing SQL injection attacks using AMNESIA. In *Proceedings of the 28th international conference on Software engineering*, 795-798. Shanghai, China: ACM. doi:10.1145/1134285.1134416.

Halfond, William, Jeremy Viegas, and Alessandro Orso. 2006. A classification of SQL-injection attacks and countermeasures. In *International Symposium on Secure Software Engineering*. IEEE Computer Society.

Han, Seungjae, Harold A Rosenberg, and Kang G Shin. 1993. *DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment*. Technical Report. University of Michigan. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.5652.

Hewlett Packard. 2011. HP WebInspect. http://www.hp.pt.

Highsmith, J., and A. Cockburn. 2001. "Agile software development: the business of innovation." *Computer* 34 (9): 120-127. doi:10.1109/2.947100.

Hirt, Marcus. 2008. *Oracle JRockit Mission Control Overview*. June. http://www.missioncontrol-whitepaper-june08-1-130357.pdf/.

Horning, James J., Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell. 1974. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, 171-187. Springer-Verlag.

Hovemeyer, David, and William Pugh. 2004. "Finding bugs is easy." *ACM SIGPLAN Notices* 39 (12): 92-106. doi:10.1145/1052883.1052895.

Hsueh, Mei-Chen, Timothy K. Tsai, and Ravishankar K. Iyer. 1997. "Fault Injection Techniques and Tools." *Computer* 30 (4): 75-82.

Huang, Y., C. Kintala, N. Kolettis, and N.D. Fulton. 1995. Software rejuvenation: analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25) Digest of Papers*, 381-390. doi:10.1109/FTCS.1995.466961.

IBM. 2011. IBM Rational Appscan. http://www-01.ibm.com/software/awdtools/appscan.

IEEE. 1990. *IEEE Standard Glossary of Software Engineering Terminology*. http://standards.ieee.org.

IEEE Computer Society. 2004. *Guide to the software engineering body of knowledge (swebok)*. Ed. Alain Abran, James Moore, Pierre Bourque, and Robert Dupuis. IEEE Computer Society.

Interface21. 2008. Acegi Security - Acegi Security System for Spring. http://www.springsource.org/spring-security/.

Iyer, Ravishankar K. 1995. Experimental evaluation. In *Proceedings of the Twenty-Fifth international conference on Fault-tolerant computing*, 115–132. FTCS'95. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=1899254.1899269.

Janzen, D., and H. Saiedian. 2005. "Test-driven development concepts, taxonomy, and future direction." *Computer* 38 (9): 43-50. doi:10.1109/MC.2005.314.

JBoss Community. 2011. Hibernate Validator. http://www.hibernate.org/subprojects/validator.html.

Jendrock, Eric, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. 2006. *The Java EE 5 Tutorial*. 3rd ed. Prentice Hall PTR.

Jendrock, Eric, Ian Evans, Devika Gollapudi, Kim Haase, and Chinmayee Srivathsa. 2011. The Java EE 6 Tutorial. July. http://download.oracle.com/javaee/6/tutorial/doc/.

Jenn, E., J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. 1994. Fault injection into VHDL models: the MEFISTO tool. In , *Twenty-Fourth International Symposium on Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers*, 66-75. IEEE, June 15. doi:10.1109/FTCS.1994.315656.

Jeon, Sang-hun. 2011. Gamja : Web vulnerability scanner. http://sourceforge.net/projects/gamja/.

Jovanovic, Nenad. 2011. Pixy: XSS and SQLI Scanner for PHP. http://pixybox.seclab.tuwien.ac.at/pixy/index.php.

Kalyanakrishnam, M., Z. Kalbarczyk, and R. Iyer. 1999. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, 178. IEEE Computer Society.

Kaner, Cem, Jack Falk, and Hung Q. Nguyen. 1999. *Testing Computer Software, 2nd Edition*. 2nd ed. Wiley, April 12.

Kanoun, Karama, and Lisa Spainhower. 2008. *Dependability Benchmarking for Computer Systems*. 1st ed. Wiley-IEEE Computer Society Press, July 28.

Kersten, Mik. 2005. AOP@Work: AOP tools comparison, Part 1. CT316. February 8. http://www.ibm.com/developerworks/library/j-aopwork1/.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. 1997. Aspect-Oriented Programming. In *11th European Conference on Object-oriented Programming*. Jyväskylä, Finland, June 13.

Koopman, P., and J. DeVale. 1999. Comparing the robustness of POSIX operating systems. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 30-37. http://ieeexplore.ieee.org/iel5/6328/16917/00781031.pdf.

Koopman, P., J. Sung, C. Dingman, D. Siewiorek, and T. Marz. 1997. Comparing operating systems using robustness benchmarks. In *Proceedings of the Sixteenth Symposium on Reliable Distributed Systems*, 72-79.

Kootbally, Z., R. Madhavan, and C. Schlenoff. 2006. Prediction in Dynamic Environments via Identification of Critical Time Points. In *Military Communications Conference, 2006. MILCOM 2006*, 1-7. doi:10.1109/MILCOM.2006.302047.

Krafzig, Dirk, Karl Banke, and Dirk Slama. 2004. *Enterprise SOA: Service-Oriented Architecture Best Practices*. 1st ed. Prentice Hall, November 19.

Kropp, N. P, P. J Koopman, and D. P Siewiorek. 1998. Automated robustness testing of off-the-shelf software components. In *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 1998. Digest of Papers*, 230-239. IEEE, June 23. doi:10.1109/FTCS.1998.689474.

Laranjeiro, Nuno. 2011. Tool Kit for Robustness Testing, Improvement, and Security Improvement. http://eden.dei.uc.pt/~cnl/papers/2011-phd-thesis-toolkit.zip.

Laranjeiro, Nuno, Rui Oliveira, and Marco Vieira. 2010. Applying Text Classification Algorithms in Web Services Robustness Testing. In *29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)*. New Dehli, India: IEEE Computer Society, November 31. about:blank.

Lee, Inhwan, and Ravishankar K. Iyer. 1995. "Software Dependability in the Tandem GUARDIAN System." *IEEE Transactions on Software Engineering* 21 (5): 455-467.

Livshits, V. Benjamin, and Monica S. Lam. 2005. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, 18-18. Baltimore, MD: USENIX Association.

Long, D. D. E., J. L. Carroll, and C. J. Park. 1990. *A Study of the Reliability of Internet Sites*. University of California at Santa Cruz. ACM.

Looker, N., M. Munro, and Jie Xu. 2005. Increasing Web Service Dependability Through Consensus Voting. In *29th Annual International Computer Software and Applications Conference (COMPSAC)*, 2:66-69. doi:10.1109/COMPSAC.2005.88.

Lyu, Michael R., ed. 1996. *Handbook of software reliability engineering*. McGraw-Hill, Inc. http://portal.acm.org/citation.cfm?id=239425&coll=Portal&dl=GUIDE&CFID=82384114&CFTOKEN=14785366.

Madeira, Henrique, Diamantino Costa, and Marco Vieira. 2000. On the emulation of software faults by software fault injection. In *Proceedings International Conference on Dependable Systems and Networks, 2000. DSN 2000*, 417-426. IEEE. doi:10.1109/ICDSN.2000.857571.

Marcus, Evan, and Hal Stern. 2003. *Blueprints for high availability: designing resilient distributed systems*. 2nd ed. John Wiley & Sons, Inc.

Maron, Jonathan. 2004. An overview of the CTS for J2EE component developers. http://www2.sys-con.com/ITSG/virtualcd/Java/archives/0701/maron/index.html.

Marsden, Eric, and Jean-Charles Fabre. 2001. Failure Mode Analysis of CORBA Service Implementations. In *Middleware 2001*, ed. Rachid Guerraoui, 2218:216-231. Berlin, Heidelberg: Springer Berlin Heidelberg.

Martin, Evan, Suranjana Basu, and Tao Xie. 2007. "WebSob: A Tool for Robustness Testing of Web Services." *Companion to the proceedings of the 29th International Conference on Software Engineering*. ICSE COMPANION '07: 65–66. doi:http://dx.doi.org/10.1109/ICSECOMPANION.2007.84.

May, Nicholas. 2009. A Redundancy Protocol for Service-Oriented Architectures. In *Service-Oriented Computing – ICSOC 2008 Workshops*, 5472/2009:211-220. Springer-Verlag. doi:10.1007/978-3-642-01247-1_22.

McCluskey, Glen. 1998. Using Java Reflection. January. http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

McConnell, Steve. 2004. *Code Complete: A Practical Handbook of Software Construction*. 2nd ed. Microsoft Press, July 7.

McKinsey & Company, and SandHill Group. 2008. *Enterprise Software Customer Survey 2008*. Survey. www.interop.com/downloads/mckinsey_interop_survey.pdf.

Menasce, D. A. 2002. "QoS issues in Web services." *IEEE Internet Computing* 6 (6) (December): 72- 75. doi:10.1109/MIC.2002.1067740.

Mendonça, Manuel, and Nuno Neves. 2007. Robustness Testing of the Windows DDK. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN '07*, 554-564. http://ieeexplore.ieee.org/iel4/4272934/4272935/04273006.pdf.

Menth, Michael, Robert Henjes, Christian Zepfel, and Sebastian Gehrsitz. 2006. Throughput performance of popular JMS servers. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, 367-368. Saint Malo, France: ACM.

Microsoft Corporation. 2004. Web Services Performance: Comparing Java 2TM Enterprise Edition (J2EETM platform) and the Microsoft® .NET Framework - A Response to Sun Microsystem's Benchmark. http://www.theserverside.net/tt/articles/showarticle.tss?id=SunBenchmarkResponse.

———. 2010. Message Queuing (MSMQ). August 16. http://msdn.microsoft.com/en-us/library/ms711472(v=VS.85).aspx.

———. 2012. .NET Framework. http://msdn.microsoft.com/en-us/netframework.

Miller, B. P., D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. 1995. *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. University of Wisconsin-Madison, Computer Sciences Dept. Google Scholar.

Miller, Claire Cain. 2011. Amazon Cloud Failure Takes Down Web Sites. *The New York Times*. http://bits.blogs.nytimes.com/2011/04/21/amazon-cloud-failure-takes-down-web-sites/.

Mookhey, K. K., and Nilesh Burghate. Detection of SQL Injection and Cross-site Scripting Attacks. http://www.securityfocus.com/infocus/1768.

Mukherjee, A., and D.P. Siewiorek. 1997. "Measuring software dependability by robustness benchmarking." *Transactions on Software Engineering* 23 (6): 366-378.

Musa, John D. 1998. *Software Reliability Engineering*. Osborne/McGraw-Hill, July 31.

Myers, G. J. 2004. *The art of software testing, Second Edition*. John Wiley & Sons Inc. New York.

NUnit.org. 2011. NUnit. http://www.nunit.org/.

Offutt, J., Wuzhi Xu, and Juan Luo. 2005. Testing Web services by XML perturbation. In *16th IEEE International Symposium on Software Reliability Engineering*, 10 pp. http://ieeexplore.ieee.org/iel4/10370/32973/01544740.pdf.

Offutt, Jeff, and Wuzhi Xu. 2004. "Generating test cases for web services using data perturbation." *ACM SIGSOFT Software Engineering Notes* 29 (5) (September): 1. doi:10.1145/1022494.1022529.

Oliveira, Rui, Nuno Laranjeiro, and Marco Vieira. 2011. A Composed Approach for Automatic Classification of Web Services Robustness. In *The 8th International Conference on Services Computing (SCC 2011)*. Washington D.C., USA: IEEE Computer Society, July 4.

Oracle. 2011a. Oracle Advanced Queuing. *Oracle Database JDBC Developer's Guide and Reference, 11g Release 1 (11.1)*. http://docs.oracle.com/cd/B28359_01/java.111/b31224/streamsaq.htm.

———. 2011b. JAXB Reference Implementation. http://jaxb.java.net/.

———. 2011c. Java Platform, Standard Edition (Java SE). http://www.oracle.com/technetwork/java/javase/overview/index.html.

———. 2012. VisualVM. January. http://visualvm.java.net/.

Ostrand, Thomas. 2002a. White-Box Testing. In *Encyclopedia of Software Engineering, 2nd Edition*, ed. John Marciniak. 2nd ed. Wiley.

———. 2002b. Black-Box Testing. In *Encyclopedia of Software Engineering, 2nd Edition*, ed. John Marciniak. 2nd ed. Wiley.

Pan, Jiantao, Philip Koopman, Daniel P. Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. 2001. Robustness Testing and Hardening of CORBA ORB Implementations. In *The 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, 141-150. IEEE Computer Society.

Pandit, Bhalchandra, Valentina Popescu, and Virginia Smith. 2009. *Service Modeling Language, Version 1.1*. http://www.w3.org/TR/sml/.

Parhami, B. 1994. "Voting algorithms." *Reliability, IEEE Transactions on* 43 (4): 617-629. doi:10.1109/24.370218.

Peisert, Sean, and Matt Bishop. 2007a. How to Design Computer Security Experiments. In *Fifth World Conference on Information Security Education*, ed. Lynn Futcher and Ronald Dodge, 237:141-148. Boston, MA: Springer US. http://www.springerlink.com/content/074t33453235m25g/.

———. 2007b. "I'm a Scientist, Not a Philosopher!" *IEEE Security & Privacy Magazine*.

Pekilis, B.R., and R.E. Seviora. 1997. Detection of response time failures of real-time software. In *The Eighth International Symposium On Software Reliability Engineering*, 38-47. doi:10.1109/ISSRE.1997.630846.

Popov, P., L. Strigini, S. Riddle, and A. Romanovsky. 2001. Protective Wrapping of OTS Components. In *Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*. Toronto.

PortSwigger. 2011. Burp Suite. http://portswigger.net/burp/.

Red Hat Middleware. 2007. JBoss Messaging. http://labs.jboss.com/jbossmessaging/.

———. 2008a. JBossMQ. http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossMQ.

———. 2008b. JBoss Application Server. http://www.jboss.org/jbossas/.

Rising, L., and N. S Janoff. 2000. "The Scrum software development process for small teams." *IEEE Software* 17 (4) (August): 26-32. doi:10.1109/52.854065.

Rodríguez, Manuel, Arnaud Albinet, and Jean Arlat. 2002. MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems. In *The 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, 267-272. IEEE Computer Society.

Rodríguez, Manuel, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. 1999. MAFALDA: Microkernel Assessment by Fault Injection and Design Aid. In *The Third European Dependable Computing Conference on Dependable Computing*, 143-160. Springer-Verlag. doi:10.1007/3-540-48254-7_11.

Rui Wang, and Ning Huang. 2008. Requirement Model-Based Mutation Testing for Web Service. In *4th International Conference on Next Generation Web Services Practices, 2008. NWESP '08*, 71-76. IEEE, October 20. doi:10.1109/NWeSP.2008.20.

Sachs, Kai, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. 2009. "Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark." *Performance Evaluation* 66 (8) (August): 410-434. doi:16/j.peva.2009.01.003.

Salas, J., F. Perez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris. 2006. "WS-replication: a framework for highly available web services." *Proceedings of the 15th international conference on World Wide Web*: 357-366.

Salatge, Nicolas, and Jean-Charles Fabre. 2007. Fault Tolerance Connectors for Unreliable Web Services. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 51-60.

Santiago, Valdivino, Ana Silvia Martins do Amaral, N. L. Vijaykumar, Maria de Fatima Mattiello-Francisco, Eliane Martins, and Odnei Cuesta Lopes. 2006. A Practical Approach for Automated Test Case Generation using Statecharts. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02*, 183-188. IEEE Computer Society.

Scacchi, Walt. 2001. Process Models in Software Engineering. In *Encyclopedia of Software Engineering, 2nd Edition*. John Wiley & Sons Inc. New York, December.

Segall, Z., D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. 1988. FIAT-fault injection based automated testing environment. In , *Eighteenth International Symposium on Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers*, 102-107. IEEE, June 27. doi:10.1109/FTCS.1988.5306.

Seung Hak Kuk, and Hyeon Soo Kim. 2009. Robustness testing framework for Web services composition. In *IEEE Asia-Pacific Services Computing Conference (APSCC 2009)*, 319-324. IEEE, December 7. doi:10.1109/APSCC.2009.5394106.

Sha, Lui, Tarek Abdelzaher, Karl-Erik årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. 2004. "Real Time Scheduling Theory: A Historical Perspective." *Real-Time Systems* 28 (2) (November 1): 101-155. doi:10.1023/B:TIME.0000045315.61234.1e.

Shelton, C.P., P. Koopman, and K. Devale. 2000. Robustness testing of the Microsoft Win32 API. In *International Conference on Dependable Systems and Networks (DSN 2000)*, 261-270.

Shun-Feng Su, Chan-Ben Lin, and Yen-Tseng Hsu. 2002. "A high precision global prediction approach based on local prediction approaches." *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 32 (4): 416-425. doi:10.1109/TSMCC.2002.806745.

Siblini, R., and N. Mansour. 2005. Testing Web services. In *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*, 135.

Siewiorek, D.P., J.J. Hudak, B.-H. Suh, and Z. Segal. 1993. Development of a benchmark to measure system robustness. In *The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23). Digest of Papers*, 88-97. doi:10.1109/FTCS.1993.627311.

Spring Source. 2008. The Spring framework. http://springframework.org/.

SpringSource. 2010. Aspect Oriented Programming with Spring. http://static.springframework.org/spring/docs/2.5.x/reference/aop.html.

Standard Performance Evaluation Corporation. 2007a. SPECjAppServer2004. http://www.spec.org/jAppServer2004/.

———. 2007b. SPECjms2007. http://www.spec.org/jms2007/.

———. SPECjEnterprise2010. http://www.spec.org/jEnterprise2010/.

Stankovic, J. A. 1988. "Misconceptions about real-time computing: a serious problem fornext-generation systems." *Computer* 21 (10): 10-19.

Stuttard, D., and M. Pinto. 2007. *The web application hacker's handbook: discovering and exploiting security flaws*. John Wiley & Sons, Inc.

Sullivan, M., and R. Chillarege. 1991. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, 2–9.

———. 1992. A comparison of software defects in database management systems andoperating systems. In *Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22). Digest of Papers*, 475-484. IEEE Computer Society, July 8. doi:10.1109/FTCS.1992.243586.

Sun Microsystems Inc. 2004. *Web Services Performance: Comparing JavaTM 2 Enterprise Edition (J2EE platform) and .NET Framework*. http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf.

———. 2007a. Open Message Queue. https://mq.dev.java.net/.

———. 2007b. Java EE 5 Technologies. http://www.oracle.com/technetwork/java/javaee/tech/javaee5-jsp-135162.html.

———. 2010. jax-ws: JAX-WS Reference Implementation. https://jax-ws.dev.java.net/.

Sun Microsystems, Inc. 2002. Java Message Service (JMS). *Sun Developer Network*. March 18. http://java.sun.com/products/jms/.

Surribas, Nicolas. 2011. Wapiti - Web application security auditor. http://wapiti.sourceforge.net/.

Susskraut, M., and C. Fetzer. 2007. Robustness and Security Hardening of COTS Software Libraries. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 61-71. doi:10.1109/DSN.2007.84.

The Open Web Application Security Project (OWASP). 2010. OWASP Top Ten Project: The Ten Most Critical Web Application Security Risks. August. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

———. 2011. Common types of software vulnerabilities. https://www.owasp.org/index.php/Category:Vulnerability.

The Web Application Security Consortium (WASC). 2010. The WASC Threat Classification v2.0. August. http://projects.webappsec.org/Threat-Classification.

Thomas, Stephen, and Laurie Williams. 2007. Using Automated Fix Generation to Secure SQL Statements. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, 9. IEEE Computer Society. http://portal.acm.org/citation.cfm?id=1269069.

Thomas, Stephen, Laurie Williams, and Tao Xie. 2009. "On automated prepared statement generation to remove SQL injection vulnerabilities." *Information and Software Technology* 51 (3) (March): 589-598. doi:10.1016/j.infsof.2008.08.002.

Transaction Processing Performance Council. 2008. TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.3. http://www.tpc.org/tpc_app/.

———. 2011. TPC-C. http://www.tpc.org/tpcc/default.asp.

Trivedi, Kishor, Boudewijn Haverkort, Andy Rindos, and Varsha Mainkar. 1994. Techniques and tools for reliability and performance evaluation: Problems and perspectives. In *Computer Performance Evaluation Modelling Techniques and Tools*, ed. Günter Haring and Gabriele Kotsis, 794:1-24. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. http://dx.doi.org/10.1007/3-540-58021-2_1.

Valeur, Fredrik, Darren Mutz, and Giovanni Vigna. 2005. A Learning-Based Approach to the Detection of SQL Attacks. In *Second International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA 2005)*, 123-140. Vienna, Austria: Springer Berlin / Heidelberg, July 7. doi:10.1007/b137798. http://dx.doi.org/10.1007/11506881_8.

Vieira, Marco. 2005. Dependability Benchmarking for Transactional Systems. PhD, University of Coimbra, July.

Vieira, Marco, Nuno Antunes, and Henrique Madeira. 2009. Using web security scanners to detect vulnerabilities in web services. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN 2009)*, 566-571. IEEE Computer Society. doi:10.1109/DSN.2009.5270294.

Vieira, Marco, Antonio Casimiro Costa, and Henrique Madeira. 2006. Towards Timely ACID Transactions in DBMS. In *12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*, 381-382. doi:10.1109/PRDC.2006.63.

Vlist, Eric van der. 2007. *Schematron*. O'Reilly. http://portal.acm.org/citation.cfm?id=1406342&coll=Portal&dl=GUIDE&CFID=22428427&CFTOKEN=84425081.

Voas, E., F. Charron, G. McGraw, K. Miller, and M. Friedman. 1997. "Predicting how badly 'good' software can behave." *IEEE Software* 14 (4) (August): 73-83. doi:10.1109/52.595959.

W3C. 1999. *XML Path Language (XPath)*. W3C Recommendation. http://www.w3.org/TR/xpath/.

———. 2008a. W3C XML Schema. http://www.w3.org/XML/Schema.

———. 2008b. XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xquery-operators/.

Walnes, Joe. 2011. XStream. http://xstream.codehaus.org/.

Wee Teck Ng, and P. M Chen. 1999. The systematic improvement of fault tolerance in the Rio file cache. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers*, 76-83. IEEE. doi:10.1109/FTCS.1999.781036.

Weyuker, E.J. 1998. "Testing component-based software: a cautionary tale." *Software, IEEE* 15 (5): 54-59.

Whittaker, James A. 2000. "What Is Software Testing? And Why Is It So Hard?" *IEEE Softw.* 17 (1) (January): 70–79. doi:http://dx.doi.org/10.1109/52.819971.

Wieringa, Roel. 1998. "A survey of structured and object-oriented software specification methods and techniques." *ACM Comput. Surv.* 30 (4) (December): 459–527. doi:10.1145/299917.299919.

Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. 2005. WSDL-based automatic test case generation for Web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE 2005)*, 207-212. doi:10.1109/SOSE.2005.43.

XMethods.net. 2008. XMethods. http://xmethods.net/ve2/index.po.

Annex A
wsrbench: Features and Architecture

This annex briefly introduces the main features provided by *wsrbench*, from a technical point-of-view. This tool, publicly available at http://wsrbench.dei.uc.pt, provides a web-based interface that allows users to perform configurations, execute tests, and visualize the results. Anyone can use *wsrbench* as it is free and easy to use. Only a very simple registration and posterior authentication is required.

A.1 Web interface and usage

After registration and authentication three key options are available for users: Configuration, Add WSDL, and My Tests. The **Configuration** option allows several configuration aspects to be defined, such as the user's email, number of finished tests to show on screen, among others.

The **Add WSDL** option allows users to add the WSDL file describing a web service to be tested for robustness. After submitting the WSDL file the user can visualize the set of operations and parameters provided by the service. This is represented in Figure A.1.

Operations not conformant with the WS-I Basic Profile (Ferris et al. 2007) will be grayed out and not tested. This standard is an industry effort to clarify the ambiguous parts of the WSDL specification and is accepted by the main service providers, including Microsoft's .NET framework version 3 (Microsoft Corporation 2012) and Sun Microsystem's Java 6 Web Services stack (JAX-WS) (Sun Microsystems Inc. 2010). By standardizing

the specification, service interoperability is assured and the development of service-based tools (such as wsrbench) is simplified. Nevertheless, external components that perform WSDL analysis are currently being tested so that we can provide increased compatibility with older versions of WSDL documents, or documents that are not compliant with the WS-I Basic Profile standard.

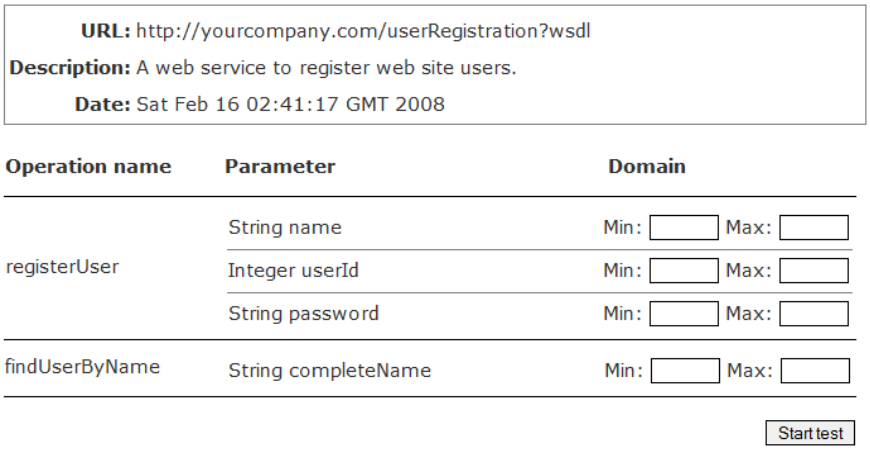


Figure A.1 – Web service parameters domain definition.

As shown in Figure A.1, for each testable operation the user may define the valid values for each parameter (i.e., the domain of the parameter). When these are not defined, the tool considers that the parameter domain is the domain of the corresponding data type. After defining the domain of the parameters the tests can start (by clicking the *Start Test* button). The user will be informed by email when the tests conclude and the results can be analyzed later on.

The **My Tests** option allows the user to visualize the tests previously performed along with information on currently ongoing tests. Detailed results for the concluded tests are available and are presented as shown in Figure A.2.

As we can see, for each operation of a web service, the results for the individual faults applied to each parameter are shown (Figure A.2 shows only a partial set). The first row for each parameter (identified by the word *none*) represents a regular request where no robustness testing is performed (i.e., no faults are injected). Clicking the 'XML' link opens a

popup where more details are provided (see Figure A.3). These include the list of requests sent and the service responses received.

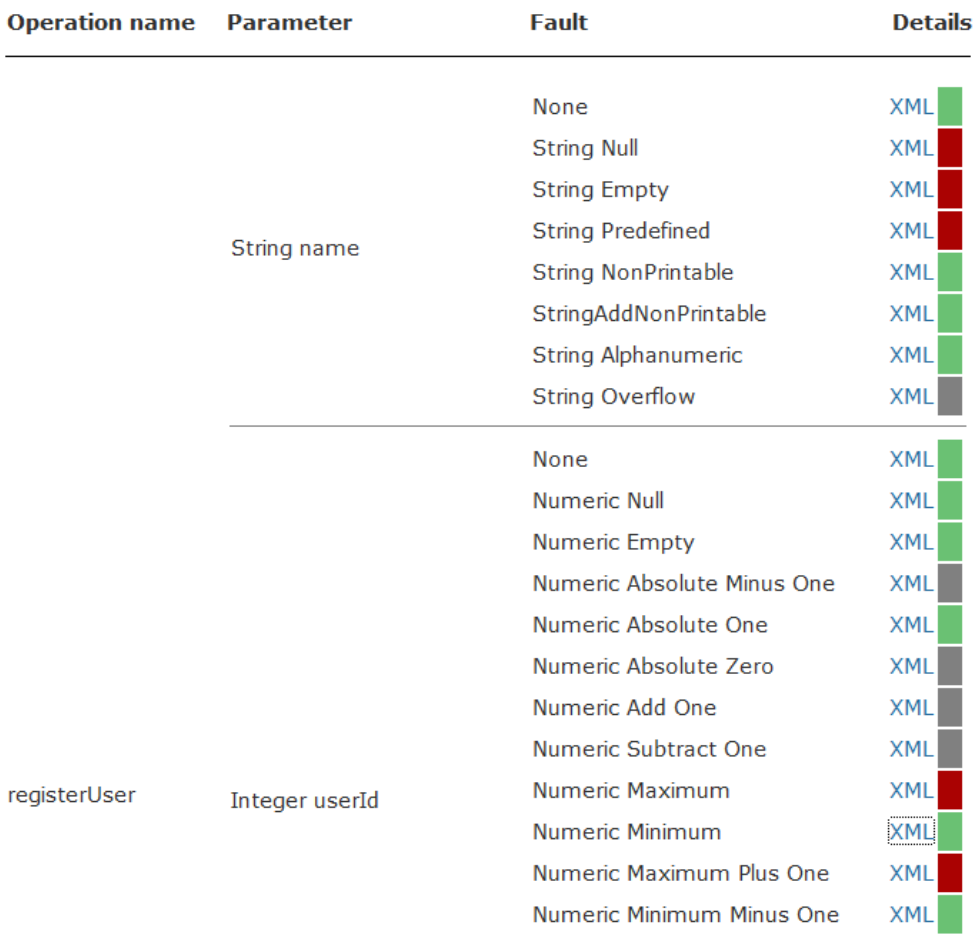


Figure A.2 - Detailed test results for a web service.

The user can then use the three available buttons to mark the service interaction as a robustness problem, a correct interaction (no problem detected), or simply leave it unmarked. This is then reflected on the previous window (shown in Figure A.2) by respectively placing a red, green or gray square in the right side. For instance, in the example shown in Figure A.2, the *StringNull* fault applied to the *name* parameter of the

registerUser operation disclosed a robustness problem. On the other hand, the *StringNonPrintable* fault did not disclose any problem.

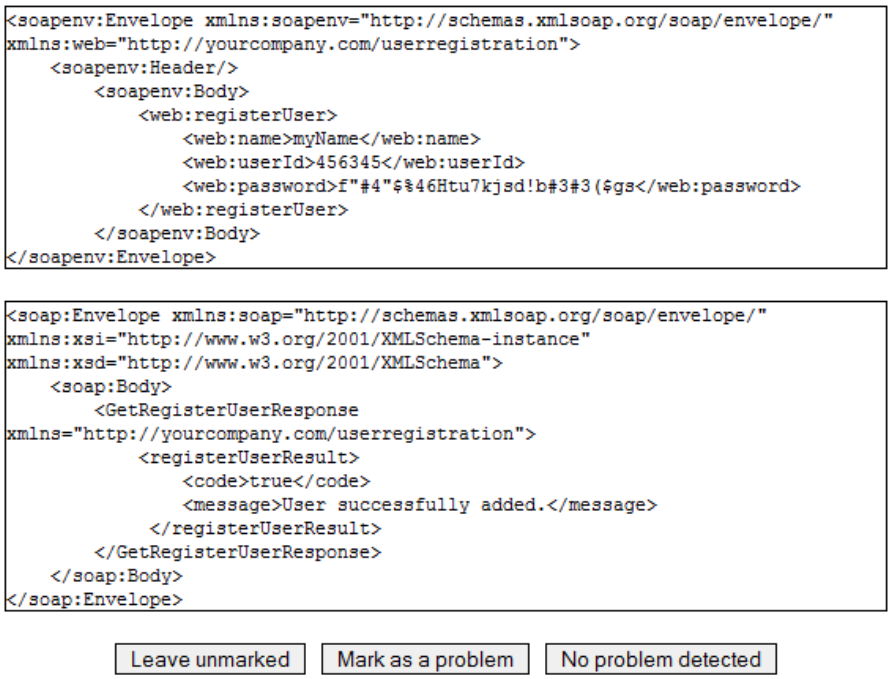


Figure A.3 – A regular request and response.

It is important to emphasize that, after testing a given web service, the tool performs an automatic analysis of the responses obtained in order to distinguish regular replies from replies that reveal robustness problems in the service being tested. However, in some cases the tool is not able to decide if a given response is due to a robustness problem or not. That is why the tool also allows users to perform this analysis manually.

A.2 wsrbench Architecture

The wsrbench application is based in a multi-tier model. In practice, wsrbench is distributed over three tiers: client tier, application server tier, and data persistence tier. Tests execution also involves a fourth tier that corresponds to the external web service being tested, however this is not

considered as part of the application itself. Figure A.4 represents this scenario, including a high level view of the application modules.

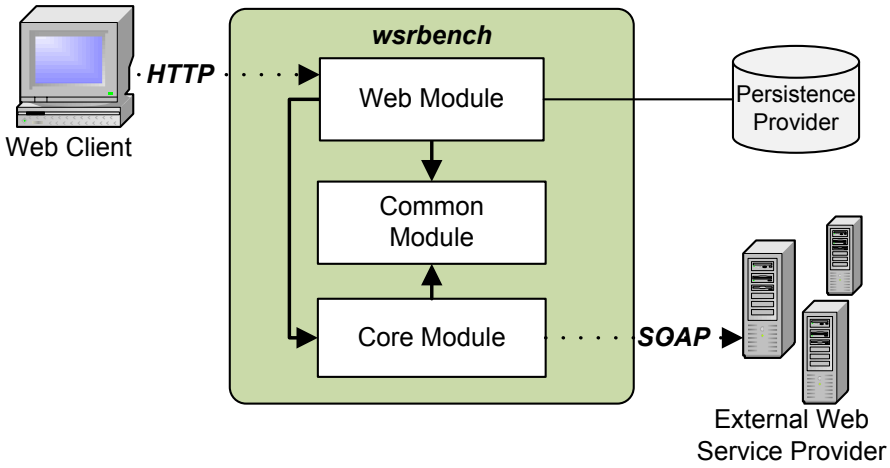


Figure A.4 – wsrbench modules.

As we can see, *wsrbench* is composed of three modules that correspond to a fully layered architecture:

- **Common Module:** includes several domain objects in POJO (Plain Old Java Object) form. It also includes abstraction for functionalities implemented by other modules. The goal was to allow development to take place as independently as possible.
- **Core Module:** provides basic core functionality, which includes SOAP request creation, fault generation, response analysis, etc.
- **Web Module:** includes advanced functionalities such as a complete persistence manager to enable database access, thread creation control (since a web application is virtually accessed by multiple users), and presentation layer code related to the web environment.

Solid arrows in Figure A.4 represent visibility relations. The core module is aware of the common module and the web module is aware of the other

two. In practice, this means that the common is a dependency of the core, and the common and core are dependencies of the web module.

It is important to emphasize that *wsrbench* was built using the latest standards and technologies for Java development, namely:

- The Spring framework (Spring Source 2008) for dependency injection, database access and connection management.
- Spring MVC for model-view-controller web layer implementation, including HTTP request validation and domain objects binding.
- Acegi Security (Interface21 2008) for fine-grained control over the security requirements of the application. It also enables a complete isolation of all security related aspects from the application logic itself.
- Maven 2 (Apache Software Foundation 2008d) for project management tasks. These include the build process, report and documentation creation, and dependency management.
- Subversion (Collabnet 2008) as a version control system.

Figure A.5 represents the internal architectural design of the application. The main components and existing dependencies are described in the following sections.

A.3 Web module design

The web module uses the MVC design pattern (Gamma et al. 1994) implementation offered by Spring, as a basis to provide multi-user support in a web environment. In this sense, the **Controller** component (see Figure A.5) serves as an entry point for HTTP requests. This entry point is represented by the flow arrow between the web client and this component. Arrows in Figure A.5 represent the main application flows that exist during execution. There are other relations and dependencies that are not included for sake of simplicity. For instance, bidirectional flows exist between the Driver Request Handler and all the other components presented at its right in the figure, but representing these would clutter the illustration unnecessarily.

The **MVC** element is divided into 3 key sub-elements: Binder, Validator and Controller. The **Binder** is the facility provided by Spring MVC that performs automatic binding to domain objects (note that more complex binding logic is defined by the programmer). The **Validator** is a set of classes that perform business logic validation over the HTTP request inputs. The **Controller** is responsible for request processing, building a domain model that is rendered by an adequate view, in our case a Java Server Page (JSP) view. This corresponds to the typical behavior of a controller in a MVC implementation.

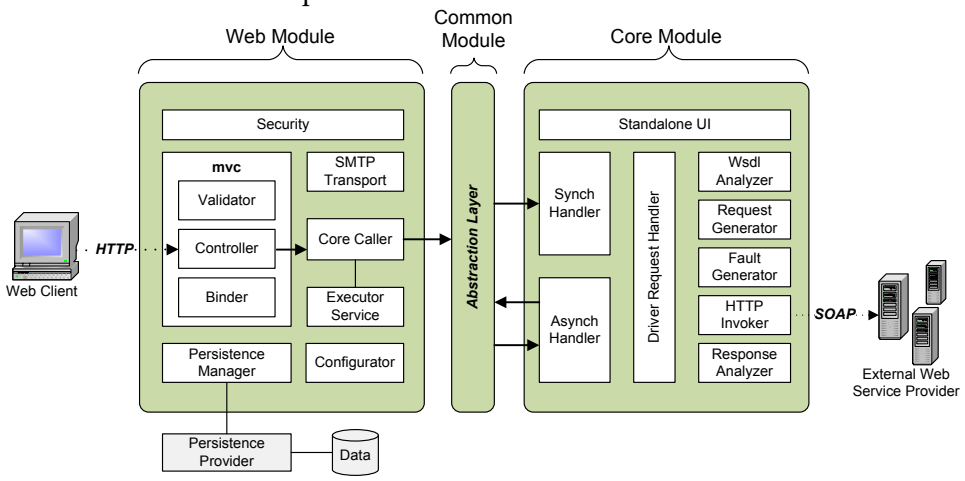


Figure A.5 – Internal architecture of the wsrbench application

The Controller communicates directly with the **Core Caller**, which is the component responsible for the interaction with the Core Module. The **Executor Service** is used to maintain a fixed pool of threads that are used whenever an asynchronous call is required. In practice, there are two ways of internally serving user requests in wsrbench:

- **Synchronous:** used when, at the client interface, a user action requires immediate response from the Core Module (e.g., adding a WSDL file and proceed to the definition of the domain values of the available parameter: see Section 3 for details).
- **Asynchronous**: used when a user action does not depend on an immediate answer from the Core Module. For instance, the user can submit a new test and does not have to wait for an answer before defining another test on a different web service.

Two different objects representing the Core Module can be created by the **Core Caller**, one for synchronous invocation and another for asynchronous invocation. Both are obtained without direct knowledge of the Core Module implementation details by using a Factory class that resides in the Abstraction Layer and creates objects by reflection (McCluskey 1998). The **Executor Service** is only used if the operation is asynchronous.

Note that for asynchronous operations, the Core Module needs a way to contact the Web Module when the operation concludes. A reference to the caller object (that implements a well known interface defined in the **Abstraction Layer**) is provided so that callbacks to the origin can occur in a simple way.

Transversely to the whole application there is a **Security** element, which is basically a configuration of the Acegi Security framework (Interface21 2008). This powerful framework provides declarative security for Spring-based applications and enables a complete separation of all security aspects into XML files. Although we could write security functionality directly into the application code, it is generally a better option to keep security concerns separate from application implementation specificities.

Another key component is the **Persistence Manager**. This is based on Spring's Java Database Connectivity (JDBC) support that eliminates the need for programmatically handling database connections (which are a frequent cause for problems in web applications) and greatly simplifies information insertion, update and retrieval.

A **Configurator** object is created for each user session and contains user specific configuration parameters such as the number of tests to present on screen, the user's email address, etc. Finally, the **SMTP Transport** component is used to send emails to users during registration and whenever a test is concluded.

A.4 Core module design

There are two ways of contacting the Core Module. One is via the **Synch Handler** and the other is through the **Asynch Handler**. These serve the synchronous and asynchronous operation modes mentioned before. Both

make use of a **Driver Request Handler** that is responsible for coordinating the whole robustness testing process.

Testing the robustness of a given web service involves the **Wsdl Analyzer** that is responsible for analyzing a submitted WSDL (Web Service Description Language) document and returns operation and parameter information to the user. After this analysis step, the Driver Request Handler calls the **Request Generator** that generates a list of requests (i.e., the workload) for the specific service being tested. It considers the maximum and minimum domain values provided by the user to generate acceptable random requests for the service.

The **Fault Generator** receives the list of generated requests as input, creates a list of faults and injects those faults into the requests workload. For each parameter of a given type, all of the applicable faults are considered and injected into the request list. Several faulty requests are generated for a single type of fault for a given parameter (but only one fault per request), following the testing procedure described in Section 2.

The **HTTP Invoker** component is responsible for sending each web service request to the provider. This unit also collects the corresponding responses that are the forwarded to the **Response Analyzer** component trough the Driver Request Handler. Response Analyzer is in charge of response interpretation. Each response is analyzed in order to distinguish regular replies from replies that reveal robustness problems in the service being tested. Currently, this module is in a preliminary phase and improvements are being studied in order to improve the automatic response analysis. As mentioned before, the automatic responses classification can be complemented by the user with a manual results analysis.

Annex B
Public Web Services Tested

This section presents detailed information regarding the public web services tested in the experimental evaluation presented earlier in Chapter 7. Table B.I presents a list of endpoints of the 250 services that were tested for robustness (theses services were last accessed in 2010, and may not be available anymore or may currently provide a different interface).

Table B.I – Public services tested for robustness

Service endpoint
http://www.foodcandy.com/WebBlogService.asmx?WSDL
http://ws.strikeiron.com/IPLookup2?WSDL
http://river.sdsc.edu/MODISTS/MODIS.asmx?WSDL
http://sas-d.sermepa.es/TPV_PC/wsdl/SerClsWSPasarelaPINPAD.wsdl
http://www.xignite.com/xhistorical.asmx?WSDL
http://businesssearchnz.co.nz/usercontrols/search/AutoComplete.asmx?WSDL
http://artskart.artsdatabanken.no/AJAXWS/NavnSok.asmx?WSDL
http://www.foodcandy.com/WebEventService.asmx?WSDL
http://vbnet.aspweb.cz/cs2vb.asmx?WSDL
http://www.gonow.no/GO/WebService/ContentServices_black.asmx?wsdl
http://orange.3gxxx.co.il/Ivr23gWs/wService.asmx?WSDL
http://www.cartel.ru/DrugstoreExternalWebService/DrugstoreService.asmx?WSDL
http://www.bronzebusiness.com.br/webservices/valida.asmx?WSDL
http://www.blberza.com/services/blse/ticker.asmx?WSDL
http://koordinatuppslag.capitex.se/koordinatuppslag.asmx?WSDL

http://survey.qut.edu.au/Member/api/v68/launch/LaunchManagementService?wsdl
http://www.nanonull.com/TimeService/TimeService.asmx?wsdl
http://www.xignite.com/xfundamentals.asmx?WSDL
http://www.hashnot.com/taxid/taxidws?wsdl
http://services.doppelme.com/PartnerService.asmx?wsdl
http://ser02.2sms.com/WebServices/1.0/SMSService.asmx?WSDL
http://bibshare.dsic.upv.es/BibShareSearchEngine/BibShareSearchEngine.asmx?WSDL
http://automanager.hu/Ws/Listing.asmx?WSDL
http://www.foodcandy.com/WebLicenseService.asmx?WSDL
http://www.webservicex.com/uklocation.asmx?WSDL
http://kbyte.ru/3w/WebServices/MIMEReader.asmx?WSDL
http://bluesurftech.com/_vti_bin/imaging.asmx?wsdl
http://smn2.cna.gob.mx/webservicessmn/service1.asmx?WSDL
http://www.comunique-se.com.br/proveDados.asmx?WSDL
http://www.xignite.com/xSurvey.asmx?wsdl
http://dotnet.jku.at/buch/samples/7/header/HeaderService.asmx?WSDL
http://dev.geoap.jp/geoap_match/geoap_trial.asmx?wsdl
http://b144.co.il/Services/CityService.asmx?WSDL
http://ws.serviceobjects.com/ft/FastTax.asmx?WSDL
http://websvc.dotnetpia.co.kr/DpNameCheck/DpNameCheck.asmx?WSDL
http://www.webxml.com.cn/WebServices/IpAddressSearchWebService.asmx?wsdl
http://opendap.co-ops.nos.noaa.gov/axis/services/Conductivity?wsdl
http://sms-txt.co.uk/bulksms.asmx?WSDL
http://www.webservicex.com/globalweather.asmx?WSDL
http://www.champions.co.nz/shop/products2.asmx?WSDL
http://www.wifi.at/kursbuchbestellung/Kursbuchbestellen.asmx?WSDL
http://kettinki.uku.fi/CCOW/services/ContextData?wsdl
http://ws2.serviceobjects.net/ft/FastTax.asmx?WSDL
https://provisioning.blueface.ie/provisioning.asmx?WSDL
http://webservices.aeroflot.aero/flightstatus.asmx?wsdl
http://webservices.macam.ac.il/MacamMailer/MacamMailer.asmx?wsdl
http://www.ebi.ac.uk/webservices/whatizit/ws?wsdl
https://www.afi.es/CreditScoringWebService/WSSearchEngine.asmx?WSDL
http://adserver.fibertel.com.ar/CheckIp/CheckIp.asmx?WSDL
http://ws2.serviceobjects.net/fw/fastweather.asmx?wsdl
http://www.webservicex.net/country.asmx?wsdl
http://metalmaker.net/metalmaker.asmx?WSDL

http://ws.strikeiron.com/CorteraCreditPulse?WSDL
https://fx.support.bankmeridian.com/support.asmx?wsdl
http://teletipo.europapress.es/wsepNoticias2006/Noticias.asmx?WSDL
http://hcms.il.wroc.pl/mz/ws/CalcHelper.asmx?WSDL
http://ws.xwebservices.com/XWebBlog/XWebBlog.asmx?wsdl
http://ewave.no-ip.com/EcallWS/CinemaSinchronization.asmx?WSDL
http://www.dis.h.u-tokyo.ac.jp/webservices/byomeisearch.asmx?wsdl
http://www.webservicex.net/medicareSupplier.wsdl
http://interfaces.bercoexpress.co.za/BercoServices/BercoWarehouse.asmx?wsdl=0
http://sdb.amazonaws.com/doc/2007-11-07/AmazonSimpleDB.wsdl
http://www.hotellinx.com/WebServiceXmlInterface/HotellinxXmlService.asmx?WSDL
http://bearmini.net/LivedoorWeatherHacks.asmx?WSDL
http://hipagservice.bisinter.net/HipagServices.asmx?WSDL
http://www.comunique-se.com.br/webDistribuidor.asmx?WSDL
http://www.collinarevalcerrina.it/KnoS/ws/Calendar.asmx?WSDL
http://www.pulmonary-rehabilitation.com.cn/WebService/GetDoctors.asmx?WSDL
http://www.webservicex.com/ValidateEmail.asmx?WSDL
http://aspalliance.com/quickstart/aspplus/samples/services/MathService/VB/MathService.asmx?wsdl
http://services1.pharmx.com.au/order/supplierlist.asmx?WSDL
http://voservices.net/Cosmology/ws_v1_0/Distance.asmx?wsdl
http://kbyte.ru/_ajaxServices/mainKbyteDotRuServices/KbyteDotRu.asmx?WSDL
http://www.xignite.com/xholdings.asmx?WSDL
http://www.smsgratisversturen.nl/sendsms.asmx?WSDL
http://allysoft.ru/BScurrency/currency.asmx?WSDL
http://www.eidsvoll.kommune.no/Layout/Controls/GlobalSearchService.asmx?WSDL
http://www.development.ccs.neu.edu/home/gali/project/math.asmx?WSDL
http://www.bangaloreone.gov.in/bOneWS/MgrService.asmx?WSDL
http://www.allysoft.ru/XML/Connector.asmx?WSDL
http://staging.cic.armstrongconsulting.com/IconWebServices/icontest.asmx?WSDL
http://quisque.com/fr/techno/eqImage/eqimage.asmx?WSDL
http://opendap.co-ops.nos.noaa.gov/axis/services/Datums?wsdl
http://www.petermeinl.de/CurrencyConverter/CurrencyConverter.asmx?wsdl
http://info-messenger.de/sms.asmx?wsdl

http://ws.sercultur.pt/rss/ws.asmx?WSDL
http://ws.strikeiron.com/Graphing?WSDL
http://www.expeditiesos.eu/_vti_bin/SpellCheck.asmx?wsdl
http://webservice.genotec.ch/conversion.asmx?WSDL
http://ws.strikeiron.com/SMSAlerts4?WSDL
https://shop.dorma-glas.com/webservices/?wsdl
http://www.sipeaa.it/wset/ServiceET.asmx?wsdl
http://webservices.tardis.gowitest.bisinter.net/XmlService.asmx?WSDL
http://www.roomex.com/services/LocationService.asmx?WSDL
http://www.beesoft.ru/BScurrency/currency.asmx?WSDL
http://www.transport.turne.com.ua/WebServices/Amadeus.asmx?WSDL
http://ws.acrosscommunications.com/mail.asmx?WSDL
http://ewave.no-ip.com/ECallws/BuyerData.asmx?WSDL
http://ws.xwebservices.com/XWebCheckOut/XWebCheckOut.asmx?wsdl
http://netservice.fossware.com/ArticlesSecure.asmx?WSDL
http://tv.animare.hu/webservice/command.asmx?WSDL
http://interpressfact.net/webservices/getjoke.asmx?WSDL
http://tvservice.ukrtechnologies.com/Service.asmx?WSDL
http://ws.strikeiron.com/ReversePhoneAddressLookup?WSDL
http://www.webservicex.net/braille.asmx?wsdl
http://www.turne.com.ua/WebServices/HotelClassCategory.asmx?WSDL
http://quisque.com/fr/chasses/crypto/cesar.asmx?WSDL
http://ws.serviceobjects.com/yp/YellowPages.asmx?WSDL
https://www.burwebxml.bfs.admin.ch/mutabox/default.asmx?WSDL
http://www.exactmobile.co.za/interactive/interactivewebservice.asmx?wsdl
http://ws.serviceobjects.com/av/AddressValidate.asmx?WSDL
http://www.checkexpress.com.br/ws_office_bacen/registration.asmx?WSDL
http://soap.cpp.com/cppsoaprequests/cppincomingrequest.asmx?WSDL
http://www.morgenstern.com.tw/TBPWin/WS_GetTbps.asmx?WSDL
http://webservices.aeroflot.aero/flightSearch.asmx?wsdl
http://www.xignite.com/xrealtime.asmx?WSDL
http://www.limerickcorp.ie/WebServices/MediaNotices/MediaNotices.asmx?WSDL
http://netpub.cstudies.ubc.ca/dotnet/webservices/chineseastrology.asmx?WSDL
http://www.cp.gov.tw/SEWebApplication/AAMediator.asmx?WSDL
http://webservices.gama-system.com/exchangerates.asmx?WSDL
http://creed.vera.net/Autentica/autentica.asmx?WSDL
http://wslite.strikeiron.com/phonenumberinfolite01/PhoneNumberInfoLite.asmx?WSDL

http://quisque.com/fr/chasses/blasons/search.asmx?WSDL
http://ict1.tbm.tudelft.nl:81/spm4341/hazmat.asmx?wsdl
http://zefix.admin.ch/webservices/zefixSvc/ZefixSvc.asmx?WSDL
http://www.l2009.dk/API/WSAPI_sublayouts/udvservice.asmx?WSDL
http://opendap.co-ops.nos.noaa.gov/axis/services/WaterTemperature?wsdl
http://wiki.vanguardsw.com/bin/ws.dsb?wsdl/Operations/Manufacturing/Manufacturing%20Run%20Cost%20Simulation
https://api.channeladvisor.com/ChannelAdvisorAPI/v1/TaxService.asmx?WSDL
http://galaxy.sagadc.com/Galaxy/wsdl
http://www.novasoftware.se/schedulewebservice/service.asmx?WSDL
http://www.deeptraining.com/webservices/weather.asmx?wsdl
http://www.html2xml.nl/Services/Stocks/Version1/StockServices.asmx?wsdl
http://epc.volvopenta.se/www/robot/RobotService.asmx?WSDL
http://www.looneo.fr/WebServices/ExistsMember.asmx?WSDL
http://www.xignite.com/xreleases.asmx?WSDL
http://www.foodcandy.com/WebTagWordService.asmx?WSDL
http://webservices.tvcabo.pt/epg/epg.asmx?wsdl
http://kettinki.uku.fi/CCOW/services/AuthenticationRepository?wsdl
https://services.nordicedge.se/smsondemandws/services/SMSOndemandWS?wsdl
http://www.xignite.com/xInvestorRelations.asmx?WSDL
http://ws.serviceobjects.com/pe/PhoneExchange.asmx?WSDL
http://webservices.gama-system.com/searchvecercom.asmx?WSDL
http://webservices.ecircle-ag.com/rpc?wsdl
http://www.gama-system.com/webservices/searchnajdisi.asmx?wsdl
http://wslite.strikeiron.com/WebServicesSearchLite01/WebServicesSearchLite.asmx?WSDL
http://ws.strikeiron.com/ZIPInfo4?WSDL
http://www.item.no/ws/nummernd7.nsf/itemNumber?WSDL
http://62.254.234.4/hoseasonspartnerapi/hoseasonspartnerapi.asmx?wsdl
http://ws.xwebservices.com/XWeb1003/XWeb1003.asmx?wsdl
http://ec.europa.eu/taxation_customs/vies/api/checkVatPort?wsdl
http://www.xignite.com/xrates.asmx?WSDL
http://jpv.is/WebServices/AutoCompleteService.asmx?WSDL
http://www.xignite.com/xquotes.asmx?WSDL
http://b144.co.il/Services/CategoryService.asmx?WSDL
http://webservices.affili.net/Publisher/Account.asmx?WSDL
http://www.exporttechnologies.com/webservices/currencyservice/currencyservice.asmx?wsdl

http://kettinki.uku.fi/CertificateService/services/EmailCertificateService?wsdl
http://voservices.net/NED/ws_v1_0/NED.asmx?wsdl
http://mathertel.de/AJAXEngine/S02_AJAXCoreSamples/OrteLookup.asmx?WSDL
http://www.webservicex.com/stockquote.asmx?WSDL
http://webconnect.akbartravelsonline.com/service.asmx?WSDL
http://www.holidaywebservice.com/Holidays/US/Dates/USHolidayDates.asmx?WSDL
http://interfaces.bercoexpress.co.za/BercoServices/BercoService.asmx?wsdl
http://opendap.co-ops.nos.noaa.gov/axis/services/WaterLevelVerifiedHourly?wsdl
http://ws.xwebservices.com/XWebForum/XWebForum.asmx?wsdl
http://www.mensajetexto.com/sendsms.asmx?WSDL
http://www.abn.business.gov.au/abrxmlpubsub/ABRXMLPubSub.asmx?WSDL
http://ws.strikeiron.com/SpeedTaxSalesTax?WSDL
http://opendap.co-ops.nos.noaa.gov/axis/services/HarmonicConstituents?wsdl
http://ewave.no-ip.com/ECallws/CinemaTransaction.asmx?wsdl
http://netservice.fossware.com/Articles.asmx?wsdl
http://www.chemspider.com/Search.asmx?WSDL
http://www.iwebmethod.net/icd1.0/icd.asmx?wsdl
http://ws.strikeiron.com/MapQuestDrivingDirections2?WSDL
http://office.microsoft.com/research/query.asmx?wsdl
http://services1.pharmx.com.au/order/createorder.asmx?WSDL
http://ws.databusiness.cl/ws_bic.asmx?WSDL
http://wslite.strikeiron.com/iplocationlite01/IPLocationLite.asmx?WSDL
http://ws.strikeiron.com/WebDomainBusIntel2?WSDL
https://services.nordicedge.se/otpondemandws/services/OTPOnDemand?wsdl
http://smsmeddelande.com/sendsms.asmx?WSDL
http://opendap.co-ops.nos.noaa.gov/axis/services/WaterLevelVerifiedSixMin?wsdl
http://donau.hiof.no/borres/dn/service1/ServiceSonette.asmx?WSDL
http://kbyte.ru/webservice/codedump.asmx?WSDL
http://ws.strikeiron.com/ForeignExchangeRate?WSDL
http://dstts.co.kr/product/WebService.asmx?WSDL
http://ewave.no-ip.com/ECallws/StadiumTransaction.asmx?WSDL
http://ewave.no-ip.com/ECallws/StadiumData.asmx?WSDL
http://autoklub.jutarnji.hr/Modules/Cars/CarWS.asmx?WSDL
http://www.l2009.dk/API/WSAPI_sublayouts/newsservice.asmx?WSDL
http://netservice.fossware.com/ForumSecure.asmx?wsdl
http://www.softdata.cl/wslme/lme.asmx?WSDL

http://www.webservicex.com/sendsmsworld.asmx?WSDL
http://hm.comprafacil.pt/SIBSClickTeste/webservice/clicksmsV4.asmx?WSDL
http://integration.ecollege.com/developers/docs/wsdl/Gradebook.wsdl
http://nestws.aspweb.cz/nestws.asmx?WSDL
http://www.geonet.es/medioambiente/WebService.asmx?WSDL
http://www.vrllogistics.in/webs/ws/vrlbookingservice.asmx?WSDL
http://ferieogfritid.sas.no/GO/WebService/ContentServices_white.asmx?wsdl
http://epc.volvopenta.se/www/robot/PckExportService.asmx?WSDL
http://mathertel.de/AJAXEngine/S01_AsyncSamples/CalcService.asmx?WSDL
http://www.looneo.fr/WebServices/TrackingService.asmx?WSDL
http://www.mytaratata.com/Artist.asmx?WSDL
http://artskart.artsdatabanken.no/AJAXWS/GUItjenester.asmx?WSDL
http://ws.strikeiron.com/GaleGroupBusinessInformation?WSDL
http://www.burson-marsteller.com/_vti_bin/imaging.asmx?wsdl
https://smsc.vianett.no/smshttp/Service.asmx?WSDL
http://autoklub.jutarnji.hr/Modules/Cars/InsurancePriceGet.asmx?WSDL
http://www.webservicex.net/OFACSDN.WSDL
http://opendap.co-ops.nos.noaa.gov/axis/services/Currents?wsdl
http://dotnet.jku.at/csbook/solutions/19/BookStoreService.asmx?WSDL
http://slk.sulam.org.il/k12/misgav/News/C1/_vti_bin/BusinessDataCatalog.asmx?wsdl
http://www.carwale.com/webservices/carvalues.asmx?WSDL
http://www.sipeaa.it/rain/CRA.clima.webservices.rain.asmx?wsdl
http://www.xignite.com/xemerging.asmx?WSDL
http://wwwacc.ehealth.fgov.be/codage_1_0/codage?WSDL
http://www.almadar.co.il/primsurvey.asmx?WSDL
http://www.xignite.com/xinsider.asmx?WSDL
http://www.agrobase.ru/ActivationService.asmx?WSDL
http://www.limerickcorp.ie/webservices/planningapps/planningapps.asmx?WSDL
http://www.dis.h.u-tokyo.ac.jp/webservices/dpcsearch.asmx?WSDL
http://ws2.serviceobjects.net/pa/phoneappend.asmx?wsdl
http://ws.strikeiron.com/GaleGroupBusinessIntelligence?WSDL
http://people.epfl.ch/wsdl/PeopleWS.wsdl
http://zeta-software.de/Translator/TranslationService.asmx?wsdl
http://plda.fgov.be/files/u1/b2b.wsdl
http://ws.strikeiron.com/IVRVoiceNotification?WSDL
http://www.xignite.com/xnews.asmx?WSDL
http://www.airside-nissan.com/web_services/ajaxUtilities.asmx?WSDL
http://www.kirupafx.com/WebService/TopMovies.asmx?WSDL

http://ewave.no-ip.com/ECallws/StadiumSinchronization.asmx?WSDL
http://www.sporttv.pt/WSSporttv/WSSporttv.asmx?wsdl
http://www.taavonportal.ir/OstanShahrestan.asmx?WSDL
http://www.webservicex.com/isbn.asmx?WSDL
http://wslite.strikeiron.com/ForeignExchangeLite01/HistoricForeignExchangeLite.asmx?WSDL
http://srvcidasc06.cidasc.sc.gov.br/appweb/webservices/wsappweb.asmx?wsdl
http://opendap.co-ops.nos.noaa.gov/axis/services/Predictions?wsdl
http://interfaces.bercoexpress.co.za/BercoServices/PostalcodeService.asmx?wsdl=0
http://netservices.sapo.pt/adsl/adsl.asmx?wsdl
http://www.webservicex.net/whois.asmx?WSDL
http://www.looneo.fr/WebServices/FullTextSearch.asmx?WSDL
http://sws.sercultur.pt/ews.asmx?WSDL
http://www.worldtime-clock.com/services/webclock?wsdl
http://ditdevsrv4.epfl.ch/cff/SSO.wsdl
http://icon.cofacecentraleurope.com/IconWebServices/icon.asmx?wsdl
http://netservice.fossware.com/Forum.asmx?wsdl
http://ict1.tbm.tudelft.nl:81/spm4341/inventory.asmx?WSDL
http://epc.volvopenta.se/www/robot/SnlService.asmx?wsdl
http://ewave.no-ip.com/ECallws/CinemaData.asmx?WSDL

Table B.II presents the distribution of the tested services by provider. The left column represents a provider domain and right column holds the corresponding number of services tested that are deployed in that particular domain.

Table B.II – List of the services providers.

Provider	Count
strikeiron.com	17
xignite.com	12
noaa.gov	8
no-ip.com	7
webservicex.com	7
webservicex.net	5
foodcandy.com	4
fossware.com	4
serviceobjects.net	4

bercoexpress.co.za	3
gama-system.com	3
kbyte.ru	3
looneo.fr	3
quisque.com	3
serviceobjects.com	3
uku.fi	3
volvopenta.se	3
xwebservices.com	3
admin.ch	2
aeroflot.aero	2
allysoft.ru	2
artsdatabanken.no	2
aspweb.cz	2
b144.co.il	2
bisinter.net	2
comunique-se.com.br	2
epfl.ch	2
fgov.be	2
jku.at	2
jutarnji.hr	2
l2009.dk	2
limerickcorp.ie	2
mathertel.de	2
nordicedge.se	2
pharmx.com.au	2
sercultur.pt	2
sipeaa.it	2
tudelft.nl	2
turne.com.ua	2
u-tokyo.ac.jp	2
voservices.net	2
2sms.com	1
3gxxx.co.il	1
acrosscommunications.com	1
affili.net	1
afi.es	1
agrobase.ru	1
airside-nissan.com	1
akbartravelsonline.com	1

almadar.co.il	1
amazonaws.com	1
animare.hu	1
armstrongconsulting.com	1
aspalliance.com	1
automanager.hu	1
awi.de	1
b-es.de	1
bangaloreone.gov.in	1
bankmeridian.com	1
bearmini.net	1
beesoft.ru	1
blberza.com	1
blueface.ie	1
bluesurftech.com	1
bronzebusiness.com.br	1
burson-marsteller.com	1
business.gov.au	1
businesssearchnz.co.nz	1
capitex.se	1
cartel.ru	1
carwale.com	1
champions.co.nz	1
channeladvisor.com	1
checkexpress.com.br	1
chemspider.com	1
cna.gob.mx	1
cofacecentraleurope.com	1
collinarevalcerrina.it	1
comprafacil.pt	1
cp.gov.tw	1
cpp.com	1
databusiness.cl	1
deeptraining.com	1
doppelme.com	1
dorma-glas.com	1
dotnetpia.co.kr	1
dstts.co.kr	1
ebi.ac.uk	1
ecircle-ag.com	1

ecollege.com	1
eidsvoll.kommune.no	1
europa.eu	1
europapress.es	1
exactmobile.co.za	1
expeditiesos.eu	1
exporttechnologies.com	1
fibertel.com.ar	1
genotec.ch	1
geoap.jp	1
geonet.es	1
gonow.no	1
hashnot.com	1
hiof.no	1
holidaywebservice.com	1
hoseasons.co.uk	1
hotellinx.com	1
html2xml.nl	1
il.wroc.pl	1
info-messenger.de	1
interpressfact.net	1
item.no	1
iwebmethod.net	1
jpv.is	1
kirupafx.com	1
macam.ac.il	1
mensajetexto.com	1
metalmaker.net	1
microsoft.com	1
morgenstern.com.tw	1
mytaratata.com	1
nanonull.com	1
neu.edu	1
novasoftware.se	1
petermeinl.de	1
pulmonary-rehabilitation.com.cn	1
qut.edu.au	1
roomex.com	1
sagadc.com	1
sapo.pt	1

sc.gov.br	1
sdsc.edu	1
sermepa.es	1
sms-txt.co.uk	1
smsgratisversturen.nl	1
smsmeddelande.com	1
softdata.cl	1
sporttv.pt	1
sulam.org.il	1
taavonportal.ir	1
tvcabo.pt	1
ubc.ca	1
ukrtechnologies.com	1
upv.es	1
vanguardsw.com	1
vera.net	1
vianett.no	1
vrllogistics.in	1
webxml.com.cn	1
worldtime-clock.com	1
zeta-software.de	1

Table B.III presents the distribution of the tested services by country. The left column holds country names and the right column presents a count of services tested found deployed in that particular country.

Table B.III – Country distribution of the services tested.

Country	Count
United States	84
Germany	14
Russian Federation	11
Argentina	8
France	7
Norway	7
Sweden	7
Austria	6
Ireland	6
Israel	6

Portugal	6
Brazil	5
Canada	5
Finland	5
Japan	5
Spain	5
Switzerland	5
Australia	4
India	4
Netherlands	4
South Africa	4
Hungary	3
Serbia and Montenegro	3
Slovenia	3
Ukraine	3
United Kingdom	3
Belgium	2
Chile	2
China	2
Croatia	2
Czech Republic	2
Denmark	2
Italy	2
Mexico	2
New Zealand	2
Republic of Korea	2
Taiwan	2
Bosnia and Herzegovina	1
Iceland	1
Islamic Republic of Iran	1
Luxembourg	1
Poland	1

Table B.IV lists all server platforms used by the services. The left column contains the platform designation and the right column contains a count of the services tested that use that particular platform.

Table B.IV – List of the server platforms supporting the services under test.

Server	Count
Microsoft-IIS/6.0	161
Unknown	23
Microsoft-IIS/5.0	20
Microsoft-IIS/7.0	18
Apache-Coyote/1.1	11
Apache	3
Apache/2.2.3 (Debian) mod_python/3.2.10	3
Apache/2.0.52 (CentOS)	2
Apache/2.0.52 (Red Hat)	1
Microsoft-IIS/6.0	1
Apache/1.3.28 (Unix) PHP/4.3.4	1
Apache/2.0.52 (Red Hat)	1
Apache/2.2	1
IBM_HTTP_Server	1
Lotus-Domino	1
Vanguard Server/5.1.4	1
WebLogic Server 8.1 SP3	1

Table B.V lists the software stacks used to support the web services tested. The left column contains the web services stack (or framework) used and the right column contains a count of the services tested that have been deployed using that particular software stack.

Table B.V – Web services stacks supporting the tested services.

Web Service Stack	Count
MS .NET	217
Apache Axis	28
.NET	1
Visual Dataflex	1
JAX-WS	1
N&A	1
Soap:Net	1