Model based testing of service oriented applications

André Takeshi Endo

Model based testing of service oriented applications

André Takeshi Endo

Advisor: **Prof. Dr. Adenilso da Silva Simão**

Doctoral dissertation submitted to the *Instituto de Ciências Matemáticas e de Computação* - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*.

USP – São Carlos
June 2013

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

```
        Endo, André Takeshi
E56m       Model based testing of service oriented
        applications / André Takeshi Endo; orientador
        Adenilso da Silva Simão. -- São Carlos, 2013.
           143 p.


           Tese (Doutorado - Programa de Pós-Graduação em
        Ciências de Computação e Matemática Computacional) --
        Instituto de Ciências Matemáticas e de Computação,
        Universidade de São Paulo, 2013.


           1. model based testing. 2. service oriented
        architecture. 3. web services. 4. finite state
        machine. 5. test case generation. I. Simão, Adenilso
        da Silva, orient. II. Título.
```

Teste baseado em modelo de aplicações orientadas a serviço

André Takeshi Endo

Orientador: **Prof. Dr. Adenilso da Silva Simão**

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*.

USP – São Carlos
Junho de 2013

Acknowledgments

Numerous people contributed to achieve this dissertation. Without them, I would not be able to carry out this work during four years of study. Particularly, I would like to thank:

God for giving me opportunity, skills, and desire to achieve this work.

My family, in particular, my parents Angelina and José and my brother Gustavo. Words are not enough to express how much I love, admire, and feel proud of you. This dissertation is dedicated to you!

My girlfriend Flávia for loving ♡, understanding, and inspiring me along four years. This work is as mine as yours.

Prof. Adenilso da Silva Simão for the friendship, knowledge, and trustworthiness in advising this work.

My friends that worked in LabES from 2006 to 2013, for the partnership, funny talks, and the coffee time. They are so many and I do not want to make a huge mistake leaving somebody out. Therefore, I am not risking listing all of them here. Special thanks for reviewing this dissertation to: prof. Angela Giampedro, Fabiano Ferrari, Faimison Porto, Marcelo Eler, Marco Graciotto, Paulo Nardi, Rafael Oliveira, Sofia Costa, and Vinicius Durelli. *Yeah yeah ☺!*

The professors of ICMC, in particular, Simone Souza, Paulo Souza, Masiero and Maldonado.

The people that welcomed me with open arms in Universität Paderborn, Germany. In particular, Prof. Fevzi Belli, Michael Linschulte, Mutlu Beyazit, Sascha Padberg, and Benedikt Krüger. Vielen Dank!

The colleagues of PUCRS that received me in Porto Alegre during that cold July tchê. In particular, Prof. Avelino Zorzo, Prof. Flávio Oliveira, Maicon Silveira, Leandro Costa and Elder Rodrigues. I also appreciate the cooperation of Dell Brazil.

The examining committee (Adenilso Simão, Eliane Martins, Itana Gimenes, Rohit Gheyi, and Silvia Vergilio) for valuable counsel on both research and writing.

The ICMC and its employees for the continuous support.

All my friends, in particular, the “malacabados”, the seinenkai of Álvares Machado, and my flatmates.

Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) (grant 2009/01486-9) for financially supporting this doctoral project and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) (grant PDEE 0332-11-9) for funding the time I spent abroad.

Agradecimentos

Inúmeras pessoas contribuíram de diversas formas para a realização desta tese. Sem elas eu não teria sido capaz de terminar esse trabalho ao longo de quatro anos de estudos. Particularmente, eu gostaria de agradecer:

A Deus, por me dar oportunidade, capacidade e vontade para realizar este trabalho.

A toda minha família, em especial, meus pais Angelina e José e meu irmão Gustavo. Palavras não são suficientes para expressar o quanto amo, admiro e tenho orgulho de vocês. Esta tese é dedicada a vocês!

A minha namorada Flávia, pelo amor ♡, compreensão e inspiração durante esses quatro anos de caminhada. Este trabalho é tão meu quanto seu.

Ao Prof. Adenilso da Silva Simão, pela amizade, conhecimento e confiança na orientação deste trabalho.

A todos meus amigos que passaram pelo LabES de 2006 a 2013, pelo companheirismo, bate-bapos descontraídos e pela hora do café. Foram tantos que nem me arrisco a listar aqui e cometer o grave erro de esquecer alguém. Um agradecimento especial pela revisão do texto para: profa. Angela Giampedro, Fabiano Ferrari, Faimison Porto, Marcelo Eler, Marco Graciotto, Paulo Nardi, Rafael Oliveira, Sofia Costa e Vinicius Durelli. *Yeah yeah ☺!*

Aos professores do ICMC, em especial, Simone Souza, Paulo Souza, Masiero e Maldonado.

Às pessoas que me receberam de braços abertos na Universität Paderborn, Alemanha. Em especial, Prof. Fevzi Belli, Michael Linschulte, Mutlu Beyazit, Sascha Padberg e Benedikt Krüger. Vielen Dank!

Aos colegas da PUCRS que me receberam em Porto Alegre no gelado mês de julho tchê. Em especial, Prof. Avelino Zorzo, Prof. Flávio Oliveira, Maicon Silveira, Leandro Costa e Elder Rodrigues. Agradeço também à Dell Brasil pela cooperação.

A comissão julgadora desta tese (Adenilso Simão, Eliane Martins, Itana Gimenes, Rohit Gheyi e Silvia Vergilio) pelos valiosos conselhos em ambos pesquisa e escrita.

Ao ICMC, bem como seus funcionários, pelo constante auxílio.

A todos os meus amigos, em especial aos “malacabados”, ao seinenkai Álvares Machado e aos companheiros de república.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) (processo 2009/01486-9) pelo apoio financeiro no decorrer do doutorado e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) (processo PDEE 0332-11-9) pelo financiamento do estágio no exterior.

Declaration of Original Authorship and List of Publications

I confirm that this dissertation has not been submitted in support of an application for another degree at this or any other teaching or research institution. It is the result of my own work and the use of all material from other sources has been properly and fully acknowledged. Research done in collaboration is also clearly indicated.

Excerpts of this dissertation have been either published or submitted for the appreciation of editorial boards of journals, conferences and workshops, according to the list of publications presented as follows. My contributions to each publication are listed as well.

Journal Papers

- Belli, F.; ***Endo, A. T.***; Linschulte, M.; Simao, A.: *"A Holistic Approach to Model-based Testing of Web Service Compositions" (Belli et al., 2013).*
  - **Journal:** Software: Practice and Experience.
  - **DOI:** http://dx.doi.org/10.1002/spe.2161
  - **Level of Contribution:** High – The PhD candidate is one of the main investigators and conducted the work together with his contributors.

- ***Endo, A. T.***; Simao, A.: *"Evaluating Test Suite Characteristics, Cost, and Effectiveness of FSM-based Testing Methods"* (Endo and Simao, 2013).
  - **Journal:** Information and Software Technology.
  - **DOI:** http://dx.doi.org/10.1016/j.infsof.2013.01.001
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisor.

Conference and Workshop Papers

- ***Endo, A. T.***; Simao, A.: *"A Systematic Review on Formal Testing Approaches for Web Services"* (Endo and Simao, 2010b).

  - **Event:** 4th Brazilian Workshop on Systematic and Automated Software Testing (SAST' 10).
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisor.

- ***Endo, A. T.***; Linschulte, M.; Simao, A.; Souza, S. R. S.: *"Event- and Coverage-Based Testing of Web Services"* (Endo et al., 2010).

  - **Event:** 2nd Workshop on Model-Based Verification & Validation From Research to Practice (MVV'10).
  - **DOI:** http://dx.doi.org/10.1109/SSIRI-C.2010.24
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his contributors.

- ***Endo, A. T.***; Simao, A.: *"Model-Based Testing of Service-Oriented Applications via State Models"* (Endo and Simao, 2011).

  - **Event:** 8th IEEE International Conference on Services Computing (SCC'11).
  - **DOI:** http://dx.doi.org/10.1109/SCC.2011.77
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisor.

- Belli, F.; ***Endo, A. T.***; Linschulte, M.; Simao, A.: *"Model-based Testing of Web Service Compositions"* (Belli et al., 2011a).

  - **Event:** 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE'11) – selected for the journal special issue.
  - **DOI:** http://dx.doi.org/10.1109/SOSE.2011.6139107
  - **Level of Contribution:** High – The PhD candidate is is one of the main investigators and conducted the work together with his contributors.

- ***Endo, A. T.***; Simao, A.: *"Experimental Comparison of Test Case Generation Methods for Finite State Machines"* (Endo and Simao, 2012a).

  - **Event:** 8th Workshop on Advances in Model Based Testing (A-MOST'12) – ***selected as the best paper***.
  - **DOI:** http://dx.doi.org/10.1109/ICST.2012.140
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisor.

- Capellari, M. L.; Gimenes, I. M. S.; Simao, A.; ***Endo, A. T.***: *"Towards Incremental FSM-based Testing of Software Product Lines"* (Capellari et al., 2012).

  - **Event:** XI Brazilian Symposium on Software Quality (SBQS'12) – ***selected as the best paper in the technical track***.
  - **Level of Contribution:** Medium – The PhD candidate helped in the definition of a testing approach as well as in the paper writing.

Technical Reports

- ***Endo, A. T.***; Simao, A.: *“Formal Testing Approaches for Service-Oriented Architectures and Web Services: a Systematic Review”* (Endo and Simao, 2010a).
  - **Institution:** ICMC – Universidade de São Paulo.
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisor.
- ***Endo, A. T.***; Simao, A.: *“An Experimental Study on Test Suite Characteristics, Cost, and Effectiveness of FSM-based Testing Methods”* (Endo and Simao, 2012b).
  - **Institution:** ICMC – Universidade de São Paulo.
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his advisor.
- ***Endo, A. T.***; Silveira, M. B.; Rodrigues, E. M.; Simao A.; Oliveira, F. M.; Zorzo, A. F.: *“Using Models to Test Web Service-Oriented Applications: an Experience Report”* (Endo et al., 2012).
  - **Institution:** FACIM – Pontifícia Universidade Católica do Rio Grande do Sul.
  - **Level of Contribution:** High – The PhD candidate is the main investigator and conducted the work together with his contributors.

Other Related Publications

- Eler, M. M.; ***Endo, A. T.***; Masiero, P. C.; Delamaro, M. E.; Maldonado, J. C.; Vincenzi, A. M. R.; Chaim, M. L. ; Beder, D. M.: *“JaBUTiService: A Web Service for Structural Testing of Java Programs”* (Eler et al., 2009).
  - **Event:** 33^{rd} Annual IEEE Software Engineering Workshop (SEW’09).
  - **DOI:** http://dx.doi.org/10.1109/SEW.2009.10
  - **Level of Contribution:** High – The PhD candidate developed the proposed Web service and helped in the paper writing.
- Estrella, J. C.; ***Endo, A. T.***; Toyohara, R. K. T.; Santana, R. H. C.; Santana, M. J.; Bruschi, S. M.: *“A Performance Evaluation Study for Web Services Attachments”* (Estrella et al., 2009).
  - **Event:** 7^{th} IEEE International Conference on Web Services (ICWS’09).
  - **DOI:** http://dx.doi.org/10.1109/ICWS.2009.48
  - **Level of Contribution:** Medium – The PhD candidate helped in the experiments planning and implementation, as well as in the paper writing.
- Simao, A.*;* ***Endo, A. T.***: *“Model based Testing”* (short course in Portuguese) (Simao and Endo, 2010).
  - **Event:** Brazilian Conference on Software: Theory and Practice (CBSoft’10).

  - **Level of Contribution:** High – The PhD candidate prepared the slides of the presentation together with his advisor.

- Durelli, V. H. S.; ***Endo, A. T.***; Simao, A.; Delamaro, M. E.: *"Towards Envisaging Software Testing in a Pervasive Computing World"* (Durelli et al., 2012).
  - **Event:** XXVI Brazilian Symposium on Software Engineering (SBES'12) – Special track on Grand Challenges in Software/System Engineering.
  - **DOI:** http://dx.doi.org/10.1109/SBES.2012.21
  - **Level of Contribution:** High – The PhD candidate helped in the proposal of the paper as well as in the writing.

Abstract

SERVICE oriented architecture (SOA) is an architectural style to structure software systems, fostering loose coupling and dynamic integration among the applications. The use of SOA and Web services to develop complex and large business processes demands more formal and systematic testing. In addition, characteristics of this type of software limit the straightforward application of traditional testing techniques. Model-based testing (MBT) is a promising approach to deal with these problems. This dissertation investigates how two modeling techniques, namely Finite State Machine (FSM) and Event Sequence Graph (ESG), can be used to support MBT of service-oriented applications. Both techniques model different aspects and can be applied in a complementary way. Initially, we define an MBT process for service-oriented applications that employs FSMs. Based on previous experience, we propose a model-based approach to test composite services using ESGs. This approach is holistic, once test suites are generated to cover both desired situations (positive testing) and unexpected behaviors (negative testing). Three experimental studies evaluate the proposed approach: (i) a case study, (ii) a cost analysis, and (iii) a study in industry. Testing tools are also presented to support its practical use.

Keywords: model based testing, service oriented architecture, web services, finite state machine, event sequence graph, test case generation, automated tests.

Resumo

A Arquitetura orientada a serviço (SOA) é um estilo arquitetural para estruturar sistemas de software de modo que exista um baixo grau de acoplamento entre as aplicações e essas possam ser facilmente integradas de forma dinâmica. A incorporação de SOA e serviços Web em sistemas que modelam processos de negócios grandes e complexos contribui para a necessidade de testes mais formais e sistemáticos. Além disso, características próprias dessa nova classe de software fazem com que técnicas de teste tradicionais não possam ser diretamente aplicadas. O teste baseado em modelo (TBM) apresenta-se como uma abordagem promissora que busca a resolução desses problemas. Esta tese investiga como duas técnicas de modelagem, Máquina de Estados Finitos (MEF) e Grafo de Sequência de Eventos (GSE), podem ser utilizadas para apoiar o TBM de aplicações orientadas a serviço. Essas técnicas modelam diferentes aspectos e podem ser aplicadas de forma complementar. Inicialmente, é definido um processo de TBM para aplicações orientadas a serviço que emprega MEFs. Com base na experiência adquirida, é proposta uma abordagem baseada em modelo para o teste de serviços compostos usando GSEs. Essa abordagem é holística uma vez que conjuntos de teste são gerados para cobrir tanto situações desejadas (teste positivo) quanto comportamentos inesperados (teste negativo). Três estudos experimentais avaliam a abordagem proposta: (i) um estudo de caso, (ii) uma análise de custo e (ii) um estudo na indústria. Ferramentas de teste também são apresentadas para apoiar o uso prático da abordagem proposta.

Palavras-chave: teste baseado em modelo, arquitetura orientada a serviço, serviços web, máquina de estados finitos, grafo de sequência de eventos, geração de casos de teste, testes automatizados.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

CC	-	Cyclomatic Complexity
CES	-	Complete Event Sequence
CFG	-	Control Flow Graph
CPP	-	Chinese Postman Problem
DT	-	Decision Table
EP, ES	-	Event Pair, Event Sequence
ERunTE	-	Event Runner for Test Execution
ESB	-	Enterprise Service Bus
ESG	-	Event Sequence Graph
ESG4WS	-	Event Sequence Graph for Web Services
ESG4WSC	-	Event Sequence Graph for Web Service Composition
ETA	-	Event Tree Algorithm
FDR	-	Fault Detection Ratio
FSM	-	Finite State Machine
MBT	-	Model-Based Testing
OASIS	-	Organization for the Advancement of Structured Information Standards
OO	-	Object Oriented / Orientation
OWL	-	Ontology Web Language
OWL-S	-	Ontology Web Language for Services
PES	-	Partial Event Sequence
PriFES	-	Private Faulty Event Sequence
PubFES	-	Public Faulty Event Sequence
RDF	-	Resource Description Framework
SLA	-	Service Level Agreement
SOA	-	Service Oriented Architecture
SUT	-	System Under Test
TSD	-	Test Suite Designer
UDDI	-	Universal, Discovery, Description and Integration
V&V	-	Verification and Validation
W3C	-	World Wide Web Consortium
WS-BPEL	-	Web Services Business Process Execution Language
WS-CDL	-	Web Services Choreography Description Language
WSDL	-	Web Services Description Language
XML	-	eXtensible Markup Language

CHAPTER

1 Introduction

Over the past decades, enterprises have evolved from stable, monolithic and centralized hierarchical structures to distributed and dynamically federated organizations. Such federations have been increasingly supported through the integration and composition of business services individually provided by each organization (Di Nitto et al., 2008). Presently, the Information Technology (IT) environments adopted by these companies are mostly heterogeneous, hampering the integration of systems implemented by different technologies. Service-Oriented Architecture (SOA) has been introduced to fill this gap, providing a *de facto* standard that enables communication among those systems. SOA is an architectural style, so that functionalities are decomposed into distinct units (services) to reach loose coupling among the systems (MacKenzie et al., 2006). In a SOA, software capabilities are encapsulated as services, which are well-defined and self-contained modules capable of providing business functionalities independently of states or contexts of other services (Papazoglou and Heuvel, 2007). SOA fosters a collection of principles that include loosely coupled services, high granularity of interfaces, dynamic discovery and binding of services, interoperability, and protocol independence (Erl, 2005; Josuttis, 2007).

Web Services (W3C, 2002) have been used as the main technology to implement the concepts of SOA. They allow applications deployed in different platforms and developed in several languages to communicate with each other through the Internet. This communication is performed with XML protocols that standardize the message format (SOAP), the interface description (WSDL), and the service discovery (UDDI) (W3C, 2002). These three standards represent the first generation of Web services. The evolution of Web services to the second generation occurred by the inclusion of new standards, referred to as WS-* (Erl, 2005). Among them, WS-BPEL and WS-CDL have drawn special attention as standards to describe service compositions. To solve more complex problems, many services can be combined in a collaborative way, following a workflow to create a new service. In this process, the so-called service composition, many services can be combined in a workflow to model and execute complex business processes. The services involved in a service composition are usually called partner services. Service compositions can be developed as either orchestration or choreography (Peltz, 2003). In service orchestration, there is a main entity responsible for coordinating the partner services. Currently, the most widespread language to implement a service orchestration is the Web Service Business Process Execution Language (WS-BPEL) (Jordan et al., 2007). In service choreography, there is no control entity and all partner services work cooperatively to achieve an agreed objective. Among the languages used to describe service choreography, the Web Services Choreography Description Language (WS-CDL) (Kavantzas et al., 2005) is the most used. Service composition is also a service (referred to as composite service) and can be reused by other services. A service that is not a composition is usually called single service.

Software systems developed applying the principles of SOA and using the Web services technologies are usually called service-oriented applications[1]. As it occurs in traditional software, the development of these applications should be conducted in a rigorous and systematic way, aiming at meeting quality standards from both users' and developers' points of view. Along the software development process, many activities of Verification and Validation (V&V) should be performed for quality assurance. Among these activities, testing is the primary activity used in industry to identify faults in the System Under Test (SUT) and verify its conformity with respect to software requirements (Ammann and Offutt, 2008). In general, testing is the process of executing an SUT to find faults (Myers et al., 2004). As it is one of the most costly activities in the development process, theoretical and experimental studies were conducted to identify efficient and effective ways of applying software testing techniques.

A tester commonly creates a mental model of the SUT to guide the tests. When this model is made explicit as an artifact, it can be shared with other team members and used to generate tests. According to Hierons et al. (2009), the use of formal models and specifications can make the testing activity more effective and ease the automation process. In this context, Model-Based Testing (MBT) is an approach to derive test cases from a formal model designed to support the testing activity. An MBT process can be divided into four main steps: *(i)* modeling, in which the tester uses its knowledge of the SUT and test purposes to design a test model; *(ii)* test generation, which consists in generating abstract test cases from the test model; *(iii)* concretization, which fills the gap between the model and the SUT to provide executable test cases; and *(iv)* test execution, in which concrete test cases derived from the model are executed in the SUT. To increase the automation level in these steps, the test model should be syntactically and semantically well-defined through the use of a formal modeling technique. In the context of MBT, two modeling techniques, namely Finite State Machine (FSM) and Event Sequence Graph (ESG), have been particularly investigated.

[1]In this dissertation, this term refers to any type of application that is developed using SOA and Web services, including both single and composite services.

FSM is a formal modeling technique widely adopted due to its simplicity and rigor. Although it has been used mainly in test case generation for protocols (Lee and Yannakakis, 1996), this technique has been successfully applied to other software classes, like reactive systems (Broy et al., 2005) and Web applications (Andrews et al., 2005a). FSM focuses on modeling four elements of the SUT: states, inputs, outputs, and transitions among states. In an FSM, a transition consumes an input and produces an output while connecting states. FSM-based testing has a solid theoretical background and, although it has been studied for over 50 years (Moore, 1956; Gill, 1962), it has recent contributions to the definition and improvement of test methods (Hierons and Ural, 2010; Ipate, 2010; Simao and Petrenko, 2010a). Testing from FSMs has been recognized mainly by the methods capable of generating test suites that, under some assumptions, cover all faults within a given domain. Among such methods are W (Vasilevskii, 1973; Chow, 1978), HSI (Luo et al., 1995), H (Dorofeeva et al., 2005b), SPY (Simao et al., 2009b), and P (Simao and Petrenko, 2010b).

Event-driven models have been used to support verification and testing (van der Aalst, 1999; Belli et al., 2006; Yuan et al., 2011), since events are essential for many different classes of systems, such as Web applications and embedded systems. One of these classes of model is the ESG, originally introduced to test graphical user interfaces (Belli et al., 2006). ESGs use a graph representation so that the events of the SUT are modeled as nodes and their valid orders as edges. ESG modeling is known to be learned in a short period, requires little manual work, and is supported by specific tools (Belli et al., 2006).

1.1 Problem Statement and Justification for the Research

The use of SOA concepts and Web services technologies in IT companies has been constantly growing. In a report called "*SOA Applications Middleware Market Shares and Forecasts Worldwide, 2010-2016*", *Research and Markets*[2] estimates that the global market in SOA generated around 3.5 billion dollars in 2009. Furthermore, according to Gartner Inc.[3], during the *Gartner Symposium/ITxpo: Emerging Trends*, SOA was used in more than 50% of the mission critical applications and business process designed in 2007 around the world and would exceed 80% in 2010. As a consequence of the industry adoption, a growing interest from the academia has also been noticed. Many research events on SOA and Web services, such as IEEE International Conference on Web Services, IEEE International Conference on Services Computing, International Conference on Service Oriented Computing, and IEEE International Conference on Service-Oriented Computing and Applications have been promoted. Journals about this topic have also been published, such as IEEE Transactions on Services Computing, Service Oriented Computing and Applications Journal, and International Journal of Web Services Research.

Engineering high-quality and robust service-oriented applications is essential for the involved enterprises and demands interest from both industry and academia. Therefore, activities of soft-

[2]http://www.researchandmarkets.com/reportinfo.asp?report_id=1212101 – last accessed on 20/02/2013.

[3]http://www.gartner.com/it/page.jsp?id=503864 – last accessed on 20/02/2013.

ware quality assurance have been found in the main service lifecycle models (Andrikopoulos et al., 2010). In the literature on the development process of SOA-based software, testing is one of the activities to assure that requirements are correctly implemented (Erl, 2005; Josuttis, 2007). The considerable effort on investigating service testing is surveyed by Canfora and Di Penta (2009) and by Bozkurt et al. (2012). SOA has posed new factors, such as distribution, lack of observability and control, dynamic integration with other applications, complex message exchange in service compositions, and presence of XML standards, which need to be considered during the testing activity. These factors increase the complexity of tests and prevent most of the consolidated testing approaches from being applied directly (Canfora and Di Penta, 2006a). Moreover, agile test case generation and high level of reliability are required, since service-oriented applications usually implement solutions for flexible and mission critical business processes.

Among the existing testing techniques, MBT is a promising candidate to be applied in the context of SOA and Web services. Besides the automatic test case generation, another relevant characteristic for testing services is the adoption of formal and black-box models. Black-box models are appropriate for service-oriented applications because internal details of services are usually not observable, the complexity of interactions and test harness can be abstracted, and the formality of the model contributes to more reliable tests. Moreover, MBT is generally more efficient when the execution of tests can also be automated (Utting and Legeard, 2006), as in service-oriented applications. Another characteristic of MBT that can be adequate for service-oriented applications is the support for requirement evolution. Changes in some requirements can be easily incorporated; when the tester updates the model, the test suite is automatically generated again, avoiding error-prone manual changes (Dalal et al., 1999; Utting and Legeard, 2006). Moreover, the appropriate application of MBT to software projects brings several benefits, such as high fault detection rate, reduced cost and time for testing, and high level of automation (Dalal et al., 1999; Broy et al., 2005; Utting and Legeard, 2006; Grieskamp et al., 2011; Zander et al., 2011).

Formal testing approaches have been proposed for service-oriented applications (for a systematic mapping of those approaches, see (Endo and Simao, 2010b,a)) and some of them are characterized as MBT approaches for single services (Heckel and Mariani, 2005; Keum et al., 2006; Frantzen et al., 2006; Dranidis et al., 2007; Bertolino et al., 2008) and service compositions (Wieczorek et al., 2009; Mei et al., 2009a; Wieczorek et al., 2010). Although these studies provide means to test single and composite services, they are not concerned with establishing a process and describing the necessary steps in detail. Moreover, the systematic testing of service compositions remains an open topic. The behavior of the composite services depends not only on the composition itself, but also on the partner services. Service compositions may establish complex communications among the integrated services, in which missing or unexpected messages can lead to a failure. Furthermore, the composition may fail due to undesirable behaviors of partner services, such as corrupted messages, unavailable servers, and long response time.

An orthogonal issue on applying MBT to service-oriented applications is which technique will be employed to describe the test model. Among several modeling techniques, FSM and ESG emerge as strong candidates to enable the MBT of service-oriented applications. FSM is a powerful technique, applied to many types of software (Lee and Yannakakis, 1996; Hierons et al., 2009) and which contains several test case generation methods (e.g., W, HSI, SPY, H, and P). Ideally, the adoption of these methods to test any type of software, as well as service-oriented applications should be based on experimental comparisons. On the other hand, ESG can be used in the context of SOA and Web services, once message exchanges in a service composition can be viewed as events that follow an order. Thus, FSMs and ESGs capture different aspects of the application and can be both used in a complementary way.

1.2 Objectives

Since SOA and Web services have been used in complex business processes and mission critical systems, the development of service-oriented applications demands quality assurance and high level of reliability. Software testing can be certainly employed to fulfill these requirements. Although the testing of SOA and Web services has been investigated by the scientific community, more contributions can be achieved by the application of MBT, as well as formal modeling techniques, like FSM and ESG.

In this context, this dissertation investigates the general research question: *"Is MBT applicable to test service-oriented applications so that test cases are generated to verify the SUT formally in a holistic way?"*. Based on this question, the main objectives of the work are described as follows:

- *Proposal of an MBT process for service-oriented applications*: we aim at revisiting the MBT process in the context of service-oriented applications, identifying steps, artifacts, and tools, and investigating the MBT process and its level of automation and practical application. In this study, test models are designed using FSMs and test cases are generated using one of the existing FSM-based test methods.

- *Comparison of FSM-based test methods*: we aim at conducting an experimental comparison of the FSM-based test methods. The adequate selection of a given method implies assessing the generated test suites with respect to their characteristics, overall cost, and effectiveness.

- *Definition of a model-based approach to test service compositions*: enriched from accumulated experience on MBT and FSM-based testing, we aim at proposing a formal approach to test composite services, involving a modeling technique and algorithms to generate holistic test suites. Furthermore, we expect the testing approach to be automated by supporting tools.

- *Evaluation of the proposed testing approach*: we aim at evaluating the proposed testing approach using data collected from three sources. First, the applicability of the approach will be evaluated in a case study. Second, the cost of applying the approach, measured by the manual effort and computational time spent will be analyzed. Third, the approach will be experimented in an industrial context with real-world applications.

The next section addresses the contributions of the dissertation in accordance with the proposed objectives, following its chapter structure.

1.3 Summary of Contributions and Dissertation Outline

Chapter 2 brings an overview of the background information that supports the topics investigated in this dissertation. Initially, the concepts of SOA and Web services technologies used in the development of service-oriented applications are reviewed. Then, foundations of software testing and its terminology are presented. The MBT approach is discussed with respect to its definitions, testing steps, modeling techniques adopted in the dissertation, and advantages/disadvantages. The chapter also describes the concepts of service testing and summarizes the results of a systematic mapping on formal approaches to test services. This systematic mapping study was published (Endo and Simao, 2010a,b) and updated for inclusion in the dissertation. Finally, the MBT of service-oriented applications is discussed in detail.

The contributions of this dissertation are described in Chapters 3, 4, 5, and 6. Chapter 3 presents an experimental evaluation of test case generation from FSMs, comparing the W (Vasilevskii, 1973; Chow, 1978), HSI (Luo et al., 1995), SPY (Simao et al., 2009b), H (Dorofeeva et al., 2005b), and P (Simao and Petrenko, 2010b) methods. The evaluation allows the tester to choose the generation method which is more adequate to test a particular SOA. The test case generation methods are analyzed with respect to different dimensions, such as test suite characteristics (number of test cases and test case length), overall cost of the test suite, and fault detection ratio. The results show that, on average, the recent methods (H, SPY, and P) produced longer test cases, but smaller test suites than the traditional methods (W, HSI). The recent methods generated test suites of similar length, though P produced slightly smaller test suites. The SPY and P methods achieved the highest fault detection ratios and HSI had the lowest. For all methods, there was a positive correlation between the number of test cases and the test suite length and between the test case length and the fault detection ratio. The chapter is based on results published in (Endo and Simao, 2013).

Chapter 4 introduces an MBT process that identifies steps, artifacts, and tools to verify service-oriented applications. To evaluate the process instantiation, we conducted an exploratory study so that FSM was employed as the modeling technique and the P method (Simao and Petrenko, 2010b) as the test case generation method. The P method was one which provided better overall performance, according to the study summarized in Chapter 3. We analyzed the process automation and its practical usage by developing a prototype tool and conducting a case study with two applications. The preliminary results show that the MBT process can be employed to service-oriented applications with reasonable effort and complexity and a high level of automation is achievable through the use of tools. The chapter is a summary of results published in (Endo and Simao, 2011).

Chapter 5 describes a holistic and event-driven approach to test Web service compositions developed from our previous studies with classical MBT (Chapter 3) and testing of service-oriented applications (Chapter 4). This approach, named Event Sequence Graphs for Web Service Composition (ESG4WSC), extends ESGs to model the behavior of composite services through a sequence of messages (events). It is holistic in the sense that test cases can be generated from a test model to verify the service composition behavior under regular circumstances (positive testing) and undesirable situations (negative testing). The model and the test case generation algorithms are formally defined for the ESG4WSC approach. Tools were developed to support the main steps of MBT, such as modeling, test generation, concretization, and test execution. The chapter is a summary of results published in (Belli et al., 2013).

Chapter 6 describes three experimental studies conducted to evaluate the ESG4WSC approach proposed in Chapter 5.

1. First, a case study was conducted to verify the applicability of the ESG4WSC approach and its tools to complex and large composite services. The results show evidences that the approach scales well with larger compositions. Faults were detected not only in the SUT, but also in the specification of a non-trivial service-oriented application. The case study was reported in (Belli et al., 2013).

2. Second, a cost analysis was performed to evaluate the computational cost and human effort involved in the use of the approach. The analysis of two algorithms to generate test cases revealed trade-offs in the execution time and cost of test execution. The human effort to modeling and concretization was also evaluated. The preliminary results have supported ESG4WSC as an intuitive technique to test composite services. Testers devoted reasonable, but usually straightforward effort in the test concretization, as observed by the collected metrics.

3. Third, an experience of the ESG4WSC use in an industrial environment using real-world applications is reported. The study provides preliminary evidences that MBT, specifically the analyzed approach, is feasible to test service-oriented applications in real and less controlled situations within an IT corporation. The results revealed a set of issues that impacts on the approach and tools and showed how they can be overcome. The experience was reported in (Endo et al., 2012).

Finally, Chapter 7 concludes the dissertation, revisiting the achieved contributions, summarizing limitations, and sketching future directions.

CHAPTER

2

Background

2.1 Overview

This chapter brings an overview of the subjects that underlie the research conducted in this dissertation. The organization of the chapter is as follows. Section 2.2 introduces the concepts of SOA and the technologies involved. Section 2.3 describes the foundations and terminology of software testing. Section 2.4 presents the model-based testing approach. Section 2.5 characterizes the related work on service testing. Particularly, Section 2.5.1 presents the results of a systematic mapping on formal approaches to test Web services. Previous versions of this systematic mapping were published in (Endo and Simao, 2010a,b).

2.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is an architectural style that fosters scalability and flexibility, being appropriate for complex and heterogeneous distributed systems (Josuttis, 2007). It provides a standardized, distributed, and protocol-independent computing paradigm to develop loosely coupled applications. MacKenzie et al. (2006) define SOA as a paradigm to organize and use distributed competences that can be controlled by different owners. It provides a uniform way to offer, discover, interact and use these competences to produce desired effects. Josuttis (2007) defines SOA as a paradigm to realize and maintain business process developed as large distributed systems. These systems are usually heterogeneous and SOA aims at connecting them easily providing high interoperability.

In SOAs, software resources are wrapped as “services”; they are well-defined and self-contained modules that provide business functionality and are independent from other service states or contexts (Papazoglou and Heuvel, 2007). A service can be viewed as a black box since its implementation details are hidden and only its interface is available. According to MacKenzie et al. (2006), a service is a mechanism to enable the access to one or more competences by means of an interface. Along the development of service-oriented applications, all functionalities are provided as services, such as business functions, transactions composed of low level functions, and system service functions (Papazoglou and Heuvel, 2007).

The interaction in a SOA is represented by three entities (Figure 2.1) (Huhns and Singh, 2005; Papazoglou and Heuvel, 2007):

Service provider represents the entity responsible for creating a service. The provider should describe the created services in a standardized way and publish them in public and centralized registries. These tasks assure that a service might be understood and found by search engines or anyone that wants to use it;

Service consumer represents the entity that uses a service created by a provider. The descriptions supplied by the provider should give the necessary information for a consumer to interact with a service. The required information can be obtained from a service registry; and

Service registry represents the entity which both provider and consumer interact with. Providers publish their services in a registry and, as a consequence, consumers can find and use them. Therefore, an interaction may occur by dynamic binding so that a service-oriented application discovers and interacts with a service at runtime.

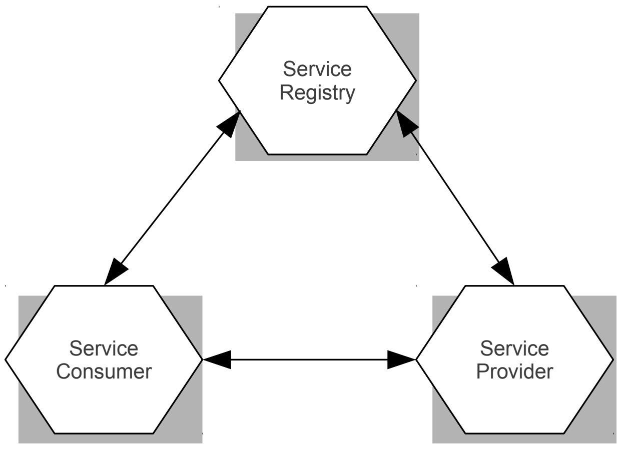


Figure 2.1: Entities in a SOA.

Services can be classified with respect to their purposes and different roles performed. Josuttis (2007) describes a technical classification usually adopted by the community:

Single/Basic/Atomic services: provide basic business functionalities that are meaningless if divided in multiple services. The role of these services is to encapsulate the backend[1] for consumers (and higher level services) to access the backend using the SOA infrastructure. There are two types of single services: data and logic. Data services read and write pieces of data in a backend system. Logic services represent fundamental business rules, processing input data and returning correspondent results.

Composite/Composed services: represent the first category of services that are composed of other services (single and/or other composite services). These services operate in a level higher than single services and represent workflows that execute in a short period of time (so-called *short-running*).

Process services: represent workflows that execute in a long period of time (so-called *long-running*) and involve human intervention. Although there are differences with respect to composite services, the process services are also assembled by the composition of other services. Thus, process services are also referred to as composite services in this dissertation. Section 2.2.2 presents further details on the composition process.

The adoption of an Enterprise Service Bus (ESB) has been considered essential for companies to fully achieve the advantages of SOAs (Josuttis, 2007). The ESB works as a backbone that supports many communication patterns over different transport protocols and provides interesting capabilities for service-oriented applications, such as routing, provisioning, service management, integrity, and security (Papazoglou and Heuvel, 2007). Figure 2.2 illustrates how the ESB acts as an intermediate layer between providers and consumers and all their communication passes through it. It provides a set of features by receiving, operating or mediating on the service messages, as they flow through the bus. Schmidt et al. (2005) describe a set of mediation patterns for ESBs, such as:

- *Monitor pattern* establishes the observation of messages that pass through the ESB, without modifying the messages. This pattern can be applied to logging, audition, monitoring of service levels, measurement of consumer usages, and so on.
- *Aggregator pattern* establishes the monitoring of messages from different services over a period of time and the generation of new messages or events. This pattern can be useful for realizing complex scenarios so that, e.g., a set of events that should be observed to trigger a certain event.

Several ESB applications are available from proprietary vendors to open source solutions[2]. In this dissertation, we adopted the open source version of a lightweight Java-based ESB application called Mule-ESB (MuleSoft, 2012).

[1]A *backend* may be any software application responsible for a specific group of data or functionalities, e.g., databases, hosts, mainframes, a group of servers, and so on.
[2]http://en.wikipedia.org/wiki/Enterprise_service_bus – last accessed on 20/02/2013.

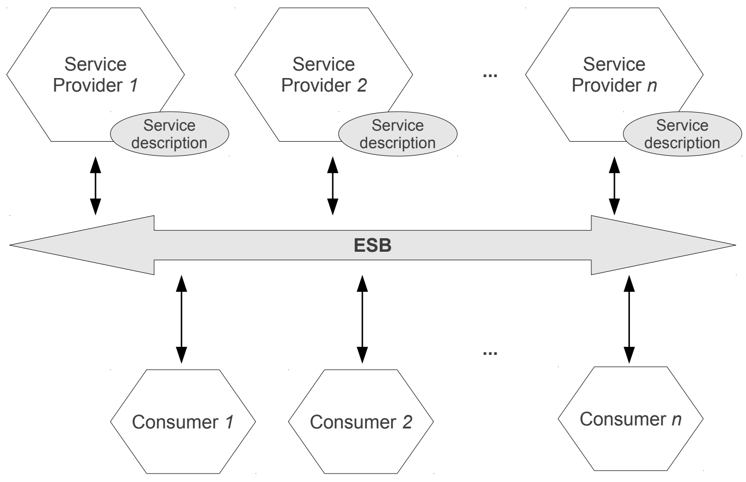


Figure 2.2: Service interaction with an ESB.

2.2.1 Web Services

Web services, the most adopted technology to implement the concepts of SOA, refer to a collection of standards that aim at interoperability (Josuttis, 2007). It allows that applications in different platforms and developed in various languages communicate with each other by means of standardized Web protocols. Moreover, it supports the development of services that are easily integrated with other applications using the Internet as their communication channel.

Web services standards are based on the *Extensible Markup Language* (XML) (W3C, 2003). XML is a general purpose markup language to describe models, formats, and data types. A well-formed XML document can be verified by validation rules using XML Schema (W3C, 2004d). It is possible to extract data from an XML document using queries in XPath (Berglund et al., 2010) or XQuery (Boag et al., 2010) and even perform transformations to other types of documents using XSLT (W3C, 1999). The most popular programming languages have supported the basic features of Web services, e.g., Java[3], C++[4], C#[5], and Ruby[6].

First Generation The first generation of Web services is based on three standards: SOAP, WSDL and UDDI (Erl, 2005), described as follows.

SOAP [7] (W3C, 2004b) is a W3C[8] protocol used to structure the pieces of information exchanged in a decentralized and/or distributed environment, allowing communication in a simple and platform/programming language independent way. In service-oriented applications, SOAP defines the structure of messages exchanged among the services. A SOAP message usually consists of the following parts: *(i)* an envelope that defines the begin and the end of a message; *(ii)* a header

[3]http://jax-ws.java.net/ – last accessed on 20/02/2013.
[4]http://axis.apache.org/axis2/c/core/index.html – last accessed on 20/02/2013.
[5]http://msdn.microsoft.com/en-us/library/ms950421.aspx – last accessed on 20/02/2013.
[6]http://wso2.org/projects/wsf/ruby – last accessed on 20/02/2013.
[7]SOAP was acronym for *Simple Object Access Protocol* in version 1.1; this is no longer the case in version 1.2.
[8]World Wide Web Consortium (W3C) – `http://www.w3c.org`

that contains optional attributes; *(iii)* a body that contains the XML data included in the message; and *(iv)* attachments that are files bound to the message. SOAP also provides structure `fault` to specify error messages. A SOAP-fault is *expected* (by the consumer) if described in the interface and used to map exceptions that happen within the service; otherwise, it is *unexpected*.

Web Service Description Language (WSDL) (W3C, 2001, 2007) is a W3C standard to describe the service interface. WSDL specifies three basic components (Newcomer, 2002; Curbera et al., 2002):

- *Data types*: WSDL includes an abstract container to record the definitions of data types used in the service interface. These data types are encoded with XML Schema within the own WSDL document or in an external referred file.
- *Operations*: each operation describes an abstract interface for a behavior or action offered by the service. Inside each operation, input and output messages are specified and correlated with data types defined. An operation can be request-only (it receives a message) or request-response (it receives a message and sends a message back).
- *Binding protocols*: WSDL allows to specify protocols of lower layers. For instance, the developer can define the protocols of the message and transport layers. In Web services, these protocols are usually SOAP and HTTP.

Universal, Discovery, Description and Integration (UDDI) (OASIS, 2004) is an OASIS[9] standard that defines functionalities to support the description and discovery of business, organizations, and other service providers. The service capabilities are defined using keywords and small descriptive tags. Thus, UDDI becomes a powerful tool for consumers to find out service providers that meet their needs (Newcomer, 2002; Cerami, 2002).

Second Generation The second generation of Web services, also known as 'WS-*', refers to standards proposed after the first generation (Erl, 2005). They were established to complement and enhance the infrastructure provided by the initial standards (SOAP, WSDL, and UDDI). Some examples of these standards are as follows.

WS-Addressing (W3C, 2004c) defines mechanisms to address services and messages. It introduces two important concepts: references to endpoints and information headers inside the message. *WS-Security (OASIS, 2006)* defines how to apply security techniques for authorization, integrity, and privacy to Web services. It describes a standardized way to embed security information, such as tokens, cryptography, and signatures. *WS-Agreement (OGF, 2007)* defines mechanisms to specify domain-independent elements of an agreement process between two parties. It is used to establish Service Level Agreement (SLA) contracts that include Quality of Service (QoS) properties like response time, latency, and reliability.

[9]Organization for the Advancement of Structured Information Standards (OASIS) – `http://www.oasis-open.org`

Other standards, like WS-BPEL and WS-CDL, developed to support service compositions are discussed in the next section.

2.2.2 Service Composition

One of the principles proposed by Erl (2005) says that services are composable, i.e., they are designed to participate in a service composition. A composition combines several services (called partner services) in a workflow to model and execute complex business process. Thus, new functionalities are defined and implemented by combining and interacting with existing services. The outcome of a composition process is a service itself that is referred to as composite service (Josuttis, 2007). Service composition allows speeding up the application development, improving service reuse, and easing the interaction with complex services (Milanovic and Malek, 2004; Kazhamiakin et al., 2006).

Composite services can be static or dynamic (Shen et al., 2007). Static compositions define and bind the partner services and their endpoints (addresses) at design time. Dynamic compositions discover and bind the partner services at runtime. A dynamic composition queries a service registry to select the appropriate services and then assigns their endpoints.

Developers can design service compositions using two approaches: orchestration and choreography (Peltz, 2003; Josuttis, 2007). These approaches are explained as follows.

Service Orchestration In orchestrations, a central entity controls the partner services and coordinates the execution of different operations, taking into account pre-defined requirements. The partner services are not aware (and do not need to be aware) of their integration and of being part of a high level business process. The central entity (coordinator) centralizes all tasks, including the business logic and the invocation order of the services. In orchestration, a business process interacts with partner services by means of messages, besides including the business logic and execution order of tasks. Figure 2.3 illustrates the service orchestration.

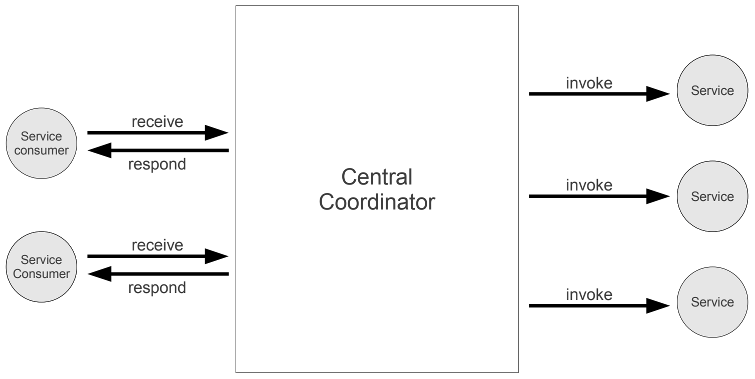


Figure 2.3: Service orchestration – adapted from (Peltz, 2003).

Web Services Business Process Execution Language (WS-BPEL) is an XML-based language that supports the orchestration of services; it was initially developed by BEA, IBM, Microsoft, SAP and Siebel and is currently standardized by OASIS (Jordan et al., 2007). The language provides a set of instructions (so-called activities) that allows the developer to implement executable orchestrations. The activities are basic or structured. A basic activity is an instruction that does not interfere in the execution flow and executes a single procedure, such as invocation of a service or manipulation of data messages. A structured activity manages the process flow, specifying the execution order, e.g., loops and conditional branches. WS-BPEL based orchestrations are executed in engines, such as ActiveVOS (ActiveVOS, 2013), Oracle BPEL (Oracle, 2013), and Apache ODE (ASF, 2010). For each available composition, the engine requires a WS-BPEL file that describes the orchestration, WSDL interfaces of partner services, and a configuration file. The engine also monitors the persistence, message queues, alarms, and many other execution details.

Service Choreography Service choreographies follow an approach different from orchestrations and do not use a central coordinator. Each service involved in the choreography knows when executing its operations and who interacts with, i.e., each service has a protocol. All partner services are aware of the business process, operations to execute, and messages to be exchanged. According to Peltz (2003), the choreography is more collaborative and allows that each part describes its own participation. The choreography tracks the message sequence among the multiple parts and sources. Figure 2.4 illustrates the service choreography. As choreographies avoid a centralized control, it can have better scalability than orchestration. However, tracking the current state of a composition and finding the cause of an unexpected behavior can be complex tasks (Josuttis, 2007).

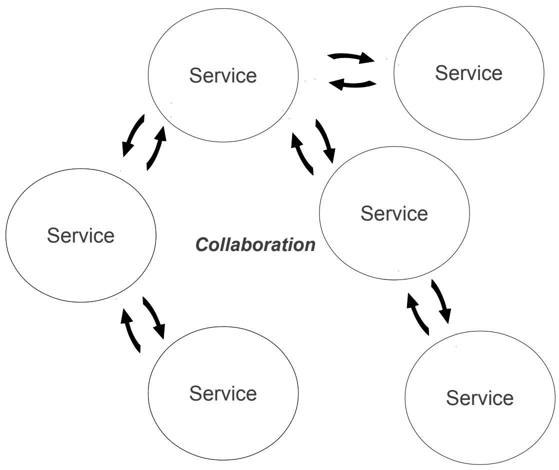


Figure 2.4: Service choreography.

Web Services Choreography Description Language (WS-CDL) (Kavantzas et al., 2005) is an XML-based language that describes pairwise collaborations of participants (service choreography) by defining a global view of their common observable behavior. It aims at describing the ordered message exchange that results in the achievement of a business goal. The language focuses on the composition of interoperable collaborations between participants, without connection with the platform or programming model used. The description of a choreography is a contract with multiple participants that models the composition from a global point of view. WS-CDL is the mean used to specify this technical contract (Kavantzas et al., 2005).

2.3 Software Testing

The software development process involves a set of activities in which, in spite of techniques, methods, and tools employed, faults in the final product can still occur. To minimize faults and risks associated, software testing is one of the main activities of V&V applied by practitioners. It is a process in which an SUT is executed with the aim of finding its faults. Software testing is a critical element in the quality assurance and represents the final review of specifications, designs, and source code (Myers et al., 2004; Pressman, 2005). The testing activity is one of the elements that provides evidences of software reliability in addition to other activities, such as formal reviews and rigorous techniques of specification and verification (Maldonado, 1991).

In the context of software testing, we use the IEEE standard 610.12 (IEEE, 1990) to define the following terms: *fault* – step, process, or data definition that is incorrect (instruction or command); *mistake* – human action that produces an incorrect result (an incorrect action made by a programmer); *error* – the difference between the actual value and the expected value, i.e., any intermediate incorrect state or an unexpected result in the execution; and *failure* – production of an output incorrect with respect to the specification.

This standard also defines *test case* as a set of test inputs, execution conditions, and expected outputs, developed for a particular goal, such as to exercise a given path in a program or to verify a given requirement. In some systems, a test case can be a sequence of events or actions to be applied and/or observed. This test case is also referred to as *test sequence*. A collection of test cases is named *test suite*.

The tests can be applied in three phases, described as follows:

Unit testing: focuses on testing the smallest modules (units) implemented in the source code of a software application. It is limited to the logic and data structures within the unit limits. An unit can be a function in the procedural paradigm, or a method or a class in the Object Orientation (OO) paradigm (Binder, 1999; Vincenzi, 2004).

Integration testing: occurs in parallel with the software integration phase and aims at revealing faults associated with the unit interfaces. As the integration testing involves the verification of parts which are not fully developed, drivers and stubs need to be used (Ammann and Offutt, 2008). The driver is a module that emulates the call for a unit under test, and the stub is a module that simulates the behavior of a called unit.

System testing: aims at exercising the system as a whole by means of several types of testing, such as recovery, security, and performance testing (Pressman, 2005). Recovery testing forces system failures in many ways, verifying if the recovery process is executed correctly. Security testing verifies the protection mechanisms of a system by means of attacks to the SUT. Performance testing evaluates how well the SUT performs at runtime under a given workload.

2.3.1 Testing Techniques

The tester needs a measure that indicates if the software has been tested enough. A testing criterion defines which properties or requirements should be tested in order to evaluate the quality of a test suite (Zhu et al., 1997). Given an SUT P, a test suite TS (that contains a subset of the inputs of P) and a testing criterion C, TS is C-adequate to test P if TS satisfies the test requirements established by C. Testing criteria can be used to decide the end of the testing phase, characterizing the *test adequacy criteria.* The criteria are useful to support test generation, guiding the tester during the selection of test cases. When the criteria are used in this task, they are referred to as *test selection criteria*.

Using the sources of information to derive test cases as basis, the testing criteria can be classified in three techniques: functional, structural and fault-based (Maldonado, 1991).

Functional Testing Functional testing, also known as black-box testing, is a technique so that the tester does not have knowledge on the internal behavior and the program structure (Myers et al., 2004). The test cases are derived from requirement specifications. Some testing criteria of the functional technique are presented as follows:

Equivalence Partitioning: divides the input domain of the SUT in a finite number of equivalence classes used to derive test cases. The equivalence classes are divided into valid and invalid groups. The tester selects test cases to cover the maximum number of valid classes. On the other hand, only one test case is selected to cover each invalid class (Myers et al., 2004). This criterion is motivated by the assumption that testing one representative value of each class is equivalent to test any other value within the same class.

Boundary-Value Analysis: is usually used together with the equivalence partitioning class, emphasizing the limits associated with input conditions (Mathur, 2008). The tester selects test cases from the class limits since it may occur more faults in these points (Pressman, 2005). The same procedure is done with the output domain, that is divided into classes and test cases are designed to produce outputs in these classes' limits (Myers et al., 2004).

Cause-Effect Graphing: establishes testing requirements based on combinations of input conditions. First, the possible input conditions (causes) and actions (effects) are elicited. Then, the elicited causes and effects are connected in a graph. The graph is usually transformed into a decision table which is efficient to model combinations of input conditions (Myers et al., 2004). Finally, test cases are derived from the decision table.

Structural Testing Structural testing, also known as white-box testing, is a technique so that the tester bases the test cases on the internal logic of an SUT (Myers et al., 2004). Most of the structural criteria use an SUT representation called Control Flow Graph (CFG). The CFG consists of establishing a relation between nodes and blocks and of indicating possible control flow between the blocks through edges. Therefore, a CFG is a directed graph in which each vertex represents an indivisible command block and each edge is a possible branch from a block to another.

Rapps and Weyuker (1985) propose an extension of the CFG named *Def-Use Graph*. The extension adds pieces of information about the data flow, characterizing associations between points where a variable is defined (variable definition) and points where this value is used (reference or variable use). The testing requirements are determined based on these associations. Two types of use are defined: c-use and p-use. C-use directly affects a computation being performed or allows that the result of a previous definition can be observed; and p-use directly affects the control flow of the SUT.

The most widespread criteria of this technique are the following:

All-Nodes: this criterion requires that an adequate test suite executes, at least, once each node in the CFG, i.e., each program command is executed at least once.

All-Edges: this criterion requires that an adequate test suite execute, at least, once each edge in the CFG, i.e., each branch in the control flow.

All-Definitions: requires that each variable definition in the def-use graph should be covered, at least, once by a p-use or a c-use.

All-Uses: requires that all associations between a definition and its subsequent uses in the Def-Use Graph are executed, at least once, through a path where the considered variable is not redefined (a definition-clear path).

Fault-Based Testing Fault-based testing uses the knowledge about common faults in the development process to define testing requirements. Two criteria of this technique are Error Seeding and Mutation Analysis.

Error seeding introduces a known amount of faults in the SUT. After the test execution, the total number of found faults is analyzed, verifying which are natural or artificial. Using statistical methods, the number of natural faults still existing in the program can be estimated (Budd, 1981; Ramamoorthy and Bastani, 1982). Error seeding can also be used to measure the effectiveness of a testing criterion (Ramamoorthy and Bastani, 1982). Different types of faults can be manually seeded in a program and, after applying a test suite adequate for a given criterion, the results show the criterion's effectiveness for each type of fault.

Mutation analysis (DeMillo, 1978) is a testing criterion that evaluates the adequacy of a test suite to reveal specific faults. To do so, a set of mutation operators are used to automatically generate programs (similar to the original SUT), but containing some fault. These modified programs are called “mutants”. A mutant is “killed” when for some test case its outputs are different from the original; otherwise it is “alive”. If there is not test case that distinguishes the mutant and the original program, this mutant is called “equivalent”. The test suite adequacy to the mutation analysis criterion is measured by the mutation score that relates the number of killed mutants with the number of generated mutants less the equivalent ones.

2.4 Model-Based Testing

Software testing can be automated through the generation of test cases from a structural or behavioral model of the SUT, so-called test model. These approaches are collectively known as Model-Based Testing (MBT) (Sinha and Smidts, 2006). Although some authors claim that testing is always model-based since implicit mental models guide the tests (Binder, 1999), the idea of MBT is to apply explicit models (Pretschner and Philipps, 2004). Utting and Legeard (2006, p. 8) define MBT as the automation of black-box testing design in which, given an appropriate test model, test sequences can be generated and transformed into executable scripts. The literature usually classifies MBT as functional testing because there is a predominance of modeling techniques that considers the SUT as a black box. MBT can also be applied in any testing phase (unit, integration, and system) (Utting and Legeard, 2006).

In this dissertation, we distinguish the terms modeling technique and test model. Test model refers to the artifact generated by the modeling activity during MBT. Modeling technique refers to the notation (language) adopted to express, in a well defined way, a test model.

2.4.1 MBT Steps

An MBT approach can be divided into four main steps (El-Far and Whittaker, 2001; Pretschner and Philipps, 2004; Utting and Legeard, 2006; Bouquet et al., 2006) described as follows.

1. Modeling: As in traditional software testing, the requirements are information sources for understanding the SUT functionalities. Moreover, the software is in an environment that involves different factors like operational systems, other applications, different types of libraries, and so on. Thus, the tester needs to learn about both software and its environment (El-Far and Whittaker, 2001). Utting and Legeard (2006) recommend the creation of test models using the requirements to maximize the independence between the test model and the SUT. Artifacts from the analysis and design phases can also be used as basis for understanding and constructing the test model.

Test models should be smaller than the SUT to reduce their costs. Furthermore, models should have enough details to describe accurately the parts of the SUT being tested (Aydal et al., 2009). Since the model is designed aiming at the tests, it is not necessary to specify all the overall system’s behavior. Several small, partial models are usually more useful than one single big and complex model. For instance, the tester designs models for each subsystem or component and, after testing each of them, builds a higher level model for the entire system (Utting and Legeard, 2006).

Models are simplifications of the concrete world. Pretschner and Philipps (2004) discuss the simplification by (1) detail omission and (2) detail encapsulation. In the detail omission, the tester discards irrelevant parts to show only the fundamental ideas, keeping the model in a higher level of abstraction. As other modeling tasks, selecting the pieces of information discarded and the ones described is a challenge. In the detail encapsulation, the tester labels parts of the SUT but does not describe them. This concept is used in OO modeling, where all code needed to implement a behavior is encapsulated in a reference that is a method's name.

2. Test generation: The test generation depends on the modeling technique chosen to describe the test model. In general, modeling techniques have properties that make the generation less costly and ease the automation (El-Far and Whittaker, 2001). Moreover, test selection criteria are required to limit the number of test cases derived from the test model. Structural testing criteria, such as control flow and data flow (shown in Section 2.3.1), can be reused to cover models. The fault-based technique can also be applied considering faults in the test model.

In this step, the aid of a tool is essential to support the automatic generation of test cases. The tool receives as input the test model and the test selection criterion and generates as output a test suite. The generated test cases are abstract because they are in an abstraction level different from the SUT and cannot be directly executed. The algorithms for test case generation are dependent on the modeling technique and test selection criteria.

3. Concretization: Concretization involves transforming abstract test cases into ones executable in the SUT. This step assures that the entire process will be automated (Utting and Legeard, 2006). Test cases derived from the model (abstract) need to be concretized before applied to the SUT by using **adaptors** (Pretschner and Philipps, 2004). An adaptor is a software component capable of translating inputs and outputs in two levels of abstraction, the test model and the SUT. Basically, the adaptor has to implement two functions: a concretization function $conc()$ and an abstraction function $abst()$.

Figure 2.5 illustrates the operation of an adaptor for abstract test case $\langle in; out \rangle$ derived from test model M. in represents an input that, when applied to the SUT, produces output out. The concretization function transforms in (i.e., $conc(in)$) and is then applied. The SUT produces output out' that is abstracted to the same abstraction level of model M using the abstraction function, i.e., $abst(out')$. Finally, the output of the system is compared to expected output out and a verdict is given.

4. Test execution: This step is the execution of abstract test cases that, after passing through concretization, can be executed in the SUT. If the execution is performed separated from generation, the testing is offline; otherwise it is online. In other words, a test input is generated and

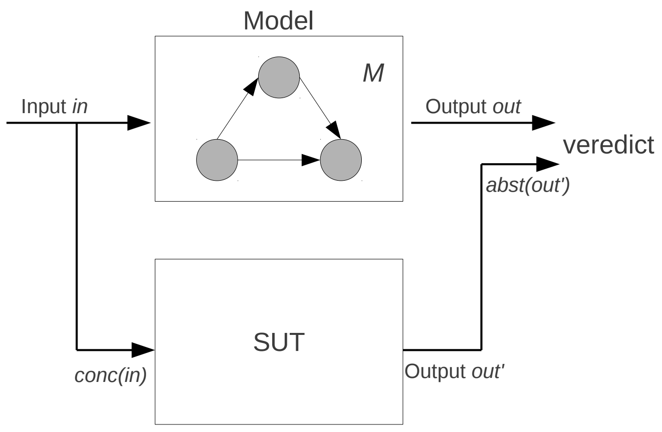


Figure 2.5: Concretization mechanism – adapted from (Pretschner and Philipps, 2004).

applied to the SUT and, based on the current output, the next procedure is decided online (Hartman et al., 2007).

The results of test execution are analyzed and corrective actions are made. If the test model specifies inputs and outputs, automatic verdicts can be given. In other words, the model works as a test oracle. The verdict can be pass, failed, or inconclusive (Tretmans, 1992). *Pass* indicates that the test case was successfully executed and the expressed goal was fulfilled. *Failed* indicates that the result is not in conformance with the goal expressed in the test case. *Inconclusive* indicates that an evidence of nonconformance was not found, but the test goal was not fulfilled.

2.4.2 Modeling Techniques

In MBT, the tester designs a test model of the SUT using a modeling technique. The modeling technique should be formal (i.e., well-defined syntactic and semantically) since the presence of formal models or specifications can lead to more efficient and effective tests (Hierons et al., 2009). According to Utting and Legeard (2006), a model is formal if it has a precise and unambiguous meaning and represents the behavior in a format manageable by software tools. As the model needs to be validated, it should be simpler than the SUT, or, at least, easier to verify, modify, and maintain (Utting et al., 2006). Nevertheless, the model should be sufficiently accurate to support the generation of meaningful test cases. In this section, we briefly introduce the three modeling techniques employed along this work.

Finite State Machines are used in several areas, such as circuits, program analysis, and communication protocols. In this dissertation, an FSM is a Mealy machine composed by states and transitions (Gill, 1962; Lee and Yannakakis, 1996). For each transition, an input symbol is consumed and an output symbol is produced. An FSM can be represented by a state diagram, which is a directed graph so that nodes are states and edges are transitions. The edges are annotated with inputs and outputs associated with the transition. Figure 2.6 illustrates an FSM for a comment printer

(Chow, 1978); in the transitions, symbol ':' separates inputs from outputs. Further references on FSMs are made in Chapters 3 and 4.

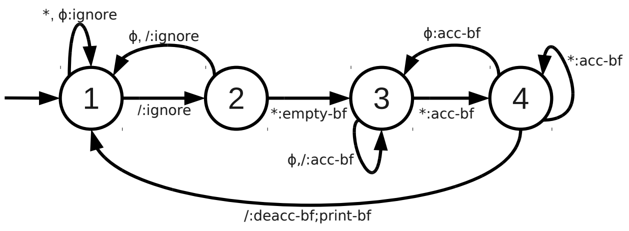


Figure 2.6: Example of an FSM – adapted from (Chow, 1978).

Decision Tables are used to represent constraints on test inputs and expected effects (outputs), as well as to model their possible combinations (Sharma and Chandra B., 2010; Feng et al., 2011). Decision Tables (DTs) have been associated to cause-effect graphs (Section 2.3.1), though they can be directly applied for testing (Binder, 1999; Myers et al., 2004; Mathur, 2008). Table 2.1 illustrates a DT that models constraints and possible effects for an operation "insert card" in an ATM. Combinations of true (T), false (F), and don't care ('-') for the constraints produce different effects. Notice that each combination may represent a test case. Further references on DTs are made in Chapters 5 and 6.

Table 2.1: A decision table for operation "insert card".

		Combinations			
		C1	C2	C3	C4
Constraints	chipcard	*T*	*F*	*T*	*T*
	valid bank card	*T*	-	*F*	*T*
	Active bank account	*T*	-	-	*F*
Effects	display the menu	✓			
	show an error message		✓	✓	✓

Event Sequence Graphs are directed graphs in which nodes are events and edges represent valid sequences of events. ESGs, also known as event-flow graphs (Yuan et al., 2011), have been used to test and verify event-driven systems (van der Aalst, 1999; Belli et al., 2006; Yuan et al., 2011). Figure 2.7 illustrates an ESG that models the events and their sequence in a "copy-cut-paste" procedure. The test generation from ESGs can be reduced to the well-know Chinese Postman Problem (CPP) on directed graphs (Aho et al., 1995). An algorithm for CPP aims to find an optimal (minimal cost) path that contains all edges. Further references on ESGs are made in Chapters 5 and 6.

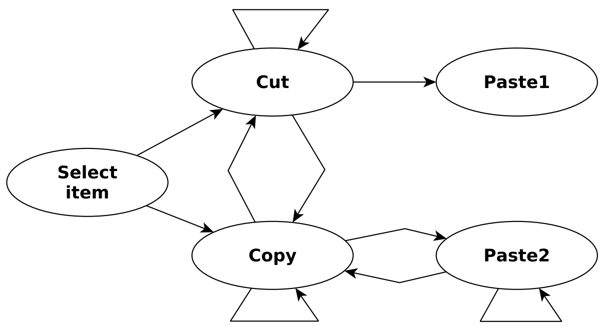


Figure 2.7: Example of an ESG for a "copy-cut-paste" procedure.

2.4.3 Advantages and Disadvantages

This section elicits the advantages and disadvantages of MBT. First, a set of benefits has been reported when MBT is correctly applied during the development process (Blackburn et al., 2004; Utting and Legeard, 2006; Grieskamp et al., 2011):

- *Automatic generation of test cases*: the adoption of test models and supporting tools allows the test case generation in an automatic and systematic way. An MBT tool can become the test generation process faster and less error-prone since costly and repetitive tasks will be automated.
- *Fault detection*: studies conducted by the industry and the academia have shown the effectiveness of model-based approaches in comparison with traditional testing (Utting and Legeard, 2006).
- *Reduced time and cost for testing*: most of the studies that compare MBT with manual testing provide results that favor MBT (Dalal et al., 1999; Farchi et al., 2002; Pretschner et al., 2005). On the other hand, an Intrasoft International study shows that the company's testing process was faster than MBT (Utting and Legeard, 2006). However, the same study concludes that other benefits balance against the greater time for MBT.
- *Improvement of testing quality*: when the testing is carried out manually, the quality of tests is more dependent on the tester' skills, the process is not repeatable, and correlating test cases with system requirements is hard. An MBT process is more systematic and repeatable (Sinha and Smidts, 2006) and allows the test case generation using rigorous test selection criteria and the easier reexecution of a test case.
- *Fault detection in requirements*: the test modeling during the first stages can support the requirements' refinement as well as the identification of specification problems. The fault detection at early phases fosters more software quality and cost savings than in advanced development periods (Pressman, 2005). MBT can be used as an additional step to validate the software requirements.

- *Requirement/system evolution*: during the development process, changes occur and the software system evolves. In MBT, functional changes are easily handled by modifying the test model and regenerating the tests. Thus, the costs of testing maintenance are reduced. The tester also handles changes in test environments since only modifications in the adaptor are necessary.
- *Traceability*: it is the capacity of relating each test case with the model, the test selection criteria, or the system requirement (Utting and Legeard, 2006). MBT allows justifying a given test case, selecting only a subset of tests for a given change in the model, as well as identifying requirements that were already tested or not.

According to Utting and Legeard (2006), a fundamental limitation of MBT is the impossibility of assuring that all differences between the model and the SUT will be found. Although the MBT approach aggregates several benefits, there exist possible disadvantages involved (El-Far and Whittaker, 2001; Utting and Legeard, 2006):

- *Inappropriate use*: some parts of the SUT might be hard or unsuitable for MBT and manual test cases would be more effective. However, the tester may not have the experience to make this decision.
- *Time to analyze a failed test case*: when a test case fails, the tester needs to identify where the fault is, in the SUT, in the model, or in the adaptor. Moreover, the test sequence can be complex and less intuitive, hindering the fault localization.
- *Meaningless metrics*: many test cases can be easily generated in MBT. Hence, traditional metrics based on the number of test cases are useless, being necessary to select other metrics, such as code coverage, requirement coverage, and model coverage (Utting and Legeard, 2006).
- *Tester's skill*: MBT requires more skilled testers with abilities on modeling and proficiency in the modeling technique chosen. This fact poses more costs with training and a high initial effort on learning.
- *State explosion*: the use of state-based modeling techniques can cause a state explosion problem. The state explosion may happen during modeling and leads the tester to design complex models that are hard to maintain. It also occurs with algorithms that traverse the test model. The algorithm can search exhaustively the state space causing an exponential cost (McMillan, 1992).

Although these disadvantages can hinder or even prevent the adoption of MBT, there are practical solutions to deal with these disadvantages. For instance, the company can overcome most of them by adopting appropriate supporting tools, establishing a well-defined testing process, and providing proper training.

2.5 Service Testing

The development of service-oriented applications has received attention from researchers due to the industrial adoption of SOA concepts and technologies (Section 2.2). SOA and Web services have been used to develop mission critical applications for different domains, such as enterprise software, embedded systems, robotics, and pervasive applications (Deugd et al., 2006; Remy and Blake, 2011). As a high level of reliability is demanded in these domains, software testing is a fundamental activity that should be performed.

Service testing has been researched in the last years; comprehensive surveys on this topic can be found in the literature (Canfora and Di Penta, 2009; Rusli et al., 2011; Palacios et al., 2011; Bozkurt et al., 2012). The research on service testing has been motivated by factors that affect the testing of this kind of applications. The following factors are identified (Canfora and Di Penta, 2006a,b, 2009):

- *Lack of observability:* the service is viewed as a black box accessed through its interface, preventing, e.g., the use of structural testing. Moreover, a WSDL document is often the unique artifact available for consumers, i.e., a syntactical description of the service interface. Nevertheless, a behavioral description is necessary for more complex services. The existence of a service model would ease not only the interaction, but also the test case generation;
- *Distribution and lack of control:* services are not physically integrated because they are deployed in different hosts (by different providers). Thus, issues involving the network, synchronization, message passing, and availability are recurring problems. Providers do not have a standardized way to notify the consumers about changes and corrections performed in the service. This implies that the consumer cannot decide a strategy to migrate to a new service version and, as a consequence, to perform regression testing (Bruno et al., 2005)[10];
- *Dynamicity and adaptability:* while it is possible to determine which components will be used in traditional systems, the same does not occur in SOAs. Dynamic compositions can be described as a workflow of abstract services that becomes concrete at runtime by means of data retrieved from one or more registries (Canfora and Di Penta, 2006a). Furthermore, service-oriented applications are adaptable so that services are replaced by others with the same functionalities;
- *XML standards:* services are designed over a set of standards defined using XML (Section 2.2.1). These standards are essential for the correct operation of service-oriented applications. In this context, generated tests should be capable of dealing with artifacts defined with these standards, such as SOAP messages and interface descriptions in WSDL;

[10]In regression testing, a subset of already performed test cases is reexecuted to provide confidence that modifications do not harm the existing behavior of the SUT.

- *Nonfunctional properties:* QoS attributes are important since SLA contracts need to be defined between consumers and providers. The testing activity also needs to consider recurring and demanding QoS attributes, such as performance, availability, and reliability; and
- *Service composition:* Although testing techniques can be reused, service composition testing is still immature with many aspects to be treated (Bucchiarone et al., 2007). The adoption of composition languages, like WS-BPEL and WS-CDL, poses new characteristics and language constructions that should be tested.

The factor "lack of observability" is related to the service testability. According to Tsai et al. (2006), the service testability has two meanings. First, it refers to the degree (of testability) in which the service is developed to *(i)* ease the establishment of quality criteria and *(ii)* execute the tests that meet these criteria. Second, it refers to how testable and measurable the service requirements are given to allow the establishment of testing criteria and execution. One of the factors that impacts in the testability is the levels of access to the service (Tsai et al., 2006), described as follows:

- Level 1: source code of the service is accessible;
- Level 2: binary code of the service is accessible;
- Level 3: model (WS-BPEL, OWL-S, formal specification, etc) is accessible; and
- Level 4: signature (WSDL interface) is accessible.

The accessible artifacts are essential to choose the testing approach that will be applied, since the testing capacity is strongly related to the available information (Bertolino et al., 2004). Therefore, the tester should know the relation between the proposed levels and potential testing approaches, as exemplified in Figure 2.8.

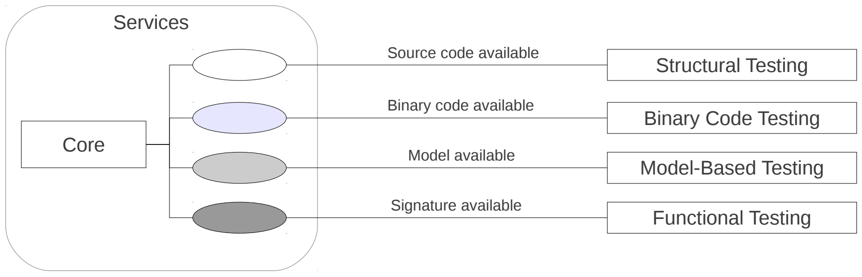


Figure 2.8: Relation between levels and testing approaches – adapted from (Tsai et al., 2006).

In service testing, there are different stakeholders and each one has its own testing goals. Canfora and Di Penta (2006b) define the following five perspectives for testers:

- The service *developer* has more resources to test. However, testing nonfunctional properties is not realistic because it takes into account neither the provider's and consumer's infrastructures, nor the configuration and network load.

- The service *provider* aims to test the SLA properties promised to the consumers. Structural testing cannot be applied and the nonfunctional testing does not reflect the execution environment of the consumer.

- The service *integrator* needs tests to gain confidence in services that will be integrated in his/her composition. To do so, the integrator has to check both functional and nonfunctional properties. Moreover, the dynamic binding adds more challenges since the integrator does not know which service will be selected and used.

- The *certifier* attests formally the quality of a service by performing specific tests. This is helpful for the integrator and the provider that can save testing resources since only the certifier performs the tests. Nevertheless, the certifier performs the tests in his/her own infrastructure and the results cannot be replicated in the integrator's infrastructure.

- The *user* interacts with a service-oriented application through a GUI and the main threats are nonfunctional properties like response time and availability.

Approaches to test services can be classified based on different contexts of a service-oriented application. Figure 2.9 shows a possible classification of service testing approaches. Initially, we divide service testing approaches into: (i) testing services individually (single); and (ii) testing the integration with other services and applications (service composition).

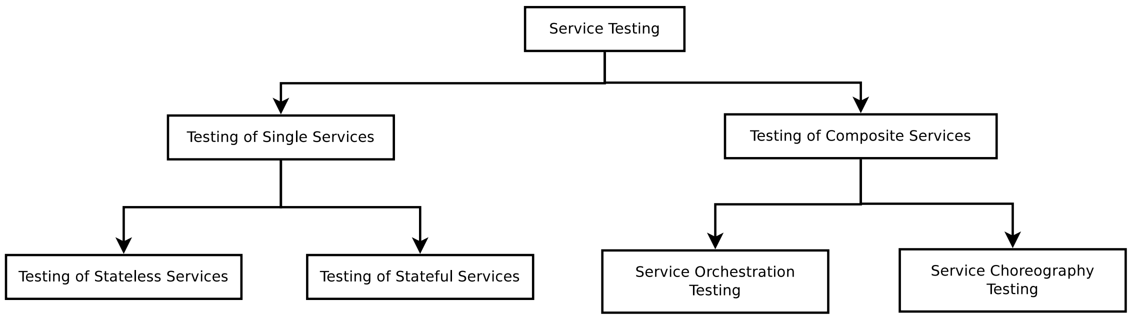

Service Testing
Testing of Single Services
Testing of Composite Services
Testing of Stateless Services
Testing of Stateful Services
Service Orchestration Testing
Service Choreography Testing

Figure 2.9: Classification of service testing approaches.

Testing of single services involves the testing of a given service without any kind of integration. It is similar to unit testing so that a given software module is tested without considering the integration with other units. If we assume that a service is a self-contained and independent software module, there is no need for stubs. The testing aims at simulating scenarios of interaction between the consumer and the service.

A fact that should be considered during the testing of single services is whether the service operations are capable of changing some state or not. If the service does not keep states and its operations only process data, this service is *stateless*. A service is *stateful* when, using its operations, it is possible to change some state, being the own service states or states from its backend system. Hence, the testing of single services in Figure 2.9 is also divided into *(i)* testing of stateless services and *(ii)* testing of stateful services.

Testing of composite services involves the integration between two or more services to implement a more complex service. According to Bucchiarone et al. (2007), more research on V&V for service composition should be conducted. Besides the characteristics common to the traditional software, service composition has a distributed nature and asynchronous behavior (García-Fanjul et al., 2006). The testing of composite services poses challenges for testing and was previously discussed in factor "service composition". In this context, the testing needs to adapt to the type of compositions: orchestration or choreography. The testing of composite services is, therefore, divided in Figure 2.9.

2.5.1 Formal Approaches to Test Services

In this section, we describe a systematic mapping conducted to identify formal approaches to test service-oriented applications. Systematic mapping is a methodology that provides a structure to categorize the research published in a given topic. It aims at giving an overview of a research area, identifying the quantity, the type of research, and available results (Petersen et al., 2008). Further details on the planning and the conduction of this systematic mapping can be found in (Endo and Simao, 2010a). We updated this systematic mapping and the last search was performed on February 2013. We analyzed 53 studies with respect to the classification of service testing, the modeling technique, and the experimental evaluation. The results are presented as follows.

Classification of Service Testing Approaches We divided the studies in accordance with the classification shown in Figure 2.9. There is research interest in both contexts, though we identified more studies for service composition testing: 21 studies in single services and 32 in composite services. We also note that there is no approach that fits for both single services and service compositions. We believe that the current approaches can be combined to provide a more complete testing strategy. However, we could not identify any research effort in this direction.

Single Service Testing: the testing approaches for single services can be classified based on service state, characterizing stateless and stateful services. We also identified approaches that are applicable for both types.

In *stateless services*, the studies focus on test case generation. The work of Tsai et al. (2005b,c,a) is based on the Swiss Cheese model derived from semantic specifications in Ontology Web Language for Service (OWL-S) (W3C, 2004a). Ma et al. (2008) use available WSDL and XML Schema specifications to support the tests. Using a different approach, Chan et al. (2007) use metamorphic relations to deal with the oracle problem and generate new test cases.

In *stateful services*, the studies focus on conformance testing supported by test case generation (Bertolino et al., 2004; Frantzen et al., 2008; Belli and Linschulte, 2008; Keum et al., 2006; Dranidis et al., 2007; Ramollari et al., 2009; Paradkar et al., 2007; Kourtesis et al., 2010; Dranidis et al., 2010). Bertolino et al. (2008) propose on the generation of stubs using a state model and

QoS properties. Li et al. (2008a) employ CFGs to model the valid sequence of operation invocations. Chakrabarti and Rodriquez (2010) present a formal approach to test RESTful web services. Dranidis et al. (2010) propose an automated technique for just-in-time testing of stateful services.

In *stateless and stateful services*, most studies are based on Graph Transformation rules (GT rules), enabling the applicability for stateless and stateful services (Heckel and Mariani, 2005; Lohmann et al., 2007; Park et al., 2009). Moreover, these papers support the process of service discovery. On the other hand, Eler et al. (2010) propose a built-in approach so that any service (stateful or stateless) is augmented with structural testing information like coverage analysis and metadata that helps the tester to improve the tests.

We notice more interest in stateful services (12 studies) than stateless services (five studies). A group of studies focuses on automating the process of test case generation mainly as consequence of the dynamic nature of Web services. Some studies consider that the syntactic specification (WSDL) was augmented with semantic descriptions (e.g., OWL-S, RDF) (Tsai et al., 2005b; Ramollari et al., 2009). Although the approaches can be applied to any type of service, only one study considers the RESTful Web services (Chakrabarti and Rodriquez, 2010). There are few approaches that can be applied to both stateless and stateful services (only four studies).

Service Composition Testing: the testing approaches for service compositions are divided based on the used paradigms: orchestration and choreography.

In *orchestration testing*, WS-BPEL is the main language used to describe the composition (19 studies). There is a particular effort on researching test case generation handling different aspects, such as WS-BPEL specific structures (Zheng et al., 2007; Liu et al., 2008; Hou et al., 2009; Bentakouk et al., 2009; Ni et al., 2011), concurrency (Yan et al., 2006; Li et al., 2008b), timing properties (Lallali et al., 2008; Cavalli et al., 2010; Gao and Li, 2011), run-time composition (Corradini et al., 2008), and testing architecture (Escobedo et al., 2010). Test generation has also been supported by using model checking (De Angelis et al., 2010; Dong et al., 2010). There are studies related to structural coverage criteria that: use workflow modeling (Karam et al., 2007), handle communication among processes (Endo et al., 2008), consider XPath artifacts (Mei et al., 2008), and evaluate traditional data flow criteria (Mei et al., 2009b). Regression testing is approached considering minimization (Li et al., 2008b) and prioritization (Mei et al., 2009c). Approaches have been proposed to support passive testing (Benharref et al., 2006; Morales et al., 2010; Cavalli et al., 2010). There is also an initial interest in testing dynamic orchestrations (Kattepur et al., 2011; Hummer et al., 2011, 2013).

In *choreography testing*, we identified six studies that deal with conformance testing (Baldoni et al., 2005; Nguyen et al., 2012), test generation (Wieczorek et al., 2009, 2010; Zhou et al., 2010), and coverage criteria (Mei et al., 2009a). Baldoni et al. (2005) propose a framework inspired in multi-agent systems for conformance testing between the peer (single service) behavior and the global behavior (choreography). Nguyen et al. (2012) adopt passive testing to support the verification of local and global conformance. Wieczorek et al. (2009) apply model checking to generate model-based integration tests for choreography models. The authors also report a case study in an industrial context (Wieczorek et al., 2010). Zhou et al. (2010) apply dynamic symbolic execution to generate test inputs and assertions to test WS-CDL programs. Mei et al. (2009a) propose some test adequacy criteria for service choreography specified in WS-CDL that manipulates XPath queries.

We identified only two studies that handle composition in a *generic* way, i.e., there is no distinction between orchestration and choreography. Ruth et al. (2007) propose an approach for safe regression testing through the usage of CFGs to identify the changes. Rabhi (2012) proposes the robustness testing of operations called within a composition by using sub-specifications augmented with rules.

There is a large difference between the number of studies based on orchestration (24 studies) and on choreography (six studies). The composition languages cited in the analyzed work are WS-BPEL (orchestration) and WS-CDL (choreography). A high number of studies in orchestration testing can be consequence of the maturity level of the composition languages. WS-BPEL has been accepted as a standard language for orchestration and business process. In contrast, there is no consensus on a choreography language. Currently, WS-CDL is the most cited one. Only the works of Ruth et al. (2007); Rabhi (2012) address the service composition in a generic way.

Modeling Techniques We discuss and group the main modeling techniques used to support the formal testing of service-oriented applications.

State Models: In this group, the studies use state-based modeling techniques to support the testing activity, ranging from simple to complex ones. These techniques are: Labeled Transition System (LTS) and extensions (Mei et al., 2009a; Bertolino et al., 2004, 2008; Frantzen et al., 2008; Bentakouk et al., 2009; Wieczorek et al., 2009, 2010; Rabhi, 2012; Escobedo et al., 2010; De Angelis et al., 2010), timed automaton and extensions (Lallali et al., 2008; Morales et al., 2010; Cavalli et al., 2010; Gao and Li, 2011), Stream X-Machine (SXM) (Dranidis et al., 2007; Ramollari et al., 2009; Kourtesis et al., 2010; Dranidis et al., 2010), extensions of FSMs (Benharref et al., 2006; Keum et al., 2006; Dong et al., 2010), finite state automaton (Baldoni et al., 2005), and Web Service Automaton (WSA) (Zheng et al., 2007). The modeling techniques are used to describe the control flow, data flow, timing, and stochastic properties. While most of the studies reuse or adapt established techniques, Zheng et al. (2007) propose the WSA to represent the semantic operation of the WS-BPEL language.

CFG and extensions: CFGs and well-known extensions like Def-Use Graphs (Section 2.3.1) have been constantly used to support structural testing. They have been mainly used to test WS-BPEL based service composition (Liu et al., 2008; Karam et al., 2007; Yan et al., 2006; Endo et al., 2008; Mei et al., 2008, 2009b; Li et al., 2008b; Eler et al., 2010). However, some studies employ CFGs to other test purposes. Li et al. (2008a) use CFGs to model the sequences of operations for testing stateful services. Ruth et al. (2007) and Mei et al. (2009c) apply CFGs and coverage criteria to support regression testing. Zhou et al. (2010) use CFGs to support the generation of test inputs and assertions to test WS-CDL programs.

GT rules: GT rules are used to augment the service specification with pre-conditions, effects, and a notion of states. A GT rule refines the service operation, adding information about the parameters and internal data. The service state can be recorded as a graph attribute, though GT rules is not a technique that models explicit states. As a consequence, GT rules based approaches are adequate for stateless and stateful services, supporting discovery, monitoring, automatic testing, and regression testing (Heckel and Mariani, 2005; Lohmann et al., 2007; Park et al., 2009).

Swiss Cheese model: Swiss cheese model is the basis for the work of Tsai et al. (2005b,c,a). It is a modeling technique based on Boolean expressions, extracted from semantic specifications (OWL-S), that are represented by Karnaugh maps and finally a Swiss Cheese map. The model is used to generate positive and negative test cases.

Other modeling techniques: We identified 12 studies which use modeling techniques that were not classified into the previous groups. In this context, we found: service composition model (Hummer et al., 2011, 2013), message sequence graphs (Hou et al., 2009; Ni et al., 2011), metamorphic relations (Chan et al., 2007), input element type model (Ma et al., 2008), ESGs (Belli and Linschulte, 2008), IOPE (Paradkar et al., 2007), BIR model (Corradini et al., 2008), POST Class graph (Chakrabarti and Rodriquez, 2010), feature diagram (Kattepur et al., 2011), and Chor (Nguyen et al., 2012).

Each study addresses the model construction in a specific form. There are two common strategies: *(i)* supposing that there exists a formal model and *(ii)* generating a model from existing service specifications. The first strategy is common in single service testing due to the fact that only syntactic specifications (WSDL, XML Schema) are frequently available. The second approach is predominant in service composition testing, since supposing that there are WS-BPEL or WS-CDL specifications in this context is reasonable. Semantic specifications are also considered in this way, albeit they are not commonly available.

Evaluation of the proposed approaches All the analyzed papers report some kind of study to evaluate the proposed approach. We opted by a generic classification defined by Do et al. (2005) in which the evaluations are divided into three types: controlled experiments, case studies, and examples. Figure 2.10 shows the percentage for each type of evaluation. Notice that a high number of studies were classified as example (49%) and case study (32%). It implies that few controlled experiments (19%) have been carried out. This fact indicates that more experimental research, mainly controlled experiments, should be conducted to evaluate the proposed approaches for formal testing of services.

Among the selected set, only two studies reported quantitative comparisons with similar testing approaches (Keum et al., 2006; Ni et al., 2011). We believe that two main facts hinder the execution of comparative studies: benchmarks and tools. First, the primary studies report experimental evaluation with their own programs and configuration. This fact hampers possible comparisons among the approaches. It would be interesting to establish a common benchmark containing a

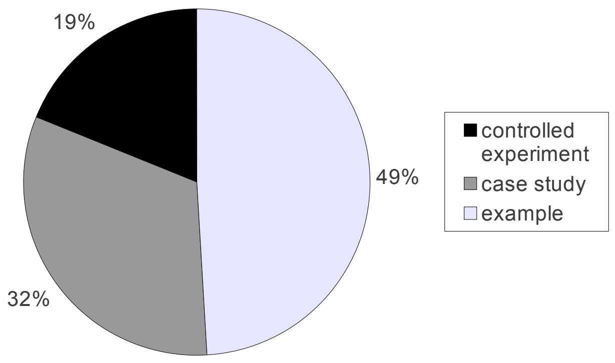


Figure 2.10: Types of evaluation.

set of services and necessary artifacts to overcome this limitation. Second, the existence of supporting tools would facilitate the execution of experiments, making this activity less manual and error-prone. However, few tools are reported in the papers and we found only three tools available for download: Jambition (Frantzen et al., 2008)[11], PUPPET (Bertolino et al., 2008)[11], and BPT (De Angelis et al., 2010)[12].

The literature on service testing also reports the issues aforementioned. Bozkurt et al. (2012) survey previous work on testing and verification of service-oriented applications, analyzing 177 papers. Their results show that 71% of the studies have no experimental validation, 18% use synthetic services, and 11% apply the approaches in real-world services. The authors conclude that one of the main problems in the topic is the lack of real-world case studies. We can argue that the high number of papers without experimental validation (71%) is also a major problem.

2.5.2 MBT of Service-oriented Applications

Based on studies selected in the systematic mapping, this section analyzes the MBT approaches for service-oriented applications.

Heckel and Mariani (2005) propose a high quality service registry that incorporates automated Web service testing before the registration. They use GT rules to specify the service behavior at the conceptual level (not the implementation level). The testing conformance is defined in terms of completeness (inside the input domain) and soundness (outside the input domain). The test case generation uses a domain-based strategy called partition testing.

Keum et al. (2006) propose the use of Extended Finite State Machines (EFSMs) to model Web services. They define a procedure to derive an EFSM by means of a WSDL specification. The procedure is based on forms filling and no tool is provided. The algorithm proposed by Bourhfir et al. (1997) is used to generate test cases, covering control flow and data flow of the model.

Dranidis et al. (2007) introduce a new approach to verify the conformance between the service implementation and a formal specification. The authors adopt SXMs to model the service behavior

[11]http://plastic.isti.cnr.it/wiki/tools – last accessed on 20/02/2013.
[12]http://bptesting.sourceforge.net – last accessed on 20/02/2013.

and generate test cases. The transitions in an SXM are labeled with processing functions, modeling the request, response, pre-conditions, and effects. The SXM testing method generates a complete suite of input sequences to verify the implementation.

Frantzen et al. (2008) present a tool, called Jambition, that generates on-the-fly test cases for Web services. The authors use the Symbolic Transition System (STS) technique to specify the functional aspects of a service. A service operation is related to a transition in the STS. There are three types of transitions: input (a message sent to the service), output (a message sent from the service), and unobservable. Variables and guard conditions can also be included. The testing approach implemented is random and online. The input data is generated based on constraints (guard transitions) using the constraint solver of GNU prolog. This data is executed and the tool selects randomly a new operation on-the-fly. Although these works are based on formal modeling techniques (Heckel and Mariani, 2005; Keum et al., 2006; Dranidis et al., 2007; Frantzen et al., 2008), they are only focused on single services that have complex and stateful functionalities and do not take into account composite services.

Benharref et al. (2006) propose a multi-observer architecture to detect and locate faults in Web service compositions. The architecture is composed of a global observer and local observers that cooperate to collect and manage faults found in the composite service. The authors use the concept of passive testing which is based on collecting and analyzing traces.

Zheng et al. (2007) propose a modeling technique called WSA to represent the operational semantics of the WS-BPEL. A WSA model is a finite state machine with signature, data structures, and message storage schema. As WSA has no hierarchy, the hierarchical dependencies are modeled using parent and child relationships among machines. Three models capture the data dependencies: *(i)* internal (a single WS-BPEL process), *(ii)* external (one process to another process), and *(iii)* global (union of internal and external). Control flow and data flow structural criteria are encoded using the temporal logics like Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). Using the WSA, inputs for the model checkers SPIN and NuSMV are generated. The model checking application for test case generation is based on recovering test cases by counter-examples.

Mei et al. (2009a) propose a modeling technique to describe a service choreography that manipulates the data flow using XPath queries. In a choreography, XPath queries can handle different XML schema files. XPath expressions are represented using XPath Rewriting Graph (XRG), which is a data structure that models the different paths defined in an XPath expression over a schema. Based on LTSs, the LTS-based Choreography model (C-LTS) is proposed with XRGs attached in transitions that represent service invocations. New types of definition-use associations are proposed and test adequacy criteria are presented.

Transforming composition specifications (such as WS-BPEL and WS-CDL) into formal models to support test generation has also been researched. Bentakouk et al. (2009) propose a mapping from WS-BPEL to STSs. Test cases are generated using symbolic execution and applied to the SUT using online testing. Hou et al. (2009) model a WS-BPEL program using message sequence graphs to generate test sequences. In an extended version (Ni et al., 2011), the authors formalize the approach and make an experimental comparison with two other approaches.

Wieczorek et al. (2009) present a model-based integration testing for service choreography using a proprietary model, called Message Choreography Model (MCM). MCM models are translated to Event-B (Abrial and Hallerstede, 2007) and test cases are generated using model checking. In (Wieczorek et al., 2010), the authors describe a case study about the application of this MBT approach to test service choreographies in a real-world project. MCM models were designed to support the tests.

Table 2.2 relates the approaches presented in this section with the four main steps of MBT (Section 2.4), showing which steps are mentioned by each work. Notice that most of the papers focus on the test generation step, which includes testing criteria, algorithms and tools. The test execution is the second most cited, which usually includes tools and software frameworks used to run the test suites into the SUT. We observed that the modeling and concretization steps, which require more manual effort from testers, are neglected in these works.

Table 2.2: Studies and steps of MBT.

	1- modeling	**2- test generation**	**3- concretization**	**4- test execution**
(Heckel and Mariani, 2005)		✓		✓
(Keum et al., 2006)	✓	✓		
(Dranidis et al., 2007)		✓		
(Frantzen et al., 2008)		✓		✓
(Benharref et al., 2006)				✓
(Zheng et al., 2007)	✓	✓		✓
(Mei et al., 2009a)		✓		✓
(Bentakouk et al., 2009)	✓	✓	✓	✓
(Hou et al., 2009; Ni et al., 2011)	✓	✓		
(Wieczorek et al., 2009, 2010)	✓	✓	✓	✓

The experimental evaluation of a testing approach should take into account two cost dimensions: human effort and CPU time (Briand, 2007). The cost reflects the effort necessary to apply a given testing approach. Classical studies in software testing have measured the cost using the number of test cases (Briand, 2007; Juristo et al., 2004). However, in MBT, much of the manual effort spent is concentrated in the modeling and concretization steps. The discussed approaches for service testing have shown case studies and experiments to evaluate their practical application, as well as the fault detection capability. However, we could not identify a detailed analysis of costs involved in applying these approaches, from modeling to test execution. These works provide the number of test cases as a cost measure which, as previously discussed, is not enough to infer the devoted effort. Moreover, the concretization is usually left out, hindering the complexity and costs to perform this step.

2.6 Final Remarks

This chapter has introduced the necessary background for the dissertation contributions described in remaining chapters. It has started with the concepts of SOA, as well as the Web service technologies. In the sequence, an overview of software testing and its terminology have been shown. We have focused on MBT because it is the main testing approach employed in this dissertation.

This chapter has also reviewed the literature on service testing. Initially, foundations of service testing have been discussed, considering main factors, levels of testability, and perspectives. Then, a systematic mapping on formal testing of services has been presented. We analyzed a set of studies that propose testing approaches for single and composite services, grouping them with respect to the adopted modeling technique and the type of evaluation. Finally, MBT approaches for service-oriented applications have been described in detail.

In this dissertation, we assume that the level of access to the service is the most common in practice, i.e., level 4: only the WSDL is accessible (Tsai et al., 2006). Moreover, we also consider that all (single and composite) services are black boxes, benefiting the testers' perspectives as provider, integrator, and certifier. The contributions we present in Chapters 3, 4, 5, and 6 aim at advancing the research on service testing and contribute to overcome some of the limitations discussed in this chapter.

We aim at applying MBT in service-oriented applications. When we take into account the application of MBT in any type of software, the selection of an appropriate modeling technique and its test generation algorithm is crucial. The next chapter describes an experimental comparison so that five FSM-based test generation methods are analyzed. The evaluation takes into account several dimensions relevant in practice, such as number of test cases, test case length, overall test suite length, and fault detection ratio.

CHAPTER

3

Comparing FSM-based Test Methods

3.1 Overview

In MBT, one of the crucial issues is to choose the modeling technique to describe the test models (Hartman et al., 2007). As discussed in Chapter 2, Section 2.5.1, several types of modeling technique have been applied in service-oriented applications. Although most of them are techniques based on states and transitions, FSMs have not been particularly investigated.

FSM-based testing is a topic studied for several decades (Moore, 1956; Gill, 1962), yet with recent advances (Simao and Petrenko, 2010b,a; Dorofeeva et al., 2010; Hierons and Ural, 2010; Pedrosa and Moura, 2012). Great effort has been spent on the development of methods that generate effective test suites, i.e., methods that detect as many faults as possible. A so-called *complete* test suite, capable of revealing all faults from a given fault domain in an implementation, can be generated from a specification if some assumptions are made. One of them is to assume that the maximum number of states in the implementation is known. When both specification and implementation have the same number of states, the generated test suite is called n-complete (Dorofeeva et al., 2005b). There are several methods in the literature that generate n-complete test suites, such as W, HSI, H, SPY, and P. These methods produce test suites with different characteristics that can only be experimentally compared. Few experimental studies on comparing different FSM-based test methods can be found in the literature (Dorofeeva et al., 2005a; Simao et al., 2009a; Dorofeeva et al., 2010). Moreover, they fall short when it comes down to considering recent contributions and analyzing different aspects relevant to the practical application of these methods in service-oriented applications.

This chapter presents an experimental study that compares five FSM-based test generation methods. We compare the test suites generated automatically using the traditional (W, HSI) and recent (SPY, H, P) methods. First, for test suites derived using the compared methods, we analyze the test suite characteristics: number of test cases (resets) and their length. Second, the total cost (i.e., the length) of each test suite is compared. Third, the effectiveness of the methods is analyzed using the mutation testing. Finally, we verify the correlations between the different dimensions analyzed.

This chapter summarizes the main results of paper *"Evaluating Test Suite Characteristics, Cost, and Effectiveness of FSM-based Testing Methods", Endo, A. T., Simao, A.,* published in the Information and Software Technology journal, DOI: 10.1016/j.infsof.2013.01.001 (Endo and Simao, 2013).

3.2 Preliminaries

A Finite State Machine (FSM) is a deterministic (Mealy) machine, defined as follows (Simao and Petrenko, 2010a). An FSM M is a 7-tuple $(S, s_0, I, O, D, \delta, \lambda)$, where:

- S is a finite set of states with initial state s_0,
- I is a finite set of inputs,
- O is a finite set of outputs,
- $D \subseteq S \times I$ is a specification domain,
- $\delta : D \to S$ is a transition function, and
- $\lambda : D \to O$ is an output function.

The tuple $(s, x) \in D$ is a *defined transition* in state s that consumes input symbol x. An FSM which has defined transitions for each input symbol in all states, i.e., $D = S \times I$, is complete. Otherwise, the FSM is partial. A sequence $\alpha = x_1...x_k \in I^*$ is an input sequence defined for state $s \in S$, if there exist $s_1, ..., s_{k+1}$ such that $s = s_1$ and $\delta(s_i, x_i) = s_{i+1}$ for all $1 \leq i \leq k$; we say that α is a *transfer sequence* from s to s_{k+1} and that s_{k+1} is *reachable* from s. An FSM is strongly connected if every state is reachable from all states. An FSM is initially connected if every state is reachable from initial state s_0. Figure 3.1 shows an example of a state diagram representation of an FSM.

Notation $\Omega(s)$ is used to denote all input sequences defined for state s and Ω_M as an abbreviation for $\Omega(s_0)$. Therefore, Ω_M represents all defined sequences for the FSM M. In this dissertation, we assume that the FSM has a reset operation that brings the machine to its initial state. The reset operation is denoted by r. Notation $\alpha\omega$ represents the concatenation of the two sequences, α and

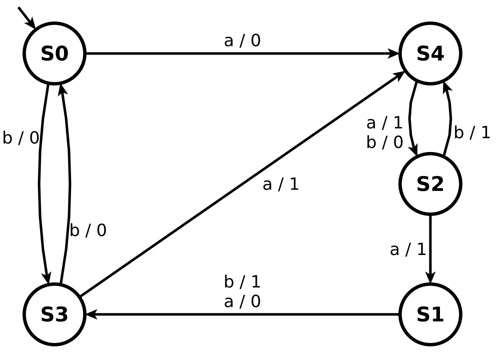


Figure 3.1: Example of an FSM – extracted from (Endo and Simao, 2013).

ω. Sequence α is prefix of sequence β, denoted by $\alpha \leq \beta$, if $\beta = \alpha\omega$, for some sequence ω. Sequence α is proper prefix of β, denoted by $\alpha < \beta$, if $\beta = \alpha\omega$ for some $\omega \neq \epsilon$. Given two sets of sequences D_1 and D_2, $D_1.D_2$ is the set of sequences obtained by concatenating all sequences in D_1 with all sequences in D_2, i.e., $D_1.D_2 = \{\alpha\beta \mid \alpha \in D_1 \text{ and } \beta \in D_2\}$.

The transition and output functions are extended for defined input sequences, including the empty sequence ϵ, as follows. For a state $s_i \in S$, $\delta(s_i, \epsilon) = s_i$ and $\lambda(s_i, \epsilon) = \epsilon$, given an input sequence $\alpha x \in \Omega(s_i)$, we have $\delta(s_i, \alpha x) = \delta(\delta(s_i, \alpha), x)$ and $\lambda(s_i, \alpha x) = \lambda(s_i, \alpha)\lambda(\delta(s_i, \alpha), x)$.

Two states $s_i, s_j \in S$ are *distinguishable* if there exists a *separating sequence* $\gamma \in \Omega(s_i) \cap \Omega(s_j)$, such that $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$. An FSM M is reduced (or minimal) if all states are pairwisely distinguishable. Given a different FSM $N = (Q, q_0, I, O, \Delta, \Lambda)$ with the same sets of inputs and outputs, we say that two machines M and N are distinguishable if there exists a sequence $\gamma \in \Omega_M \cap \Omega_N$, such that $\lambda(s_0, \gamma) \neq \Lambda(q_0, \gamma)$.

A test case of M is an input sequence $\alpha \in \Omega_M$ starting with symbol r. A test suite of M is a finite set of test cases of M, such that there are no two test cases α and β, such that $\alpha < \beta$. In fact, if a test case α is a proper prefix of a test case β, the execution of β will always imply the execution of α, and can thus be removed without altering the test result. The number of symbols (length) of a sequence α is represented by $|\alpha|$. This notation is extended for a test suite T, $|T|$, which is the sum of the length of its test cases.

Assuming a specification M, we denote by $\Im$ the set of all deterministic FSMs with the same input alphabet as M for which all sequences in Ω_M are defined, i.e., for each $N \in \Im$, $\Omega_M \subseteq \Omega_N$. Set $\Im$ is called a *fault domain* for M. Let $m \geq 1$ be an integer, $\Im_m$ denotes all FSMs of $\Im$ with at most m states. Given a specification M with n states, a test suite $T \subseteq \Omega_M$ is m-complete, $m \geq n$, if, for each $N \in \Im_m$ distinguishable from M, there exists a test in T that distinguishes M from N. An m-complete test suite has *full fault coverage* for the defined domain and is able to detect all faults in any implementation with at most m states. In this work, we consider n-complete test suites that represent the case in which $m = n$.

A set of input sequences Q is a *state cover* of M if, for each state $s_i \in S$, there exists a sequence $\alpha_i \in Q$ that transfers the FSM from the initial state to s_i. This set includes sequence ϵ

to reach the initial state. A set of input sequences P is a *transition cover* of M if, P includes the empty sequence ϵ and for each transition $(s, x) \in D$ there exist the sequences $\alpha, \alpha x \in P$ such that $\delta(s_0, \alpha) = s$.

To identify states and check transitions, traditional methods use predefined sets, such as characterization sets and separating families. A *characterization set*, also known as W set, is a set of defined input sequences that contains at least a separating sequence for each pair of states in the FSM. A *separating family* is a set of state identifiers H_i for a state $s_i \in S$ that satisfies the following condition (Luo et al., 1995): for any two different states s_i, s_j, there exist sequences $\beta \in H_i$ and $\gamma \in H_j$ that have a common prefix α, i.e., $\alpha \leq \beta$ and $\alpha \leq \gamma$, such that $\alpha \in \Omega(s_i) \cap \Omega(s_j)$ and $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$.

Recent methods, such as H, SPY, and P rely on sufficient conditions to support test case generation. The sufficient conditions provide mechanisms to check if a given test suite is m-complete. However, a test suite that does not satisfy these conditions may be m-complete, i.e., these conditions are not *necessary*. The sufficient conditions for the test suites found in the literature are described in (Dorofeeva et al., 2005b), (Simao and Petrenko, 2010a), and (Simao and Petrenko, 2010b) and used by the H, SPY, and P methods, respectively.

3.3 Test Generation Methods

In this section, we briefly introduce the W, HSI, H, SPY, and P methods used in the experiments. For each method, an overview of the test suite construction for the example in Figure 3.1 is also provided. All these methods are applicable to initially connected, reduced, complete, and deterministic machines. The HSI, H and P methods are also applicable to partial FSMs.

W The works of Vasilevskii (1973) and Chow (1978) are the seminal papers in test case generation methods for FSMs, proposing the W method. The method uses the transition cover set P to reach states and transitions and the characterization set W for state identification. If $m = n$, a test suite is generated by the concatenation of sets P and W and removal of the proper prefixes.

For the example in Figure 3.1, given that $P = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aaaa, aaab\}$ and $W = \{a, bb\}$, after $P.W$ and removing the sequences which are prefixes of other tests, we obtained the test suite $\text{TS}_\text{W} = \{raba, rabbb, rbaa\ rbabb, rbba, rbbbb, raaba, raabbb, raaaaa, raaaabb, raaaba, raaabbb\}$, $|\text{TS}_\text{W}|= 64$.

HSI The HSI method (Luo et al., 1995) uses the separating family to check states in both state identification and transition testing. The separating family can be obtained from a characterization set W, which, in the worst case, will be the W set itself.

Given the same P and the harmonized state identifiers $H_0 = \{a, b\}$, $H_1 = \{a, b\}$, $H_2 = \{a, b\}$, $H_3 = \{ab\}$, $H_4 = \{ab\}$, the test suite is obtained by concatenating all sequences $\alpha \in$

P with H_i such that $\delta(s_0, \alpha) = s_i$. After removing proper prefixes, we obtained the test suite $\text{TS}_{\text{HSI}} = \{raba, rabb, rbab, rbba, rbbb, raabb, rbaab, raaaab, raaabb, raabab, raaaaab, raa\ abab\}$, $|\text{TS}_{\text{HSI}}| = 62$.

H Similarly to the HSI method, the H method (Dorofeeva et al., 2005b) also adopts separating families and can be seen as an improved method. The difference is that the H method selects state identifiers on-the-fly during the transition testing phase (Dorofeeva et al., 2005b). Using this strategy, the method is able to reduce the length of the test suites produced.

The first part of the method adds sequences that distinguish reached states for each pair of sequence in Q. Given the state cover $Q = \{\epsilon, a, b, aa, aa\ a\}$ and the separating family $H_0 = \{a, b\}$, $H_1 = \{a, b\}$, $H_2 = \{a, b\}$, $H_3 = \{a, ba\}$, $H_4 = \{a, ba\}$, the first part is equivalent to $Q.H_i = \{ba, bba, aba, aab,\ aaaa, aaab\}$, after removing prefixes. Notice that the separating family used here is different from the one used in the HSI method. The second part verifies transitions with respect to the sufficient conditions. If transitions are not verified, adequate state identifiers are selected *on-the-fly*. For instance, to check transition (s_3, a) we have sequence ba such that $\delta(s_0, ba) = s_4$. We can apply the previously defined state identifier H_4, but sequence aaa has been chosen for this task. The resulting test suite for H method is $\text{TS}_{\text{H}} = \{rbba, rbbb, rabaa, rbaaaa, raaabaaa, raabaaa,\ raaaaaaa\}$, $|\text{TS}_{\text{H}}| = 42$.

SPY The SPY method (Simao et al., 2009b) was proposed to reduce the length of the test cases, using a strategy that shortens the number of test cases by avoiding creating new branches in the test suite. Thus, the SPY method also chooses state identifiers on-the-fly so that it avoids creating new branches by the extension of existing test cases. Using SPY, we obtained the test suites $\text{TS}_{\text{SPY}} = \{raaaaab, raaaab, raaabab, raaabb, raabab, raba, rbabb, rbba, rbbbaabb\}$, $|\text{TS}_{\text{SPY}}| = 53$.

P The P method (Simao and Petrenko, 2010b) was initially proposed to obtain an n-complete test suite by incrementing a user-defined test suite provided. However, if we consider that the user-defined test suite contains only the empty sequence, the P method can be used to generate n-complete test suites in the traditional way. The P method is also able to choose sequences on-the-fly to distinguish states. The method iteratively checks sufficient conditions and applies defined rules to derive the test suite.

The P method basically works in two steps. First, it builds a test suite so that the n states are distinguished. In the second step, the method checks the transitions not confirmed by the sufficient conditions. For each new test added to the test suite, rules can be triggered and other transitions can be checked. At the end, after removing prefixes and adding resets, we have the final test suite $\text{TS}_{\text{P}} = \{rababba, raabba, rabaaab, rabba, rbabb, rbbb\}$, $|\text{TS}_{\text{P}}| = 34$.

3.4 Experimental Study

We conducted experiments to evaluate test suites generated by different methods, from traditional (W, HSI) to recent ones (H, SPY, and P). We aimed at answering the following research questions:

- RQ1: What are the characteristics of test suites generated by different methods, concerning number of resets and test case length?
- RQ2: Which method has the lowest cost, i.e., test suite length? Are their cost differences significant?
- RQ3: Which method is more effective (higher fault detection) when the implementation has more states than estimated?
- RQ4: How are test case characteristics, test suite length and fault detection ratio correlated?

As there are various generation methods in literature, the tester needs to consider different variables when choosing one of them. RQ1 addresses the number of test cases and what expects from the length of test cases generated. RQ2 addresses the overall cost of the test suites, when the number of inputs applied to the SUT is the most important measure. RQ3 addresses cases where estimations about the implementation's states are not accurate. Finally, RQ4 addresses if these variables can be analyzed solely or they are dependent.

Reduced and deterministic FSMs were randomly generated, varying the number of states, inputs, outputs, and transitions. For each FSM configuration (#states, #inputs, #outputs, and #transitions), 100 machines were generated and the average measures were calculated. In total, 5200 FSMs were used in this experiment. The raw data collected and analyzed in this dissertation are available on the Internet (Endo and Simao, 2012c). For each test suite, the number of resets (or test cases) and the average test case length were calculated. For a given configuration, the average for these data was also calculated.

We compared the cost of the methods with respect to the test suite length (with resets and no proper prefixes). The reduction and reduction ratio were calculated comparing the test suite length with the W method. Given a method κ, the reduction ratio was calculated using the W method as a reference, i.e., $|\text{TS}_\kappa|/|\text{TS}_\text{W}|$, where $|\text{TS}_\kappa|$ is the average test suite length produced by method κ.

All compared methods are able to generate n-complete test suites. To evaluate the fault detection, we consider a scenario such that n-complete test suites will not detect all faults, i.e., when the implementation is not in fault domain $\Im_n$. We use a strategy based on high order mutation testing (Jia and Harman, 2008). First, a mutant m_1 with $n+1$ states was generated. Second, other mutants were produced applying other operators to m_1. Thus, second order mutants were used to evaluate the fault detection of test suites generated by the methods.

Using the measures collected, we employed the nonparametric Wilcoxon matched-pairs signed ranks test to verify if two methods are different for a significance level of 0.05. As alternative hypothesis, we consider that test suites generated by one method is greater than the ones generated by other. Moreover, we provide measures to analyze the effect size, which are applied to complement hypothesis tests, indicating practical significance, importance, or meaningfulness (Kampenes et al., 2007). To calculate the effect size, we employed *(i)* unstandardized measures represented by median and mean differences; and *(ii)* standardized measures represented by standard mean difference (SMD) d and Hedges' g (Kampenes et al., 2007). To analyze the dependence/relationship between two variables, we employed the Spearman rank order correlation coefficient.

3.5 Analysis of Results

3.5.1 RQ1: Test Suite Characteristics

In this section, we analyze two measures concerning the test suites generated for each method. Given a method κ, we define:

- Number of resets (NR_κ) as the number of resets in a test suite. It also represents the number of test cases since there exists a reset before each test case.
- Length of test case (L_κ) as the average length of test cases in a test suite. We also analyze the minimum and maximum test case lengths.

Number of Resets Figure 3.2 shows how the number of resets varies as a function of the number of states[1]. All methods present an approximated linear growth, having W the highest number of resets and P the lowest. The SPY and H methods have a quite similar number of resets, becoming different only after 20 states. There exists a strong positive correlation between the number of resets and number of states (over 0.98); thus, the methods tend to generate test suites with more resets when adding more states to the machine.

Minimizing the number of resets is a known strategy to reduce the final length of a test suite. It can be obtained, e.g., by reducing the number of branches in the test suite (see (Simao et al., 2009b) for more details). The SPY, H, and P methods clearly show the lowest numbers of resets, presenting small differences among them that can be explained by properties of their algorithms and sufficient conditions. HSI has a smaller number of resets in comparison with W, since the characterization set is usually greater than individual separation families. As a consequence, the concatenation with the transition cover will produce more test cases (resets).

[1]The machines have four inputs and four outputs, and the number of states varies from four to 30 states. This configuration is repeated in the following graphs.

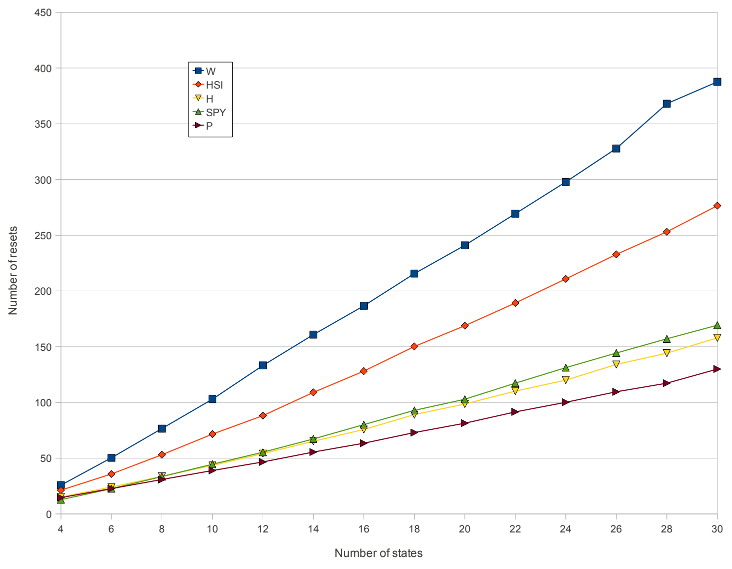


Figure 3.2: Number of resets varying the number of states – extracted from (Endo and Simao, 2013).

Test Case Length Figure 3.3 shows how the average test case length varies as a function of the number of states. Although a wide range of states was used, the variation in the test case length was not high. The minimum test case length ranges from 2 to 3.1; the average test case length ranges from 3.1 to 6.2; and the maximum test case length ranges from 3.8 to 12.3. There is a high positive correlation between the average test case length and the number of states (over 0.78); thus, the methods tend to generate longer test cases when adding more states to the machine.

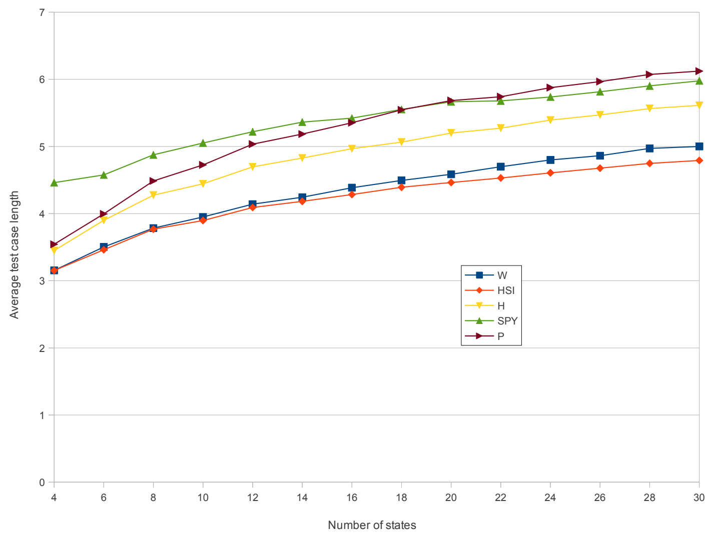


Figure 3.3: Average test case length varying the number of states – extracted from (Endo and Simao, 2013).

As the FSMs have more states, long sequences are necessary to reach and verify these states. However, this increase was not so accentuated. The HSI and W methods have the shortest test cases. H has an intermediate position when analyzing minimum, maximum, and average test case lengths. In minimum and average test case lengths, we observe that SPY and P have similar results. For maximum lengths, SPY shows the longest test cases with constant difference of over two inputs symbols with respect to the P method.

In general, W and HSI methods present similar test case lengths, with the shortest test cases among the methods. On average, HSI generates slightly shorter test case lengths than W. The SPY method presents the longest test cases in all configurations, followed by P and H. Table 3.1 shows the pairwise comparison between the average test case length of each method using 3200 complete FSMs. Each pair of different methods is compared, showing the alternative hypothesis which was considered (the null hypothesis was rejected), median and mean differences, SMD d, and Hedges' g. By the comparisons, the following order is observed and statistically supported $L_{SPY} > L_P > L_H > L_W > L_{HSI}$. The difference between W and HSI is small and they produce test cases that have virtually the same length on average. The SPY method has the longest test cases, though in practice this represents an average of ≈ 1.1 more input symbols than W and HSI.

Table 3.1: Pairwise comparison among the methods with respect to the average test case length – extracted from (Endo and Simao, 2013).

		W	HSI	SPY	H
HSI	alt. hypothesis	$L_W > L_{HSI}$	-	-	-
	median diff	0.035	-	-	-
	mean diff	0.081	-	-	-
	SMD d	0.141	-	-	-
	Hedges' g	0.141	-	-	-
SPY	alt. hypothesis	$L_{SPY} > L_W$	$L_{SPY} > L_{HSI}$	-	-
	median diff	1.2	1.234	-	-
	mean diff	1.1	1.173	-	-
	SMD d	1.88	2.19	-	-
	Hedges' g	1.88	2.19	-	-
H	alt. hypothesis	$L_H > L_W$	$L_H > L_{HSI}$	$L_{SPY} > L_H$	-
	median diff	0.544	0.579	0.655	-
	mean diff	0.507	0.588	0.585	-
	SMD d	0.801	0.997	0.987	-
	Hedges' g	0.801	0.997	0.987	-
P	alt. hypothesis	$L_P > L_W$	$L_P > L_{HSI}$	$L_{SPY} > L_P$	$L_P > L_H$
	median diff	0.817	0.852	0.381	0.273
	mean diff	0.812	0.893	0.280	0.304
	SMD d	1.196	1.398	0.436	0.442
	Hedges' g	1.196	1.398	0.436	0.442

3.5.2 RQ2: Test Suite Length

This section presents graphs and observations about the overall cost of the methods which is measured by the test suite length. Figure 3.4 shows how the test suite length varies as a function of the number of states. We observed that the test suite length grows as the number of states increases. This is shown by a strong positive correlation between the number of states and the test suite length (over 0.98). We also observed the order $|\mathrm{TS_W}| > |\mathrm{TS_{HSI}}| > |\mathrm{TS_{SPY}}| > |\mathrm{TS_H}| > |\mathrm{TS_P}|$. The difference among SPY, H, and P is small from four to 20 states, but becomes larger and constant after 22 states. We also analyzed the same data using the reduction ratio as the Y-axis. It varies until 12 states and presents a constant difference after 14 states. This constant difference represents a reduction of circa 30% for HSI, 50% for SPY, 55% for H, and 60% for P, in comparison with the test suites generated by the W method.

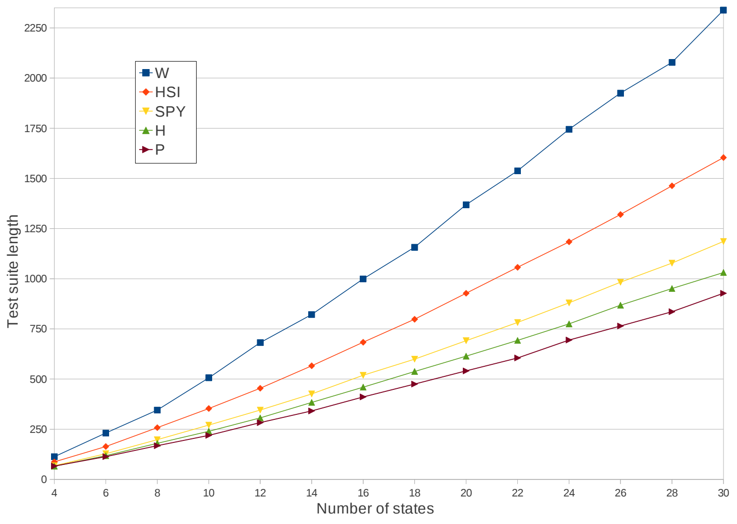


Figure 3.4: Test suite length varying the number of states – extracted from (Endo and Simao, 2013).

Figure 3.5 shows boxplots for each method in three configurations (four, 18, and 30 states); notice that the range of boxplots for FSMs with different numbers of states varies. In the configuration with four states, the recent methods present shorter test suites than W and HSI. However, the interquartile ranges of SPY, H, and P are quite similar. These method performances are to some extent discriminated with more states (18 and 30). Moreover, the method distributions seem to be more symmetric with the increase of states, except for the W method that has a skewed distribution in machines with 30 states. The W method also shows the highest number of outliers for the presented distributions.

Table 3.2 shows the pairwise comparison of the test suite length for each method using 3200 complete FSMs. Each pair of different methods is compared, showing the alternative hypothesis which was considered (the null hypothesis was rejected), median and mean differences, SMD d, and Hedges' g. The standardized effect sizes d and g are not defined when comparing the methods with W because this method was used to calculate the reduction ratio. Given that two methods κ_1 and κ_2, the notation $\kappa_1 >_{a(b)} \kappa_2$ means that κ_1 produces test suite lengths significantly longer than κ_2, a and b represent the median difference and the mean difference, respectively. Thus, we can observe the following expression $W >_{0.3(0.28)} HSI >_{0.17(0.16)} SPY >_{0.05(0.04)} H >_{0.04(0.03)} P$. Notice that the standardized effect size measures SMD d and Hedges' g are provided as an alternative measure. Although there is no reference values for them, the values presented here can be used in future research. For instance, a reasonable improvement from SPY over HSI (median difference of 0.17) has values for d and g of over 1.0, while small improvements, e.g., 0.04 for P over H, have values of around 0.3.

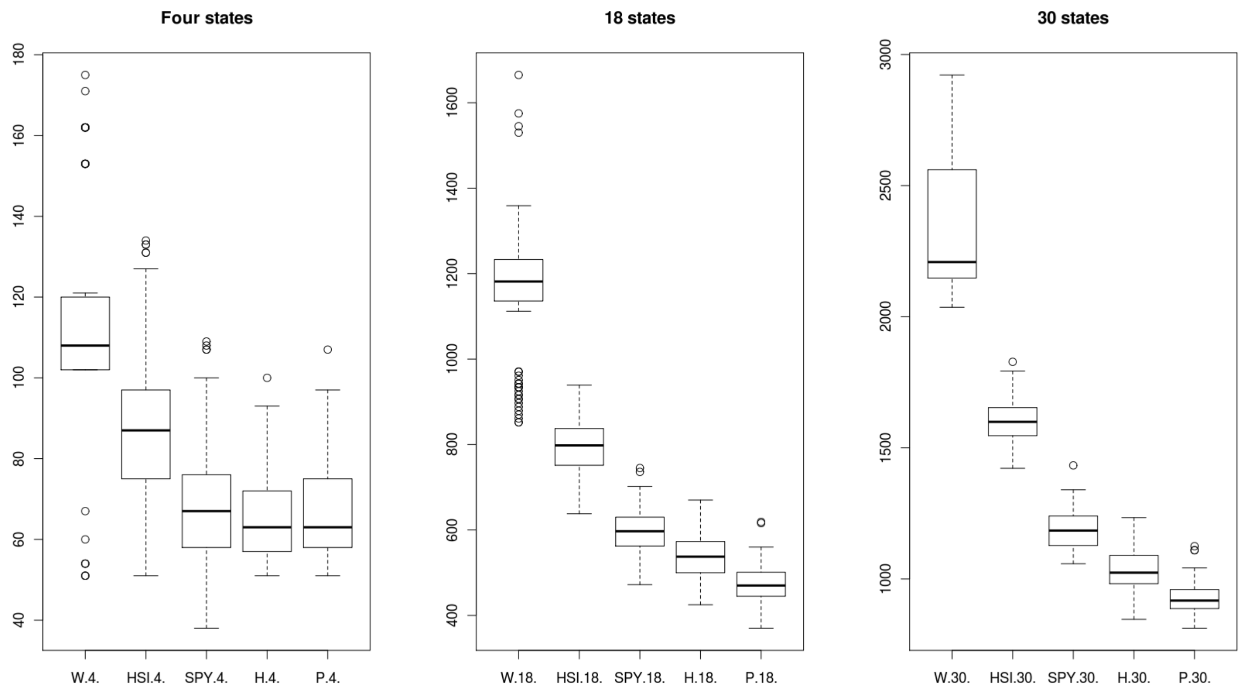


Figure 3.5: Boxplots for the test suite length varying the number of states (four, 18, and 30 states) – extracted from (Endo and Simao, 2013).

Table 3.2: Pairwise comparison among the methods with respect to the test suite length using reduction ratio over W – extracted from (Endo and Simao, 2013).

		W	HSI	SPY	H
HSI	alt. hypothesis	$\|TS_W\| > \|TS_{HSI}\|$	-	-	-
	median diff	0.30	-	-	-
	mean diff	0.28	-	-	-
	SMD d	-	-	-	-
	Hedges' g	-	-	-	-
SPY	alt. hypothesis	$\|TS_W\| > \|TS_{SPY}\|$	$\|TS_{HSI}\| > \|TS_{SPY}\|$	-	-
	median diff	0.47	0.17	-	-
	mean diff	0.45	0.16	-	-
	SMD d	-	1.11	-	-
	Hedges' g	-	1.11	-	-
H	alt. hypothesis	$\|TS_W\| > \|TS_H\|$	$\|TS_{HSI}\| > \|TS_H\|$	$\|TS_{SPY}\| > \|TS_H\|$	-
	median diff	0.52	0.22	0.05	-
	mean diff	0.50	0.21	0.05	-
	SMD d	-	1.51	0.40	-
	Hedges' g	-	1.51	0.40	-
P	alt. hypothesis	$\|TS_W\| > \|TS_P\|$	$\|TS_{HSI}\| > \|TS_P\|$	$\|TS_{SPY}\| > \|TS_P\|$	$\|TS_H\| > \|TS_P\|$
	median diff	0.56	0.26	0.09	0.04
	mean diff	0.54	0.25	0.08	0.03
	SMD d	-	1.78	0.69	0.30
	Hedges' g	-	1.78	0.69	0.30

3.5.3 RQ3: Fault Detection

Let TS be a test suite, $\#tm$ be the total of mutants, $\#km$ be the number of mutants killed by TS, and $\#em$ be the number of equivalent mutants, the *fault detection ratio* (FDR) of TS is calculated by $\text{FDR}_{\text{TS}} = \#km/(\#tm - \#em)$, also known as mutation score.

Figure 3.6 shows the FDR as a function of the number of states. All methods present a growth that is higher until 12 states and smaller afterwards. This is shown by a strong positive correlation between the number of states and the FDR for all methods (over 0.97); thus, the test suites tend to detect more faults (mutants) in machines with more states. All FDRs are superior to 0.9 for more than ten states. SPY has the highest FDR and HSI the lowest one. The P, H, and W methods present similar FDRs between HSI and SPY.

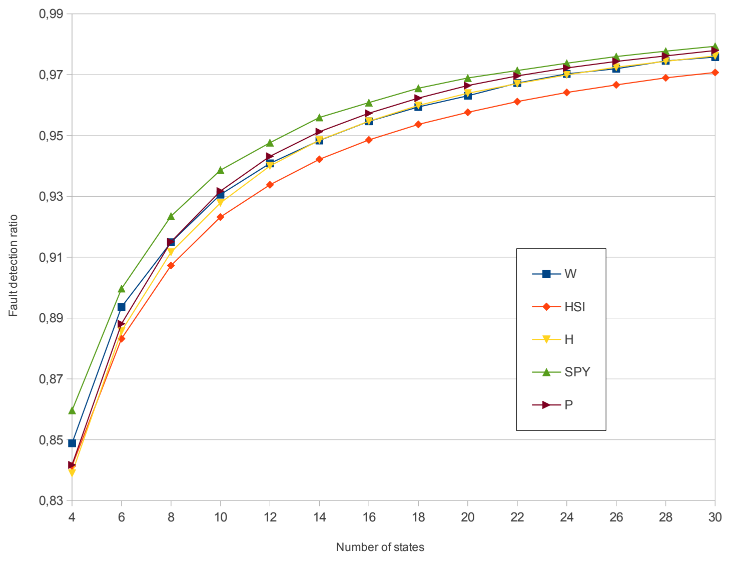


Figure 3.6: FDR varying the number of states – extracted from (Endo and Simao, 2013).

Table 3.3 shows the pairwise comparison of the FDR for each method using 3200 complete FSMs. Each pair of different methods is compared, showing the alternative hypothesis which was considered (the null hypothesis was rejected), median and mean differences, SMD d, and Hedges' g. By using the pairwise comparison, the following order is observed $\text{FDR}_{SPY} > \text{FDR}_P > \text{FDR}_W > \text{FDR}_H > \text{FDR}_{HSI}$.

All methods have a high average FDR in the interval $[0.9295, 0.9418]$. It means that most of the faults inserted by the mutation operators, even with a wrong estimation (-1) of number of states, are detected by n-complete test suites. Although it is statistically shown that the methods have different FDRs, the differences between them are quite small. For instance, the difference between the two methods in the limits, SPY and HSI, is around 0.01. In other words, SPY detects an average of 1% more faults (mutants) than HSI.

These differences between the methods' FDRs are relevant and can represent many undetected faults since the universe of faults is huge. For example, an average of $484,871$ mutants was produced for FSMs with 30 states, four inputs, and four outputs. This means that it is likely that by using SPY instead of HSI, 4849 more faults can be detected.

Table 3.3: Pairwise comparison among the methods with respect to the FDR – extracted from (Endo and Simao, 2013).

		W	HSI	SPY	H
HSI	alt. hypothesis	$\text{FDR}_W > \text{FDR}_{HSI}$	-	-	-
	median diff	0.007	-	-	-
	mean diff	0.007	-	-	-
	SMD d	0.27	-	-	-
	Hedges' g	0.27	-	-	-
SPY	alt. hypothesis	$\text{FDR}_{SPY} > \text{FDR}_W$	$\text{FDR}_{SPY} > \text{FDR}_{HSI}$	-	-
	median diff	0.006	0.01	-	-
	mean diff	0.005	0.01	-	-
	SMD d	0.20	0.49	-	-
	Hedges' g	0.20	0.49	-	-
H	alt. hypothesis	$\text{FDR}_W > \text{FDR}_H$	$\text{FDR}_H > \text{FDR}_{HSI}$	$\text{FDR}_{SPY} > \text{FDR}_H$	-
	median diff	0.003	0.003	0.01	-
	mean diff	0.002	0.004	0.008	-
	SMD d	0.10	0.16	0.31	-
	Hedges' g	0.10	0.16	0.31	-
P	alt. hypothesis	$\text{FDR}_P > \text{FDR}_W$	$\text{FDR}_P > \text{FDR}_{HSI}$	$\text{FDR}_{SPY} > \text{FDR}_P$	$\text{FDR}_P > \text{FDR}_H$
	median diff	0.0006	0.006	0.007	0.003
	mean diff	0.0002	0.006	0.005	0.002
	SMD d	0.01	0.26	0.21	0.09
	Hedges' g	0.01	0.26	0.21	0.09

3.5.4 RQ4: Correlations

In this section, we show the correlations between the aspects analyzed in the questions RQ1, RQ2, and RQ3. Hence, we analyze the correlations between number of resets (RQ1), test case length (RQ1), test suite length (RQ2), and FDR (RQ3) for each method. Table 3.4 shows the correlations between the variables analyzed in this study for each method in complete machines.

Table 3.4: Correlations between the variables analyzed – adapted from (Endo and Simao, 2013).

Line	Variable 1	Variable 2	W	HSI	SPY	H	P
1	avg test case length	#resets	0.24	0.24	0.36	0.24	0.32
2	avg test case length	test suite length	0.35	0.33	0.45	0.35	0.45
3	# resets	FDR	0.40	0.41	0.48	0.42	0.42
4	test suite length	FDR	0.48	0.47	0.53	0.51	0.52
5	# resets	test suite length	0.99	0.99	0.99	0.99	0.98
6	avg test case length	FDR	0.88	0.85	0.74	0.87	0.89

Line 1 shows the correlations between the average test case length and the number of resets. We observe a weak positive correlation for all methods. Line 2 shows the correlations between the average test case length and the test suite length. We observe a moderate positive correlation.

Line 3 shows the correlations between the number of resets and the FDR. Notice a moderate positive correlation. Intuitively, adding more test cases (resets) will likely increase the FDR. However, this is not so evident for the observed results. Line 4 shows the correlations between the test suite length and the FDR. The results are quite similar to the correlation between number of resets and FDR (Line 3). More test cases do not necessarily guarantee that more faults will be detected.

Observe that the four correlations analyzed (Lines 1-4) do not show a consistent behavior, since the correlations are weak or moderate. Notice that this is not the case for the next two analyses.

Line 5 shows the correlations between the number of resets and the test suite length. There exists a strong positive correlation for all methods (over 0.98). As there are more resets (test cases) in an n-complete test suite, the test suite length tends to be greater. Although this is obvious for test suites in general, we are analyzing specific n-complete test suites.

Line 6 shows the correlations between the average test case length and the FDR. We observe a high positive correlation (over 0.74). These results support that n-complete test suites with longer test cases tend to detect more faults in the considered scenario.

3.6 Discussion of Results and Limitations

As aforementioned, the FSMs were randomly generated. In this case, the similarity between these FSMs and the ones used in practice is unclear. Another issue is that a randomly generated FSM can represent a very rare and special case in which one of the methods does not perform well. To overcome this threat, we use a high number of different FSMs for each configuration and calculate average measures, aiming to reduce the influence of this factor on the results.

A limitation of this study is that we did not compare all existing methods (in the literature) able to generate n-complete test suites. As we selected a subset of the existing methods, other methods like DS, UIO, and Wp were left out. However, we believe that a representative set was selected, from the traditional methods to recent advances in the topic. Moreover, all the experiment data is available to facilitate the comparison with other test case generation methods (Endo and Simao, 2012c).

We have also adopted mutation testing to simulate the faults in the experiments. There is no consensus on the representativeness of mutants with respect to actual faults, albeit some studies give evidences that results using mutation operators are trustworthy (Andrews et al., 2005b; Do and Rothermel, 2006). Particularly, the employed operators represent common faults in FSM specifications and were exhaustively applied in the machines. Moreover, all equivalent mutants were removed, reducing the inconsistent data.

As stated by Arcuri (2010), there has been little research on the impact of the length of test cases. In this study, we provided a set of measures that can be used to shed light on this topic in the context of FSM-based testing. The recent methods tend to produce test suites with longer test cases, while traditional methods produce more (by the resets' data) and shorter test cases. This information can be useful for testers to support decision making during the MBT process. In general, shorter test cases are easier to execute and debug and can be more appropriate if they are executed by hand. However, if tests are automated and the cost is more important, the overall length of the test suite should be considered.

The results presented have some practical implications. There is a significant cost difference among the traditional and recent methods (around 50%). Even if we consider only recent methods, their reduction ratio varies from 50 to 60%. For instance, the mean difference between P's and H's test suite lengths (in FSMs with 30 states, 4 inputs, and 4 outputs) is 103.6. In other words, the tester using H will execute a test suite with around 103 more inputs (system's events) than one using P. This difference can be significant for models with many states, an SUT with an expensive test execution environment, or projects in which test sequences are executed by hand.

The fault detection ratios of test suites generated by the methods are quite close, though they are statistically different when pairwisely compared. A small percentage of undetected faults can represent a large number of faults since the fault domain is huge. Moreover, MBT has been applied in safety-critical software with special attention to embedded systems (Zander et al., 2011). In this scenario, a small optimization of the fault detection ratio might be of high practical importance for safety-critical software (Kampenes et al., 2007).

The set of analyses about the methods' behavior in different configurations can support the choice of the best method to fit a given project. The tester can identify the tradeoffs between test case characteristics, test suite length, and fault detection ratio. For instance, the tester may want a method that produces n-complete test suites with shorter test case lengths and generated from partial FSMs. In this case, the HSI method would be the best choice. On the other hand, the SPY method would be better to detect faults when the estimated number of states of the implementation is likely wrong. Other decisions can be supported by the results presented in this dissertation. Moreover, the data used in the experiments is available on the Internet (Endo and Simao, 2012c) and can be used to replicate this study or support the comparison with a new method proposed.

3.7 Final Remarks

This chapter has presented an experimental evaluation of test case generation methods for FSMs. The W, HSI, SPY, H, and P methods have been compared. On average, SPY produced the longest test cases and HSI has the shortest ones. The recent methods resulted in shorter test suites compared with traditional methods. The P method generated the shortest test suites, although the difference was smaller in comparison with H and SPY. In the analyzed scenario, all methods showed a high average fault detection ratio of over 92%. The SPY and P methods presented the highest fault detection and H and HSI had the lowest ratio. Different correlations have also been identified, especially the positive correlations between the number of test cases and the test suite length, and between the test case length and the fault detection ratio.

The results presented in this chapter can support the choice of an FSM-based test method to be applied in a given context. We select the P method as the more appropriate for service-oriented applications. P has the shortest test suite length, and although it is the second most effective, P is also applicable to partial machines (which are commoner in practical scenarios). While the modeling technique and the test generation method are worth from an MBT point of view, the testing process should also be researched for applications developed using SOA and Web services. The next chapter describes an MBT process for service-oriented applications so that FSMs are used for modeling and test case generation.

CHAPTER
4

MBT Process of Service-Oriented Applications

4.1 Overview

As discussed in Section 2.5, the testing activity is essential for ensuring the quality of services (Canfora and Di Penta, 2009; Bozkurt et al., 2012). There are several studies that propose the formal testing of services by using different types of modeling techniques, such as state models (Keum et al., 2006; Mei et al., 2009a), CFG and extensions (Li et al., 2008b; Mei et al., 2008), GT rules (Heckel and Mariani, 2005; Lohmann et al., 2007), and others (Chan et al., 2007; Hou et al., 2009); see a systematic mapping on this topic in Section 2.5.1 (Endo and Simao, 2010a,b). In particular, MBT has been researched to test service-oriented applications, as shown in Section 2.5.2. Although those studies provide approaches to test service-oriented applications, they are not concerned about establishing a process and describing the necessary steps in detail.

These approaches also use state models mostly to describe complex interactions among the services, albeit FSMs have not been researched in depth. FSMs have been widely studied by the software testing community and applied to various software domains (Lee and Yannakakis, 1996; Hierons et al., 2009), and several test generation methods have been proposed and experimentally compared, as shown in Chapter 3.

This chapter describes an MBT process for service-oriented applications using state models. In this study, the adopted modeling technique is FSM and the test case generation method is P (Simao and Petrenko, 2010b). The foundations of FSM-based testing and the rationale for selecting the

P method are described in Chapter 3. We also evaluate the process instantiation by means of the development of prototype tools and the conduction of a case study.

This chapter is a summary of paper *"Model-based Testing of Service-Oriented Applications via State Models", Endo, A. T., Simao, A.,* published in the Proceedings of the 8^{th} IEEE International Conference on Service Computing (SCC'11), DOI: 10.1109/SCC.2011.77 (Endo and Simao, 2011).

4.2 Motivating Example

In this section, we introduce an example to illustrate the issues we address in this chapter. `ThirdPartyCall-SOA` is an application based on the third party call service proposed by the Parlay-X services (Open Service Access, 2009). We have extended it with extra functionalities and modules to consider more SOA concepts. Figure 4.1 presents the communication in the `ThirdPartyCall-SOA` application. It is composed of three services:

- *Third Party Call Service* (TPCS): this service contains the main functionalities for a third party application to manage call sessions among different participants. Seven operations are available in the WSDL interface of this service. TPCS provides an extra operation to receive notifications about the participants connected in a call.

- *Call Monitor Service* (CMS): this service represents an instance of a monitor that observes the participant's status in a call. CMS provides two operations to respectively subscribe and unsubscribe in a monitoring process of a participant's status. This status can be connected, hang up, not reached, and busy.

- *Service Registry* (SR): this service is a registry specific to provide information on the available CMS services. SR was implemented as an instance of the Apache jUDDI (Apache.org, 2013), an open-source Java implementation for the UDDI standard.

A typical interaction with `ThirdPartyCall-SOA` is as follows. Initially, some client application starts a call session with one or two participants. For instance, a stock quote monitoring application starts a call session between a stockbroker and a customer because some stocks reach a threshold value (Open Service Access, 2009). TPCS invokes an SR operation that searches for available CMS services. TPCS selects an instance of CMS and subscribes for each participant using CMS operations. At this point, CMS dynamically creates a stub to access the notification operation of TPCS. Thus, if some participant's status changes, CMS uses this stub to notify TPCS. CMS was implemented to simulate the possible status' changes using random choices. When a call is established, different actions can be made via TPCS operations, such as add/remove/transfer participants, query the call/participant's status, and end the call session.

The application of MBT in service-oriented applications, such as `ThirdPartyCall-SOA`, is motivated by the potential for automation. In the adaptive and dynamic context of SOAs, it is

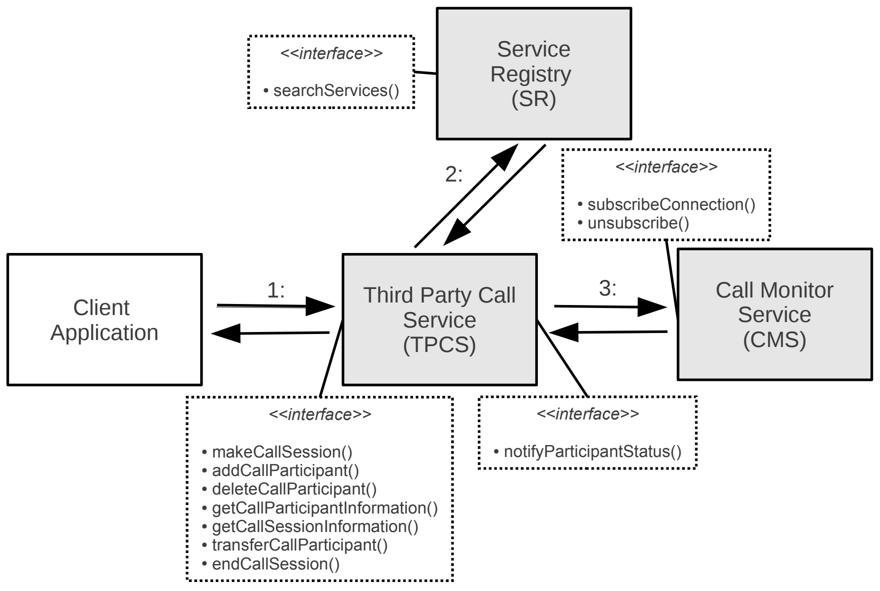


Figure 4.1: The `ThirdPartyCall-SOA` application – adapted from (Endo and Simao, 2011).

important that tests are automated and can be easily changed. MBT usually reacts promptly to changing requirements and can be an interesting solution in this context. However, the distribution of services over the Internet and the usage of XML standards hinder the information gathering during the tests and the establishment of a test harness. For instance, in a SOA environment it is difficult to monitor other integrated services, besides the service under test.

TPCS provides a set of operations that can change the service state. Moreover, there are interactions with other services, such as SR and CMS. These interactions can be modeled by using state models. Generating tests that cover state models and check their states is an accurate way to verify these interactions. In `ThirdPartyCall-SOA`, there are different aspects that should be tested, such as individual services and their interaction. Considering all these aspects in one state model would increase its complexity and size. In MBT, it is particularly necessary to manage the complexity and size of the models to avoid the state space explosion, while meaningful tests can still be generated. As noted in the example, we can build complex models even for simple service-oriented applications. For instance, `ThirdPartyCall-SOA` has different contexts that should be tested, e.g., the participant's status, number of participants in a call session, and the communication between TPCS and CMS.

There are also issues on using MBT and state models to test a service-oriented application like `ThirdPartyCall-SOA`. The test harness is complex due to the distributed nature of SOAs and the adoption of XML standards. The order of invocations and the state of the services are important to assure the reliability of service-oriented applications. This results in complex and large state models and, thus, limits the tests and the model maintainability.

4.3 Testing Process

Motivated by the issues discussed in the previous section, we propose a model-based process to test service-oriented applications with state models. We add elements to the process that were necessary in a SOA context. Elements present in generic MBT processes (Utting and Legeard, 2006; Pretschner and Philipps, 2004) are revisited, highlighting the differences whenever necessary. Figure 4.2 represents the elements of the process as artifacts (Section 4.3.1), tools (Section 4.3.2), and the services under test. We also discuss the steps of MBT in service-oriented applications in Section 4.3.3.

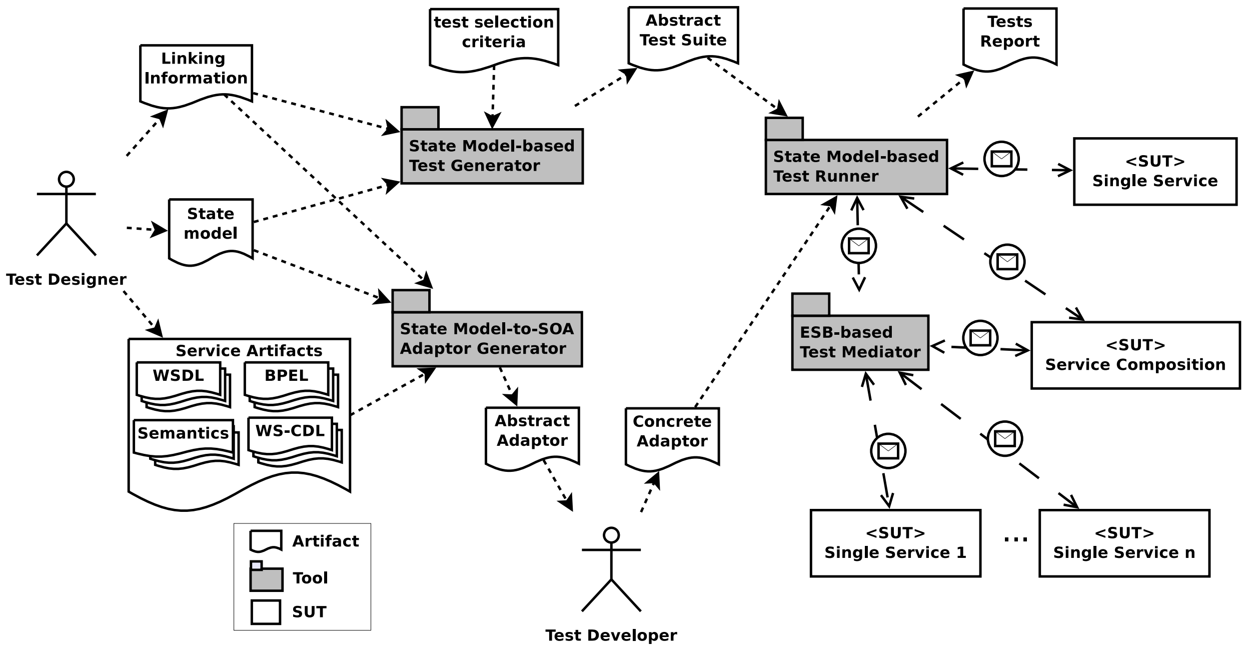


Figure 4.2: State-based testing process for service-oriented applications – extracted from (Endo and Simao, 2011).

4.3.1 Artifacts

There are three types of artifacts in the proposed testing process: *(i)* legacy, *(ii)* manual, and *(iii)* generated artifacts. Legacy artifacts are resources produced during the development of service-oriented applications. In Figure 4.2, we refer to them as service artifacts, consisting of interface descriptions (e.g., WSDL), composition specifications (e.g., WS-BPEL, WS-CDL), and semantic information (e.g., documents in Web Ontology Language for Web Services – OWL-S).

Manual artifacts are produced during the process by the testers. In Figure 4.2, the manual artifacts are the state model, the test selection criteria, the concrete adaptor, and the linking information. Linking information is an artifact that connects elements of the test model with the service artifacts. This artifact is important during the testing process because it includes mechanisms that enhance the automation, providing information to automatically generate code used in the test harness.

Generated artifacts are automatically produced by the supporting tools. In Figure 4.2, they include the abstract test suite, the abstract adaptor, and the tests report. The idea of the abstract adaptor artifact is to provide the necessary infrastructure for the tester to implement the concrete adaptor.

4.3.2 Supporting Tools

In the process, artifacts are processed by tools that produce new artifacts. Figure 4.2 presents the software tools needed to automate some tasks of the process. We describe each of them as follows.

State Model-based Test Generator: this tool receives as input the state model and test selection criteria and generates abstract test cases. It is desirable that test selection criteria are already implemented in the tool in order to automate the test case generation. The linking information artifact can also be considered during the test case generation. In this case, the algorithms and test criteria should be adapted to use the information included in this artifact.

State Model-to-SOA Adaptor Generator: this tool aims at decreasing the effort to develop the adaptor. It also reduces the complexity of the test harness, generating as much test code as possible. Thus, the tester can focus only on the implementation of the concrete adaptor, leaving the tedious and error-prone work to the tool. The state model, linking information, and service artifacts are received as input to generate an abstract adaptor that can be realized in a flexible way. However, the abstract adaptor should contain pre-defined and fixed structures used to enable the test execution by State Model-based Test Runner.

State Model-based Test Runner: this tool is responsible for mapping the abstract test cases to executable ones using the concrete adaptor. It is able to invoke operations of single and composite services under test. As output, it produces a tests report detailing the execution of each test case. In complex test scenarios involving integration with other services, it communicates with ESB-based Test Mediator to obtain more information and analyze the results.

ESB-based Test Mediator: this tool is essential when the service-oriented application involves a service composition. It can be used in two modes: monitoring and simulating. In the monitoring mode, it can be used to observe and record SOAP messages and to provide the necessary data for the State Model-based Test Runner tool evaluates the tests. In the simulating mode, the mediator works as stubs for services that are unavailable or under development. This tool is integrated with an ESB, using its capabilities to monitor and simulate the involved services. If the service-oriented application is already based on an ESB, the mediator can be included in the bus as a component. Otherwise, more configurations for deploying the SUT in the ESB are needed, though this task can also be automated. Scripts may be used to deploy the services under test in the ESB.

4.3.3 Steps of the Process

After describing artifacts and tools, we discuss each step of the MBT process for service-oriented applications using state models.

1. Selection of a test scenario: It is usually infeasible to create and maintain only one test model for the entire application. In order to circumvent such limitation, a specific part or feature of the SUT should be selected, i.e., to define a test scenario. Thus, the tester should identify test scenarios that are complex enough to create a model and that are reasonably simple to avoid too many states and transitions. As consequences of these scenarios, meaningful tests can be generated and the state space explosion problem is controlled.

Identifying and specifying test scenarios are helpful for SOAs since the tester can divide the tests into less complex contexts. For instance, a test scenario can be initially defined for each service. If a single service is too complex, two or more scenarios can be selected for this service. In a later step, more scenarios can be selected for the service composition. Moreover, the tester can use the available artifacts, such as interface specifications (WSDL) and composition descriptions (WS-BPEL, WS-CDL), as sources of information to define test scenarios.

2. Building of the test model: The tester designs a state model to represent the behavior observed in the test scenario. In this context, we assume that the model is created exclusively to support the tests. This assumption will guarantee a redundancy necessary to the testing: the intended and the actual behavior (Pretschner and Philipps, 2004). Moreover, reusing analysis and design models may not be possible since some service-oriented applications are defined and assembled dynamically in an agile context. This step should be performed by a tester with more skills in modeling (we refer to as test designer in Figure 4.2).

Considering the SUT as a black box, a state model can be designed by controlling inputs and observing outputs. Thus, the model abstracts away the complexity and the tester can focus on states and transitions that compose the test scenario. Thus, the tester can concentrate exclusively on the modeling activity. Although there are a couple of differences between single services and compositions, state models are applicable to both contexts (Keum et al., 2006; Mei et al., 2009a). State models also help in testing dynamical services since, even though the real services that will be integrated are unknown, it is possible to model its abstract behavior.

At this step, the linking information artifact is produced connecting test models and service artifacts. The service artifacts are used as supporting resources. In a composite service, the presence of the composition description can help in the modeling process. A model can be partially or completely derived from these descriptions. Another possibility is to use these descriptions to verify the models built by the tester.

3. Definition of test selection criteria: Usually, an infinite number of test cases can be generated from a model. Thus, it is necessary to restrict the size of the test suite by defining test selection criteria. These criteria are usually related to model coverage and are implemented by algorithms that traverse the model. Examples of these algorithms (methods) are shown in Chapter 3. This step can be completely automated if standard test selection criteria are adopted and implemented in the State Model-based Test Generator tool.

4. Generation of abstract tests: This step is automated using the State Model-based Test Generator and State Model-to-SOA Adaptor Generator tools. These tools produce a set of abstract test cases and an abstract adaptor. Abstract test cases are derived from state models as input/output sequences. Possible changes in the SUT are handled by regenerating tests for a updated test model. This step and the following ones can be performed by testers with other skills (we refer to as test developer in Figure 4.2).

5. Concretization of tests: In this step, the abstract test cases are transformed into executable ones. An adaptor that mediates the level of abstraction between the model and the SUT is developed. The tester uses the abstract adaptor as the initial point to concretize the tests. As interface descriptions (WSDL) are common in Web services, much effort is saved since the source code used to interact with the service can be automatically generated. If the tester connects test models and service artifacts using the linking information artifact, more effort can be saved.

6. Execution of tests: This step uses the abstract test suite and the concrete adaptor to execute the tests on the SUT. The State Model-based Test Runner and ESB-based Test Mediator tools perform this step automatically.

7. Analysis of results: The State Model-based Test Runner tool also automates this step. If some fault is detected, the tester can use a set of resources/artifacts to identify and correct it, such as abstract and concrete test cases, recorded SOAP messages, and the state model.

8. Verification and validation of produced artifacts: This orthogonal step verifies and validates artifacts produced along the process. State models and concrete adaptors are the main artifacts that need verification and validation in this step. The state model needs to be validated with respect to the requirements of the service-oriented application. As an advantage of MBT, the model can be built in initial stages of the development process and faults can be revealed earlier. If there exist semantic or composition descriptions, these artifacts are an additional information to verify the test model. The concrete adaptor should be also verified since, as a program, it can also include faults. The execution of the tests (Step 6) can be used as a test for the adaptor.

4.4 Exploratory Study

We conducted an exploratory study to analyze the MBT process for service-oriented applications. The goal is to provide an initial evaluation of the testing process concerning the level of automation and its practical usage in service-oriented applications. We divided the study into two parts: *(i)* we developed a prototype tool, named JStateModelTest, to instantiate the tools proposed in the process; and *(ii)* a case study was performed using two service-oriented applications.

4.4.1 JStateModelTest

The JStateModelTest tool is composed of four modules, namely `test-generator`, `adaptor-generator`, `runner`, and `mediator`, following the tools proposed in the testing process shown in Section 4.3.2. The tool, with its four modules, has around 3500 Lines of Code (LoC). We developed the tool using the programming language Java and its available frameworks for Web services. We also assume that Java is the programming language used to write the tests. The modeling technique supported by the tool is the Mealy FSM presented in Chapter 3. Each module is described in detail as follows.

`test-generator`: This module implements the P-method (Simao and Petrenko, 2010b) for test case generation from FSMs. The method is able to generate a full fault coverage test suite and also includes optimizations to generate shorter test sequences. It is also able to increment user-defined test suites, enhancing their fault detection capability. An example for this method can be found in Section 3.3. Although we chose the fault coverage, it is not difficult to implement other test selection criteria or integrate external tools. The only restriction is to generate abstract test suites in a format readable by the runner.

`adaptor-generator`: The basic idea to implement adaptors is to associate the adaptor with a Java class and input/output symbols with its methods. These associations are done with Java annotations. The annotation information is kept at runtime by the JVM, because the `runner` module needs to access this information to run the tests. As a consequence, we have a simple and flexible way of creating concrete adaptors. Moreover, `adaptor-generator` is able to create an abstract adaptor (template) from the test model. The necessary structure is generated and the tester just needs to implement the input and output events. Another possibility is to use the adaptor class as a facade for developing complex interactions with the SUT.

`runner`: This module uses Java reflection mechanisms to call the corresponding method for each input or output being tested. This information is retrieved by the usage of Java annotations. Each input and output of a test case is called in sequence and if, in the end, all respective methods returned true, the test case passed. Otherwise, `runner` stops the test case execution and shows a fail message with the problematic input or output. Another functionality is a timeout threshold that can be assigned, limiting the execution time of each input and output.

`mediator`: This module was based on Mule-ESB (MuleSoft, 2012). To use the `mediator` module, it is necessary that all services whose messages will be monitored be routed by ESB, i.e., all the communication with these services will first pass through the bus. We developed this module as a Web service integrated with the Mule ESB, providing operations to retrieve messages (*monitoring*). It is also possible to intercept and produce different messages to simulate certain situations (*simulation*), such as timeout occurrences and emulating unavailable services.

A Java interface can be added to an adaptor class through dependency injection (the `runner` module is responsible for adding the dependency at runtime). Thus, specific messages can be accessed and checked inside the adaptor written by the tester. Another characteristic is that other ESBs can be used without changing the developed adaptors. A new mediator may be developed for a different ESB by implementing the standard mediator interface and configuring the `runner` module.

4.4.2 Case Study

In this section, we present a case study aiming at evaluating the feasibility of the testing process. We used the JStateModelTest tool to support the case study conduction. In this study, two service-oriented applications were tested: `ThirdPartyCall-SOA`, a small application shown in Section 4.2; and `QualiPSo-Factory` (QualiPSo, 2010), a more complex and real-world application.

ThirdPartyCall-SOA We selected two scenarios: *number of participants* and *participant's status*. The former verifies the number of participants in a call session of the TPCS service. This scenario involves different ways to increase/decrease the number of participants, besides restrictions like a maximum number of participants and session termination after removing all participants. The latter models the possible participant's status during a call session. Figure 4.3 depicts the FSM test model for the *number of participants* scenario.

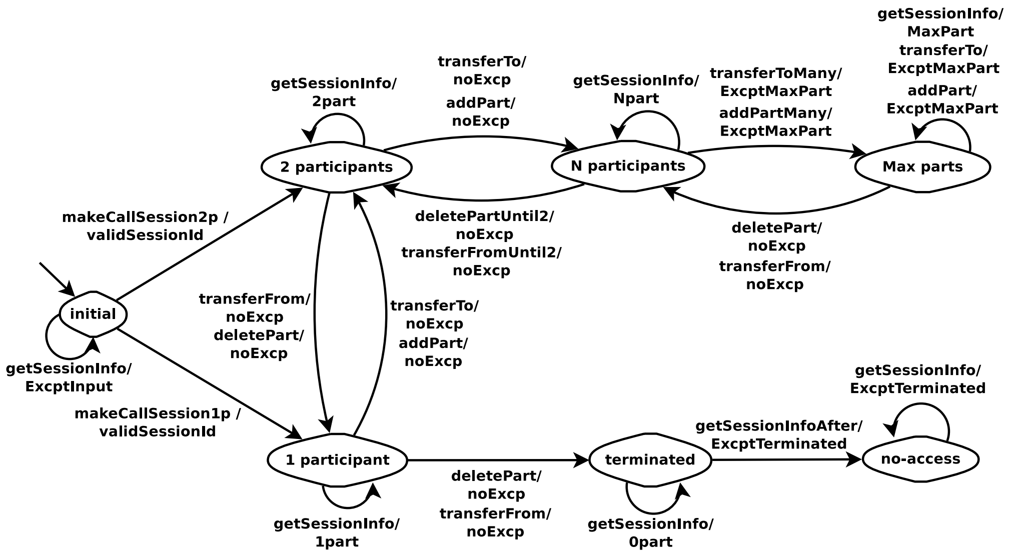


Figure 4.3: FSM for the *number of participants* test scenario (*SC-1-1*) – extracted from (Endo and Simao, 2011).

QualiPSo-Factory `QualiPSo-Factory` is a collaborative environment to support the development of open source software (QualiPSo, 2010). It was developed as a service-oriented application by different partners from industry and academia in the context of the *Quality Platform for Open Source Software* (QualiPSo) project[1].

`QualiPSo-Factory` is composed of services which include functionalities concerning project management, issue tracking, version control, coverage testing, calendar, VOIP, and so on. The services are integrated with the core module that provides essential functionalities, such as security, notification, and semantics. The core implements two types of services, internal and external. The internal services are invoked only by other trusted services deployed inside the factory. The external services are visible for the Factory users and can be accessed by them as Web services. We applied the proposed process to test external services provided by the core module (QualiPSo-Factory 0.6). In this context, four test scenarios were selected. Figure 4.4 depicts an FSM model for a test scenario involving access control. To conduct this study, we needed to integrate the JStateModelTest tool with JUnit (JUnit, 2011) and Maven (ASF, 2011).

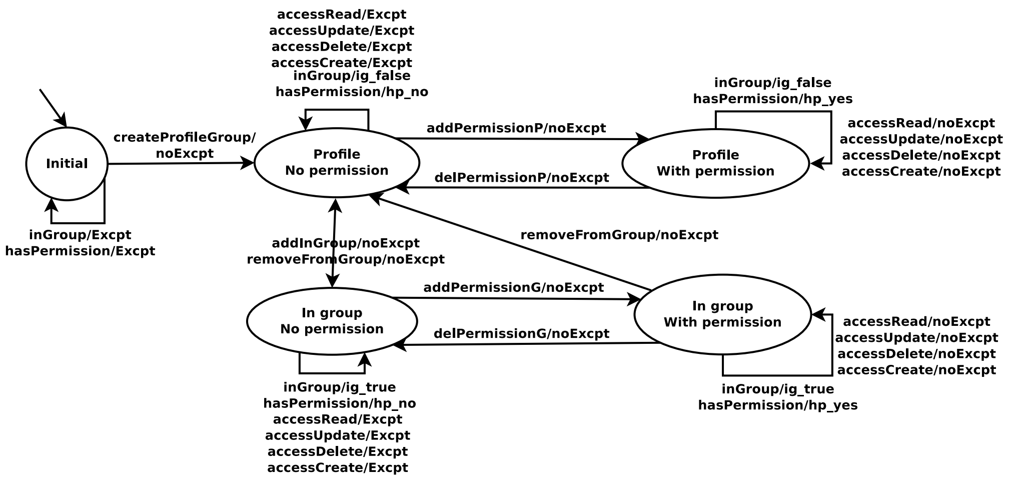


Figure 4.4: FSM for profile and group access control (*SC-2-3*) – adapted from (Endo and Simao, 2011).

Analysis of Results Table 4.1 summarizes the data collected for the two applications. We selected two test scenarios for `ThirdPartyCall-SOA` (*SC-1-1, SC-1-2*) and four for `QualiPSo-Factory` (*SC-2-1, ..., SC-2-4*). For each test scenario, Table 4.1 shows data about the FSM test model, the concrete adaptor, and the test suite.

The FSM models have from four to seven states, loops and an infinite number of possible paths. FSMs were partially specified and, as seen in Figures 4.3 and 4.4, presented a manageable size to be manipulated by graphical tools. We measured the effort necessary to implement the concrete adaptors using LoC. The total LoC for each adaptor depends on the test model size, more specifically the number of inputs and outputs. Some developed code was automatically generated and reused. Moreover, the annotation-based structure of the abstract adaptor (produced by the `adaptor-generator` module) fosters code of low complexity ($\approx$ 2.2 CC). It produced an aver-

[1]http://www.qualipso.org – last accessed on 20/02/2013.

Table 4.1: Data about test scenarios of `ThirdPartyCall-SOA` and `QualiPSo-Factory` – adapted from (Endo and Simao, 2011).

	ThirdPartyCall-SOA		QualiPSo-Factory			
Measure	**SC-1-1**	**SC-1-2**	**SC-2-1**	**SC-2-2**	**SC-2-3**	**SC-2-4**
FSM Test Model						
# states	7	5	4	5	5	5
# transitions	26	20	13	28	33	22
# input symbols	12	10	8	10	13	9
# output symbols	10	7	3	6	6	3
Concrete Adaptor						
# LoC	487	334	206	216	368	199
# methods	31	22	13	18	23	14
average # LoC for input symbol	11.9	9.5	13.8	8.7	9.6	10.8
average # LoC for output symbol	14.4	15.3	1.6	2.3	7.3	1.6
average Cyclomatic Complexity (CC)	2.4	2.2	2.2	1.8	2.3	2.0
Test Suite						
# test cases	41	31	12	28	33	30
average test length	4.4	3.6	4.6	4.0	4.6	5.3

age of ≈10.7 LoC and ≈7.1 LoC to implement each input symbol and each output symbol, respectively. Data about the concrete adaptors were collected using Eclipse Metrics (SourceForge.net, 2005). `test-generator` and `runner` were essential to automate the generation and execution of test suites. `mediator` was used to observe and verify messages exchanged during the tests. The test suites have an average length of 4.4 input symbols per test case. The number of test cases varies according to the model complexity (number of states, transitions, inputs, and outputs).

Although the two applications have different sizes and domains, the data considering isolated test scenarios are similar. The key difference is the number of test scenarios that will increase according to the application complexity and size. We identify four main results of the case study. First, the selection of test scenarios restricts the test model size, keeping the model manageable and meaningful. Moreover, it shows that FSMs can be applied to test service-oriented applications. Second, a reasonable effort for developing adaptors can be reached by using adequate structures and source code generation. Third, test cases are automatically generated and more tests can be included by adopting alternative test methods. Fourth, the JStateModelTest tool was helpful to support the testing process usage.

4.5 Final Remarks

This chapter has presented an MBT process for service-oriented applications that uses state models. The process has been described discussing improvements to overcome issues on testing service-oriented applications. Finally, we have presented an exploratory study of the process, evaluating its feasibility and aspects of implementation.

The contributions of this chapter are twofold: *(i)* an MBT process for service-oriented applications that adds additional steps, tools, and artifacts, to tackle specific characteristics of this software class; and *(ii)* an exploratory evaluation of the process, considering its automation (tools) and practical usage (case study). Our proposal advances toward providing a formal, flexible and automated process to test service-oriented applications. We do not differ between single services and service composition during the process description, being applied for the both contexts. Thus, the tester does not need to change the strategy when testing different levels of integration. We have achieved some practical results in the case study, presenting reasonable development effort and complexity. Regarding the tools, the development of prototypes has provided evidences that a high level of automation can be reached.

In this study, we have used the modeling technique (FSMs) and a related generation method (P) to support modeling and test generation during the process instantiation. No further extensions were proposed to the modeling technique and the test generation. It characterizes a limitation since specific features of SOAs and Web services are not considered in these steps. The use of FSMs is quite intuitive with single services since it is direct for testers to relate requests and responses with inputs and outputs, respectively. On the other hand, abstracting states and obtaining a reduced model remain a challenge for testers. Although state models have been applied in service compositions as shown in Section 2.5.1, their application has some limitations and event-driven techniques like ESGs seem promising to support MBT of composite services. In the next chapter, we elaborate a novel approach using the gained experience in Chapter 3 and being inspired by steps, artifacts, and tools described in this chapter.

CHAPTER

5

Holistic Testing of Service Compositions

5.1 Overview

To assure the delivery of high quality and robust service-oriented applications, service testing has received much attention (as shown in Section 2.5). In Chapter 4, we have proposed a process that identifies steps, tools, and artifacts to support the MBT of service-oriented applications. Although the exploratory study gives evidences of the process' practical use, specific mechanisms are still necessary when testing service compositions. First, the behavior of the composite services is highly dependent on the partner services. In service compositions, there exist complex communications among the integrated services in which missing or unexpected messages can lead to a failure. Furthermore, the composition may also fail due to an undesirable behavior of partner services, such as unavailable servers, corrupted messages, and long response time. Testing those cases systematically is a challenging task (Ilieva et al., 2011).

An ESG-based approach is promising due to several reasons. First, message exchanges in a service composition can be viewed as an ordered sequence of events. Second, ESG modeling can be learned in a short period, demands little manual work, and has supporting tools (Belli et al., 2006). Furthermore, artifacts (e.g., standardized service descriptions in WS-BPEL or WS-CDL) are not necessary to create an ESG or any other model for a service composition. That is, an ESG can be constructed in an ad-hoc way by the tester wherever no model is available.

This chapter introduces a holistic approach, called ESG for Web Service Compositions (ESG4WSC), we propose to generate cost-effective test cases for composite services. It is assumed that the tester can observe and modify the exchanged messages using an ESB, i.e., the service composition is considered as a black box, but the tester has control over messages exchanged by the partner services. The novelties and merits of this chapter are summarized as follows:

- An event-based approach is proposed to support service composition testing by: *(i)* extending the basic notions of ESG (Belli et al., 2006), referred to as ESG4WSC, for testing the service composition behavior under regular circumstances (positive testing) and undesirable situations (negative testing); and *(ii)* introducing algorithms to generate positive and negative test cases from an ESG4WSC model.

- Two tools are introduced to support automation: *(i)* Test Suite Designer (TSD) provides a graphical user interface which allows to model the SUT and to generate test cases; and *(ii)* Event Runner for Test Execution (ERunTE) automates test execution by composing three modules: a Web service, a test runner, and an ESB component.

This chapter is a summary of paper *"A Holistic Approach to Model-based Testing of Web Service Compositions", Belli, F., Endo, A. T., Linschulte, M. and Simao, A.,* published in the Software: Practice and Experience journal, DOI: 10.1002/spe.2161 (Belli et al., 2013); the authors' names are alphabetically ordered in this paper. The algorithm and its definitions concerning the event tree, and the model metrics are proposed in this dissertation.

5.2 Introducing the ESG4WSC Approach

To facilitate the description and understanding of the approach, a "running example" is introduced first. Then, the ESG4WSC model is described in detail.

5.2.1 Running Example

The business process to grant loans called `xLoan`, proposed in (Bentakouk et al., 2009), is the running example used to illustrate the approach. The example involves three services: *LoanService* (LS), *BankService* (BS), and *BlackListInformationService* (BLIS). *LoanService* represents the own business process `xLoan` whose workflow is implemented using WS-BPEL. It contains three operations: request, cancel, and select. *BankService* represents the financial agency that approves (or not) loans, providing offers to its clients when a loan is approved. The operations used in the example are approve, offer, confirm and cancel. *BlackListInformationService* provides an operation checkBL to check if a client has debits with some financial organization.

The example presented in (Bentakouk et al., 2009) is extended to add parallel flow (a common entity of service compositions) in the process by including a new service called *CommercialAssociationService* (CAS). Similar to *BlackListInformationService*, CAS provides operations to check whether a client has debits with some commercial organization. In the extension, both services are supposed to be called in parallel. If the client has debit according to one of them, the client needs the bank approval.

5.2.2 The ESG4WSC Model

This section introduces an event driven modeling technique, named ESG4WSC, that represents the request and response messages exchanged between services involved in a service composition. When a given event is refined by input parameters that determine the next events, decision tables are associated to augment the representation. A Decision Table (DT) logically links constraints ("if") with events ("then") that are to be triggered, depending on combinations of constraints ("rules"). DTs are powerful mechanisms for handling sequences of events which depend on constraints, and refining data modeling of calls to invoked services (Belli and Linschulte, 2010). DTs are formally defined as follows.

Definition 1. A (simple/binary) *decision table* $DT = \{C, E, R\}$ represents events that depend on certain constraints, where:

- C is the nonempty finite set of constraints (conditions), that can be evaluated as either *true* or *false*,
- E is the nonempty finite set of events, and
- R is the nonempty finite set of rules each of which forms a Boolean expression connecting the truth/false configurations of constraints and determines the executable or awaited event.

Definition 2. Let R be a set of rules as in Definition 1. Then a *rule* $r_i \in R$ is defined as $r_i = (C_{True}, C_{False}, E_x)$ where:

- $C_{True}, C_{False} \subseteq C$ are the disjoint sets of constraints that have to be evaluated as *true* and as *false*, respectively,
- $E_x \subseteq E$ is the set of events that should be executable if all constraints $c_t \in C_{True}$ are resolved to true and all constraints $c_f \in C_{False}$ are resolved to false. In this work, $|E_x| = 1$ for all rules to avoid nondeterminism.

Note that, under regular circumstances, C_{True} and C_{False} partition C, i.e., $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$. In certain cases, it is inevitable to have constraints with a *don't care* (noted as '-' in a DT). In this case, such a constraint is not considered in a rule and is neither in C_{True} nor in C_{False}.

Table 5.1 presents a DT that models the invoking process of operation checkBL of BLIS. It contains two constraints on input parameter uniqueID, three successor events (inBList, notInBList, SOAPFault), and three rules (R1, R2, R3). Rules are used to determine the successor event of operation checkBL. For instance, R1 means that if uniqueID is valid and also in the blacklist (i.e., both constraints are true), then the next event (namely, event inBList standing for uniqueID being in the blacklist) should be completed. In R3, if uniqueID is not valid, then the other constraint does not matter (namely, '-') and the next event is SOAPFault.

Table 5.1: A DT for operation checkBL of BLIS – adapted from (Belli et al., 2013).

		Rules		
	checkBL(uniqueID)	R1	R2	R3
Constraints	uniqueID is valid	T	T	F
	uniqueID in Blacklist	T	F	-
Events	inBList	✓		
	notinBList		✓	
	SOAPFault - invalid identification			✓

DTs are useful to describe constraints, but they are not appropriate for describing service composition interactions. Hence, the ESG notion is extended and combined with DTs in order to consider additional aspects, such as communication, parallel flow and conditional activities.

Definition 3. An *event sequence graph for web service compositions* $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ is a directed graph, where

- V is a nonempty finite set of vertices (representing events),
- $E \subseteq V \times V$ is a finite set of arcs (edges),
- M is a finite set of refining ESG4WSC models,
- $R \subseteq V \times M$ is a relation that specifies which ESG4WSCs are connected to a refined vertex,
- DT is a set of DTs that refine events according to function f,
- $f : V \to DT \cup \{\varepsilon\}$ is a function that maps a vertex $v \in V$ to a decision table $dt \in DT$. If $v \in V$ is not associated with a DT, then $f(v) = \varepsilon$,
- $\Xi \subseteq V$ is a finite set of distinguished vertices with $\xi \in \Xi$ called entry nodes, wherein for each $v \in V \setminus \Xi$ there exists at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from $\xi \in \Xi$ to $v_k = v$, for $i = 0, \ldots, k-1$ and $(\xi, v_0) \in E$, and
- $\Gamma \subseteq V$ is a finite set of distinguished vertices with $\gamma \in \Gamma$ called exit nodes, wherein for each $v \in V \setminus \Gamma$ there exists at least one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $(v_k, \gamma) \in E$.

Definitions 4 and 5 elaborate Definition 3, formalizing the set of vertices and the set of DTs, respectively.

Definition 4. Let V be as in Definition 3. Then, the set of vertices V is partitioned into V_e, $V_{refined}$, V_{req} and V_{resp}, i.e., $V = V_e \cup V_{refined} \cup V_{req} \cup V_{resp}$ and V_e, $V_{refined}$, V_{req} and V_{resp} are pairwise disjoint, where

- V_e is a set of generic events,
- $V_{refined} = \{v \in V \mid \exists m \in M \wedge (v, m) \in R\}$ is a set of vertices refined by one or more ESG4WSCs. A refinement with more than one ESG4WSC represents behavior running in parallel,
- V_{req} is a set of vertices modeling a request to its own interface/operations (public) or an invoked service (private), and
- V_{resp} is a set of responses to a public or private request. Therefore, it is also remarked as public or private.

Definition 5. Let DT be defined as in Definition 3. Then, the set of decision tables DT is partitioned into DT_{seq} and DT_{input}, where

- DT_{seq} is the set of decision tables that model the execution restrictions for following events, and
- DT_{input} is the set of decision tables that model constraints for input parameter of invoked operations.

Since service compositions always initiate with one or more request events, set Ξ contains only vertices $v \in V_{req}$. To mark the entry and exit of an ESG4WSC, all $\xi \in \Xi$ are preceded by a pseudo vertex $[\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex $] \notin V$. For two events $v, v' \in V$, event v' can follow the execution of v if and only if $(v, v') \in E$. In this case v' is also called *successor* of v and v is called *predecessor* of v'. If vertex $v \in V$ has more than one successor, then v can be refined by a DT.

The semantics of an ESG4WSC model is as follows: Any $v \in V$ represents an event, e.g., a request or a response which occurs during the invocation of another service. Request and response events can be divided into *public* and *private*. A public request is controlled by the tester, i.e., it is an operation call to the composition itself which is supposed to be done by a consumer or a tester. A public response is expected to be an answer of the composition to a public request and therefore should be observable by the consumer/tester. Private requests and responses, which represent events of the partner services, are usually not observable by a consumer; however, it is assumed that they are observable by the tester. Private requests are to be observed by the tester and the tester should control and (if necessary) send back the appropriate response.

Example 6. Figure 5.1 shows an ESG4WSC model for `xLoan`. Requests are represented by gray vertices in circle shapes; responses are represented by gray vertices in ellipse shapes. Vertices with a bold line represent public requests and responses. Vertices refined by DTs are double-circled. Event check (dashed box) is a refined event with two refining ESG4WSCs which represent operations checkBL (for service BLIS) and inDebtorsList (for service CAS) that are to be executed concurrently. The corresponding ESG4WSC using the defined sets and functions looks like this:

$ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ with

- $V_e = \{$Timeout>2h$\}$,
- $V_{refined} = \{$check$\}$,
- $V_{req} = \{$*LS:requestLoan*, *BS:approveBank*, *BS:offer*, *LS:cancel*, *LS:SelectOffer*, *BS:cancelBank*, *BS:confirmBank*$\}$
- $V_{resp} = \{$*BS:approved*, *BS:notApproved*, *LS:notApprovedMSG*, *BS:Offers*, *LS:replyOffers*, *LS:wrongOffer*, *LS:replySelect*$\}$
- $E = \{($*LS:requestLoan*, *BS:approveBank*$)$, $($*LS:requestLoan*, check$), \ldots\}$,
- $M = \{M_{BLIS}, M_{CAS}\}$,
- $R = \{($check$, M_{BLIS}), ($check$, M_{CAS})\}$,
- $DT = DT_{seq} \cup DT_{input} = \{dt_{check}\} \cup \{dt_{LS:requestLoan}, dt_{LS:cancel}, dt_{LS:SelectOffer}\}$,
- $f($check$) = dt_{check}$,
  $f($*LS:requestLoan*$) = dt_{LS:requestLoan}$,
  $f($ *LS:cancel*$) = dt_{LS:cancel}$,
  and $f($ *LS:SelectOffer*$) = dt_{LS:SelectOffer}$.
- $\Xi = \{$*LS:requestLoan*$\}$,
- $\Gamma = \{$*LS:notApprovedMSG*, *BS:cancelBank*, *LS:replySelect*$\}$

The ESG4WSC model can be seen as a simplified representation of possible and expected executions of a service composition. It aggregates sequences of events which describe walks of execution and message exchanges along the execution.

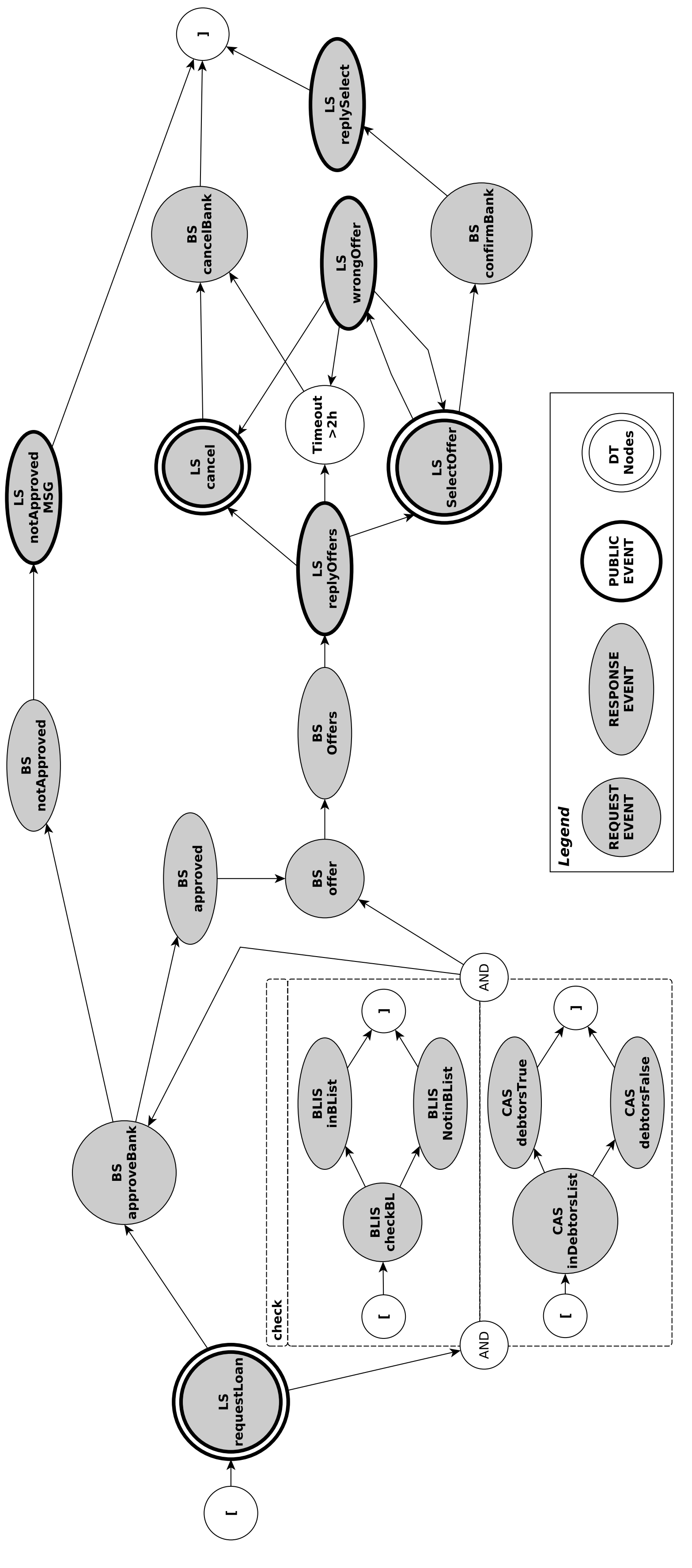


Figure 5.1: ESG4WSC for the `xLoan` example – adapted from (Belli et al., 2013).

Definition 7. Let V and E be defined as in Definition 3. Then any sequence of vertices $\langle v_0, \ldots, v_k \rangle$ is called an *event sequence* (*ES*) if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$. The length l of an ES $\langle v_0, \ldots, v_k \rangle$ is defined as the number of vertices $|\langle v_0, \ldots, v_k \rangle|$, that is, $l(ES) = |\langle v_0, \ldots, v_k \rangle| = k+1$. An $ES = \langle v_i, v_k \rangle$ of length 2 is called an *event pair* (*EP*). Furthermore, an ES is partial (or, it is called a *partial event sequence, PES*), if $v_0 \in \Xi$. An ES is complete (or, it is called a *complete event sequence, CES*), if $v_0 \in \Xi$ and $v_k \in \Gamma$.

Definition 8. Two events or ESs a and b that are to be executed in parallel are denoted as $a||b$. The operator $||$ is commutative, i.e., $a||b = b||a$, and associative, i.e., $(a||b)\,||c = a||\,(b||c)$.

Example 9. For the `xLoan` example given in Figure 5.1, services CAS and BLIS are to be executed in parallel, e.g., following sequence might hold:

```
⟨BLIS:checkBL, BLIS:inBList⟩ || ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩
```

Based on Definitions 7 and 8, sequences of events can be derived from the ESG4WSC and may represent parallel execution. A PES represents a sequence of events that starts with an entry node; a CES represents an event sequence that starts with an entry node and ends with an exit node.

5.3 Positive Testing

This section introduces the underlying fault model and explains how test cases are generated for positive testing a composite service.

5.3.1 Fault Model

A CES (Definition 7) describes a specific execution of a service composition that has to be enforced during testing. Thus, it is expected that exactly those events in the specified order are executed. According to this, the following faults might occur during the execution:

- there are calls to services which are *not defined* in the CES,
- there are *missing* calls to services which are defined in the CES,
- the *sequence* of calls is different from the sequence given by the CES, and
- the *parameter* of calls to the invoked services do not correspond to the expected ones.

In order to cause and control a specific CES of the composite service, we need to take control of partner services' messages, since they communicate with the SUT and the flow of the composition might depend on a returned response. The modeled constraints of DTs enable to validate the data passed to the service operations. If the passed data values do not fit to the constraints, we have an irregular behavior.

5.3.2 Test Case Generation

To detect faults in the service composition (due to the fault model described in previous section), a test suite that covers all EPs is generated. Covering EPs is related to criterion all-edges as in structural testing (Section 2.3.1) and transition cover as in FSMs (Section 3.2). If possible, the cost should be minimized when generating CESs to cover all EPs. The problem of generating minimal CESs is related to the Chinese Postman Problem (CPP), as in (Belli and Budnik, 2004). The algorithm for deriving CESs from an ESG4WSC model is described in following steps:

1. Generate CESs for the refined vertices first (recursive call)
2. Add multiple edges (representing EPs) to the ESG4WSC
   (a) If a refined vertex has a DT restricting the ongoing execution
      i. identify the valid successor for each CES with respect to the DT
      ii. add an edge from the refined vertex to the allowed successor
   (b) If a refined vertex has not a DT
      i. add an edge from the refined vertex to the successor (there should be only one) for each CES
3. Generate CESs according to the CPP algorithm (i.e., cover all EPs by CESs of minimal total length)
4. Replace refined vertices in the resulting CES set of Step 3 with the CESs derived in Step 1 with respect to their allowed successors

Note that Step 2 adds multiple edges to the underlying ESG4WSC, that is, every edge represents a CES of the refined vertex and its valid successor. The benefit of this approach is that the resulting CES set derived in Step 3 contains the refined vertex and its corresponding successor as much as needed so that the CESs of Step 1 can be combined completely with the CESs of Step 3 (recall that every EP/edge is to be covered in Step 3). After generating CESs, a DT for the initial composition call is to be defined and evaluated. Therefore, constraints that are associated to the initial input data are selected and added to the initial request event. If a constraint set cannot be fulfilled, CESs can be deleted, e.g., when two contradicting constraints are to be satisfied. A formal description of the CES generation process can be found in Appendix A, Algorithm 1. Example 10 shows the test generation process for the running example.

Example 10. According to Figure 5.1, the test generation algorithm looks as follows:

Step 1 - *generate CESs for refined vertices*: the CPP algorithm is applied to the two refining ESG4WSCs in event check (Figure 5.1). The event sequences of each ESG4WSC are combined with operator $||$. The following sequences for refined vertex check have been generated:

```
S1:  ⟨⟨BLIS:checkBL, BLIS:inBList⟩ || ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩
S2:  ⟨⟨BLIS:checkBL, BLIS:inBList⟩ || ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩
S3:  ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩
S4:  ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩
```

Step 2 - *add edges*: event check has a DT that restricts the execution of next events BS:offer and BS:approveBank, in Table 5.2. To cover all rules in this table, event pair (check, *BS:approveBank*) needs to be covered three times (R1, R2, R3) and event pair (check, *BS:offer*) once (R4). Thus, the algorithm adds the following edges according to Table 5.2:

- 3 edges (check, *BS:approveBank*) for sequences S1 to S3
- 1 edge (check, *BS:offer*) for sequence S4

An intermediate ESG4WSC is produced with extra edges added, as illustrated in Figure 5.2.

Table 5.2: DT for vertex check of Figure 5.1 – extracted from (Belli et al., 2013).

dt$_{check}$	**R1**	**R2**	**R3**	**R4**
event: BLIS:inBList happens	T	T	F	F
event: BLIS:NotInBList happens	F	F	T	T
event: CAS:DebtorsTrue happens	T	F	T	F
event: CAS:DebtorsFalse happens	F	T	F	T
BS:offer				✓
BS:approveBank	✓	✓	✓	

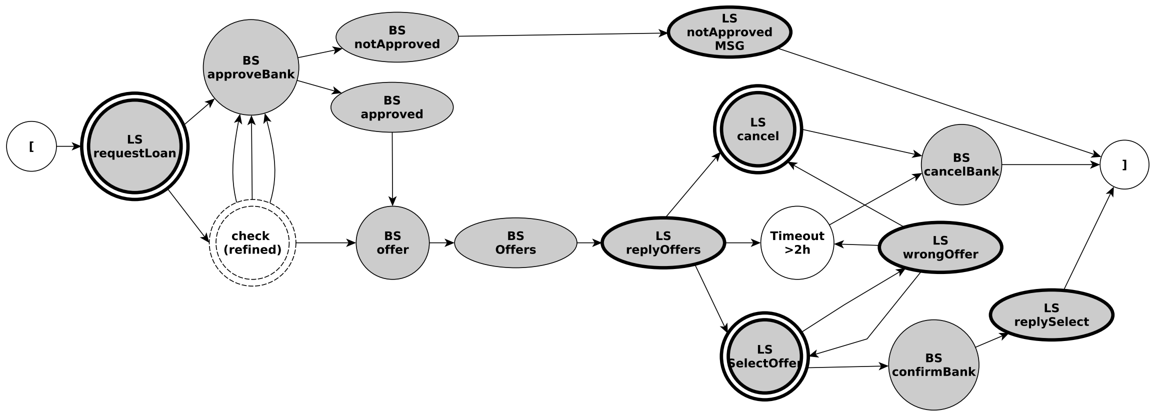


Figure 5.2: ESG4WSC for the `xLoan` example extended by additional edges – adapted from (Belli et al., 2013).

Step 3 - *generate CESs:* The CPP algorithm is applied on the intermediate ESG4WSC (produced in Step 2) to generate CESs. In this step, the refined events (e.g., `check`) are considered as simple vertices (Figure 5.2). The following CESs have been generated (refined vertices and their successor are emphasized with a bold font):

```
CES1:  ⟨LS:requestLoan, BS:approveBank, BS:Notapproved, LS:notApprovedMSG⟩
```

CES2: ⟨LS:requestLoan, **check**, **BS:approveBank**, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:cancel, BS:cancelBank⟩

CES3: ⟨LS:requestLoan, **check**, **BS:approveBank**, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:cancel, BS:cancelBank⟩

CES4: ⟨LS:requestLoan, **check**, **BS:approveBank**, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:SelectOffer, LS:wrongOffer, Timeout > 2h, BS:cancelBank⟩

CES5: ⟨LS:requestLoan, **check**, **BS:offer**, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect⟩

CES6: ⟨LS:requestLoan, **check**, **BS:offer**, BS:Offers, LS:replyOffers, Timeout > 2h, BS:cancelBank⟩

Step 4 - *replace refined vertices by sequences of Step 1:* Refined events are searched in the CES set generated in Step 3 and are replaced by the corresponding CESs derived from the refining ESG4WSCs in Step 1. In the final test suite, event pair (check, *BS:approveBank*) is covered exactly three times in CES2, CES3, CES4 using S1, S2, S3, respectively, to replace `check`. Event pair (check, *BS:offer*) is covered twice in CES5 and CES6, S4 is used in both CESs to replace `check`.

CES1: ⟨LS:requestLoan, BS:approveBank, BS:Notapproved, LS:notApprovedMSG⟩

CES2: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL**, **BLIS:inBList**⟩ **||** ⟨**CAS:inDebtorsList,CAS:debtorsTrue**⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:cancel, BS:cancelBank⟩

CES3: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL**, **BLIS:inBList**⟩ **||** ⟨**CAS:inDebtorsList,CAS:debtorsFalse**⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:cancel, BS:cancelBank⟩

CES4: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL,BLIS:NotinBList**⟩ **||** ⟨**CAS:inDebtorsList,CAS:debtorsTrue**⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:SelectOffer, LS:wrongOffer, Timeout > 2h, BS:cancelBank⟩

CES5: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL,BLIS:NotinBList**⟩ **||** ⟨**CAS:inDebtorsList,CAS:debtorsFalse**⟩⟩, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect⟩

CES6: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL,BLIS:NotinBList**⟩ **||** ⟨**CAS:inDebtorsList,CAS:debtorsFalse**⟩⟩, BS:offer, BS:Offers, LS:replyOffers, Timeout > 2h, BS:cancelBank⟩

The generation of data based on the initial DT is related to the *constraint satisfaction problem* (*CSP*). A CSP is defined by a set of variables X_1, X_2,...,X_n and a set of constraints, C_1,C_2,...,C_m.

Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset (see (Russell and Norvig, 2003)). Each rule of the DT under consideration represents a CSP.

The described algorithm is used to generate a test suite that covers all edges, namely EPs. In other words, the test suite covers all ESs with length 2. Similar processes can be performed to cover ESs with higher length k. For this purpose, it is necessary to transform the ESG4WSC model after Step 1 (see (Belli et al., 2011b) for further details on the transformation). The coverage of this graph will deliver the desired test suite.

5.3.3 Generating CESs using Event Tree's

In this section, we propose an algorithm to generate CESs that cover all EPs. It is an alternative option to the CPP algorithm. As a consequence, this algorithm does not produce a minimum-cost solution for test execution. This algorithm is based on a tree structure called *event tree* which is built using the ESG4WSC model.

Definition 11. Let $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ be an ESG4WSC, as in Definition 3. Then $T = (V_T, E_T, root)$ is an *event tree* of $ESG4WSC$ iff:

- T is a tree,
- V_T is a set of tree nodes that are pairs $\langle v, i\rangle$ such that $v \in V$ and $i \in \mathbb{Z}^+$,
- $E_T \subseteq V_T \times V_T$ is a set of edges $(\langle v, j\rangle, \langle u, k\rangle) \in E_T$ connecting parent node $\langle v, j\rangle$ to child node $\langle u, k\rangle$, such that $(v, u) \in E$ and $j, k \in \mathbb{Z}^+$, and
- $root$ is the pseudo-vertex $[$ and its child nodes are composed by entry vertices in the $ESG4WSC$. In other words, given a node $\langle v, i\rangle$ that is child of $root$, then $v \in \Xi$.

Tree nodes are defined as pairs since an event of the ESG4WSC model can appear zero, one or more times in the tree. The positive integer is used to enumerate and identify the occurrences of a given event, as an *index*.

Definition 12. An event tree T of $ESG4WSC$ is *complete* (or, it is called a *complete event tree*) if all its leaf nodes contain exit nodes.

As the steps to deal with DTs and refined events are equal to the algorithm described in Section 5.3.2, we focus on describing the steps to generate CESs out of an ESG4WSC model (Step 3). The algorithm is divided into the three following steps:

1. *Building the event tree*: in this step, vertex $[$ is labeled as the tree root. For all nodes, the event is expanded and its successors (in the ESG4WSC) are added in the tree as child nodes.

This process is repeated until all events are expanded exactly once following a breadth-first search. Thus, it guarantees that all edges (EPs) are covered once in the tree. This procedure is similar to build a testing tree from an FSM, more details in (Chow, 1978). For the ESG4WSC in Figure 5.2, the event tree built in this step is shown in Figure 5.3 (part above the black horizontal lines). The index of each tree node is represented by the number within a small box near the event vertex.

2. *Completing the tree*: for all leaf nodes in tree that do not contain an exit node, append the shortest path (event sequence) to an exit node. This process is represented in Figure 5.3 by bold dashed edges (part below the black horizontal lines). This step ensures that the event tree is *complete*.

3. *Deriving CESs*: CESs are generated using the *complete event tree* built in the previous steps (as exemplified in Figure 5.3). CESs are derived by a breadth-first search that traverses the entire tree, i.e., each path from the root to a leaf node is a CES. After this process, the set of CESs covers all EPs in the ESG4WSC model.

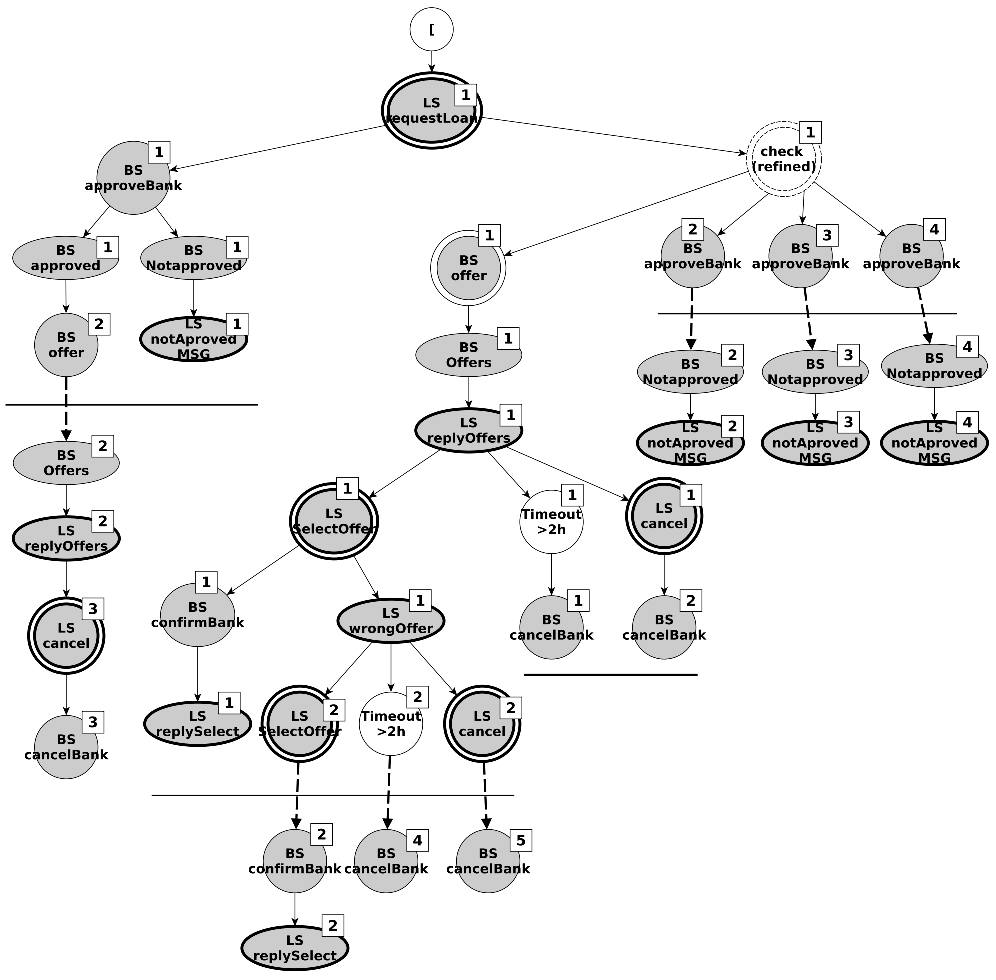


Figure 5.3: Complete event tree for the model in Figure 5.2.

Using this algorithm, 11 CESs are generated, while the CPP-based algorithm produces six CESs, as shown in Section 5.3.2. As a consequence, more events need to be executed to cover all EPs. A detailed analysis of the trade-offs between these two algorithms is presented in Chapter 6, Section 6.3.

5.4 Negative Testing

Section 5.3 described the testing process for expected/desired behaviors. However, it is also important to test undesired situations where partner services do not work as expected. Thus, a holistic approach is worthwhile that generates positive (desired) and negative (undesired) tests.

The negative testing checks separately unexpected behavior in public events and private events. These two situations are represented by *public faulty event sequences* (*PubFESs*) and *private faulty event sequences* (*PriFESs*). Sections 5.4.1 and 5.4.2 present the definitions and algorithms to generate PubFESs and PriFESs from an ESG4WSC model.

5.4.1 Negative Testing of Public Events

The negative testing for public events involves generating sequences that cover unspecified event pairs for public events, i.e., request and response messages of the service composition interface. This part considers that a composition can be viewed and tested as a single service. First, the ESG4WSC is used to derive an ESG4WS[1] representing only public events of the service composition interface. Second, faulty pairs are derived from this ESG4WS using the algorithm proposed in (Belli and Linschulte, 2010). Finally, each faulty pair is covered by an event sequence that contains a faulty edge at the end (a negative test case). Given an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$, let $\digamma$ be a special event used to represent any *faulty event*, such that $\digamma \notin V$.

Definition 13. The ordered pair $(pes; \digamma)$, such that pes is a PES $\langle v_0, \ldots, v_k, v_{k+1} \rangle$ (Definition 7), is a *public faulty event sequence* (*PubFES*), if the last two events of pes are public and there is no edge connecting both, i.e., v_k and v_{k+1} are public events and $(v_k, v_{k+1}) \notin V$. In a PubFES, pes is expected to produce a faulty event $\digamma$.

Example 14. For the `xLoan` example given in Figure 5.1, the following pair is a PubFES:

(⟨`LS:requestLoan, BS:approveBank, BS:approved, BS:offer, BS:offers, LS:replyOffers, LS:cancel, LS:SelectOffer`⟩; $\digamma$)

As there is no edge between the last two events `LS:cancel` and `LS:SelectOffer`, they were selected as a faulty pair. For this undesired case, it is expected that the composition produces a faulty event $\digamma$. If no faulty event is produced, this test sequence fails; otherwise it passes. The algorithm for deriving PubFESs from an ESG4WSC is as follows:

[1]ESG4WS is a modeling technique defined in (Belli and Linschulte, 2010) that represent the behavior of singles services.

1. Transformation from ESG4WSC to ESG4WS: an ESG4WS is a model that contains only the (public) request and response events for a single service. It can be obtained from an ESG4WSC by removing the private events and keeping edges between any public events v_i and v_j when there exists an ES from v_i to v_j. Thus, the obtained ESG4WS will contain only events related to the composition itself and there is no private event. A formal description of this transformation is shown in Appendix A, Algorithm 3.

2. Inclusion of faulty edges: in the produced ESG4WS, faulty edges are added between pairs of events with no edges. The formal description of this step is also shown in Appendix A, Algorithm 3.

3. Generation of test sequences: for each faulty edge (v_i, v_j) in the ESG4WS, find a PES pes that leads to v_i in the ESG4WSC. The algorithm to find a PES is implemented by a breadth-first search, from the start event $[$ to v_i, that considers the refining ESG4WSCs involved. A formal description of this procedure is shown in Appendix A, Algorithm 2. Then, append v_j to pes referred to as $pes \oplus v_j$. Finally, create the PubFES $(pes \oplus v_j; F)$.

The formal description of this algorithm for generating PubFESs can be found in Appendix A, Algorithm 4.

Example 15. Using `xLoan`, the ESG4WS in Figure 5.4 is obtained after the transformation. The faulty edges are represented by gray dashed lines. The faulty edges are created by:

- connecting all response events with request events,
- connecting the start event $[$ with all request events, and
- connecting request events with request events,

in case that there is no edge connecting them. The self-loop from `LS:cancel` to `LS:cancel` was also considered since it represents a one-way operation.

After obtaining the ESG4WS, test cases have to be generated to cover the all faulty edges. Then, for each faulty edge (v_i, v_j), a PES from the ESG4WSC (Figure 5.1) is derived to reach the event v_i. The event v_j is included after v_i and the faulty event F is added to the tuple. For the faulty edge `(LS:replySelect, LS:cancel)`, the following PubFES is obtained:

(⟨`LS:requestLoan`, ⟨⟨`BLIS:checkBL,BLIS:NotinBList`⟩ `||` ⟨`CAS:inDebtorsList, CAS:debtorsFalse`⟩⟩, `BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank,` **`LS:replySelect, LS:cancel`**⟩; F)

This PubFES tests a scenario in which the client successfully selects an offer (event `LS:replySelect` is a confirmation message) and tries to cancel it afterwards (event `LS:cancel`). This is not allowed by the composition and a fault should be produced. To generate the final test suite of public negative tests, the same procedure is repeated for each faulty edge in the ESG4WS (Figure 5.4).

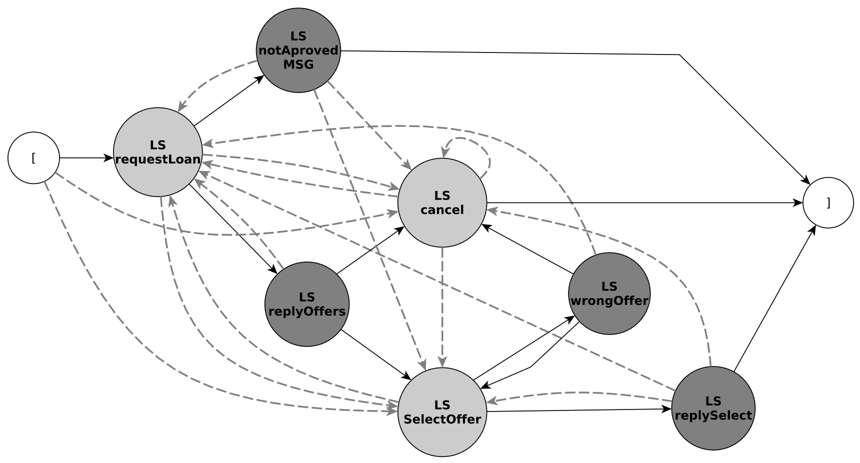


Figure 5.4: ESG4WS for the `xLoan` public interface – adapted from (Belli et al., 2013).

5.4.2 Negative Testing of Private Events

It is often not clearly defined what happens with a composition if an partner service is not working as expected. In this section, we propose an algorithm to generate test cases that cover unexpected behavior of partner services. Event sequences are produced to reach private request and response events and create undesirable situations according to some predefined fault classes. The concept of sensitive events is also proposed in this section to deal with the oracle problem in negative testing of private events.

In this work, seven fault classes are defined based on fault taxonomy and fault injection literature (Chan et al., 2009; Cavalli et al., 2010; Ilieva et al., 2011). The fault classes are the following:

No response: the invoked service does not send back a response for a request-response operation, e.g., due to internal problems or modified behaviors.

Long time response: the invoked service needs inappropriate long time to send a response back.

Missing service: the service is missing, e.g., the server hosting the service is not available or the service address (URL) has changed.

Unexpected fault: it can be an unexpected SOAP-fault returned by the invoked service or a fault produced by the environment, e.g., a fault caused by pre/post-processing in the ESB.

Wrong XML schema: the invoked service sends back a message that is invalid for the known XML Schema, e.g., due to some (untold) changes of the service by the provider.

Wrong XML syntax: the response of the invoked service contains a corrupted XML file, e.g., due to some noise in the network.

Right schema, wrong data: the response is a well-formed message that contains invalid data, e.g., an invalid date.

Testing these undesirable situations is important to the robustness of the composite service under test. Fault classes “no response”, “missing service” and “unexpected fault” can be tested for every private request event of an ESG4WSC model. Fault classes "longtime response", "wrong XML schema", "wrong XML syntax" and "right schema, wrong data" can be tested for every private response event of an ESG4WSC. Table 5.3 summarizes this information and also includes the symbols we use to represent each class in PriFESs.

Table 5.3: Fault classes and their relation to events – extracted from (Belli et al., 2013).

			event of ESG4WSC	
		symbol	request	response
fault class	no response	$\digamma_{\text{NR}}$	✓	
	long time response	$\digamma_{\text{LR}}$		✓
	missing service	$\digamma_{\text{MS}}$	✓	
	unexpected fault	$\digamma_{\text{UF}}$	✓	
	wrong XML schema	$\digamma_{\text{WSc}}$		✓
	wrong XML syntax	$\digamma_{\text{WSy}}$		✓
	right schema, wrong data	$\digamma_{\text{WD}}$		✓

When one of the fault classes is provoked, an automated test oracle needs to be established to verify the expected test outputs. A tester can decide to evaluate and define the expected behavior for every single situation by hand. However, this would mean a lot of manual work and does not scale well. A more straightforward approach we introduce is to mark events of the ESG4WSC model as *sensitive*, i.e., these events are not allowed to show up after provoking one of the faulty situations. In this case, the sensitive events are used as an oracle to automatically evaluate the test cases. If after the execution of a PriFES no sensitive event is observed, the test case passes; otherwise, it fails.

The tester should identify and highlight the sensitive events in the ESG4WSC. From a theoretical point of view, any event can be marked as sensitive. However, assuming that the sensitive event is caused by the composition under test, public response events and private request events are the main candidates. Among them, the tester needs to analyze which of these events are critical according to the domain of the SUT. The tester should check which events cannot be observed in the SUT if some faulty situation occurs.

Definition 16. Given an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$, the nonempty set $S \subset V$ represents the events marked as *sensitive*. The sensitive events are not allowed to ocurr after provoking a faulty situation.

Definition 17. Given that pes is a PES $\langle v_0, \ldots, v_k \rangle$, $\digamma$ is any faulty event, and $s \subseteq S$ is a set of sensitive events, the triple $(pes; \digamma; s)$ is a *private faulty event sequence (PriFES)* if $v_k \in V_{req} \cup V_{resp}$ is a private event and s is not empty.

Example 18. For the `xLoan` example given in Figure 5.1, let `LS:replyOffers` be a sensitive event; the following triple is a PriFES:

($\langle$`LS:requestLoan`, $\langle\langle$`BLIS:checkBL`$\rangle$ `||` $\langle$`CAS:inDebtorsList, CAS:debtorsFalse`$\rangle\rangle$; $\digamma_{\text{MS}}$; {`LS:replyOffers`})

In this example, response event `LS:replyOffers` represents the approved loan and its offers. Thus, event `LS:replyOffers` is selected as sensitive because the loan must not be granted if any unexpected behavior happens in the process. $\mathcal{F}_{\text{MS}}$ represents the fault class "missing service" that must be provoked, i.e., service BLIS is not available. This sequence passes if no sensitive event (`LS:replyOffers`) is produced by the composition after executing the PES and provoking $\mathcal{F}_{\text{MS}}$; otherwise it fails.

The negative testing for private events generates sequences that cause some unexpected behavior in the partner services, i.e., in the private events, and check the service composition in these cases. The algorithm for deriving PriFESs from an ESG4WSC is as follows:

1. Let V_{RR} be the set of all private request and response events in the ESG4WSC model. Then, for each $e \in V_{RR}$, find the shortest PES pes that reaches e (Algorithm in Appendix A, Algorithm 2);
2. If e is a private request event,
   (a) Copy pes and mark e with $\mathcal{F}_{\text{NR}}$ to provoke the "no response" fault.
   (b) Copy pes and mark e with $\mathcal{F}_{\text{MS}}$ to provoke the "missing service" fault.
   (c) Copy pes and mark e with $\mathcal{F}_{\text{UF}}$ to provoke the "unexpected fault" fault.
3. If e is a private response event,
   (a) Copy pes and mark e with $\mathcal{F}_{\text{LR}}$ to provoke the "longtime response" fault.
   (b) Copy pes and mark e with $\mathcal{F}_{\text{WSc}}$ to provoke the "wrong XML schema" fault.
   (c) Copy pes and mark e with $\mathcal{F}_{\text{WSy}}$ to provoke the "wrong XML syntax" fault.
   (d) Copy pes and mark e with $\mathcal{F}_{\text{WD}}$ to provoke the "right schema, wrong data" fault.
4. For all sequences produced in previous steps, add the set of sensitive events s that is not covered by pes to the PriFES, i.e., (pes; $\mathcal{F}$; s).

The formal description of this algorithm can be found in Appendix A, Algorithm 5.

Example 19. Consider private request event `CAS:inDebtorsList` of the `xLoan` example. The first step is to find the shortest PES that reaches `CAS:inDebtorsList`, i.e., pes=⟨`LS:requestLoan`, ⟨⟨`BLIS:checkBL`,`BLIS:NotinBList`⟩ || ⟨**`CAS:inDebtorsList`**⟩⟩⟩.

Notice that `CAS:inDebtorsList` is part of refined event `check`. In this case, CESs must be generated for the other parallel ESG4WSCs so that there is no influence on event `CAS:inDebtorsList` and the provoked fault. Next, pes is copied to pes_1 and event `CAS:inDebtorsList` is marked with $\mathcal{F}_{\text{NR}}$. This PriFES tests the scenario in which the service composition calls operation inDebtorsList and no answer/response is sent back. The same procedure is performed for $\mathcal{F}_{\text{MS}}$ and $\mathcal{F}_{\text{UF}}$, with the copies pes_2 and pes_3, respectively.

Let $s = \{$`LS:replyOffers`$\}$ be the set of sensitive events, the resulting PriFESs are as follows: (pes_1; F_{NR}; s), (pes_2; F_{MS}; s), and (pes_3; F_{UF}; s). As the client reputation must be good in both services, BLIS and CAS, any fault in event `CAS:inDebtorsList` must not produce a successful approval represented by `LS:replyOffers`, marked as sensitive. The same steps are repeated for all other private request and response events.

5.5 Tool Support

For large models, test generation and execution can hardly be done by hand. Therefore, two tools have been developed and used to support test case generation and test execution for the ESG4WSC approach[2]. The tools described in this section were developed in cooperation with researchers from Paderborn Universität, Germany. Particularly, the candidate implemented the event tree algorithm (Section 5.3.3) and the model metrics for the tool described in Section 5.5.1. The tool described in Section 5.5.2 was fully developed by the candidate. It is important to emphasize that the two tools were inspired by the supporting tools of the MBT process defined in Chapter 4, Section 4.3.2.

5.5.1 Test Generation

A tool called Test Suite Designer (TSD) provides a graphical user interface for the tester to model all features of an ESG4WSC that are necessary for test generation. Figure 5.5 shows a screenshot of this tool. The TSD tool implements the algorithms described in Sections 5.3.2, 5.3.3, 5.4.1, and 5.4.2 for generating positive and negative test suites. It also allows generating test data out of DTs. Figure 5.6 shows a screenshot of how DTs are manipulated in TSD.

An XML format was defined to describe test cases generated by TSD. The resulting XML files are used as input to the test execution environment. This integrates the test generation and test execution and reduces the dependency between both environments. Thus, new tests can be generated and no extra effort is necessary for concretization, except for the adaptors.

Figure 5.7 presents an example of a test case in the XML format. The file starts with an element `<TestCase>`, followed by an element `<CompleteEventSequence>` that represents a CES. Events are represented by the element `<Event>` with attributes to label and classify (type and public) the event. Events with associated DT (Line 03) have an attribute to define which rule must be tested and may include child elements `<Param>` to provide input data (Lines 04 and 05). A refined event is represented by the element `<RefinedEvent>` (Line 07) and can include one or more CESs. In the example, there are two CESs in parallel, the first in Lines 08-15 and the second in Lines 16-21. When a private response event is supposed to answer a specific message, the element `<Message>` can be used within the event, like in Line 31. A pre-defined SOAP

[2]The tools and a guide for their use can be found at `http://adt.uni-paderborn.de/en/test-tools.html`

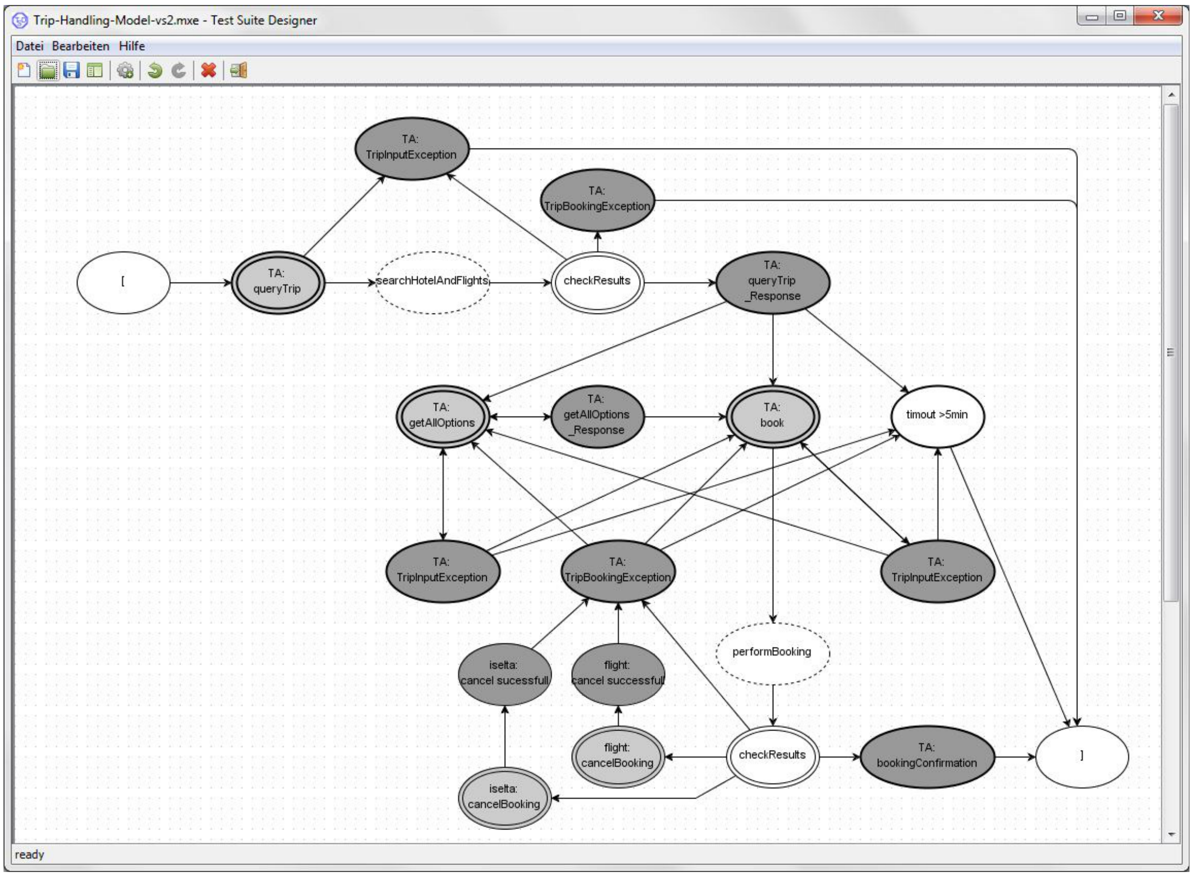
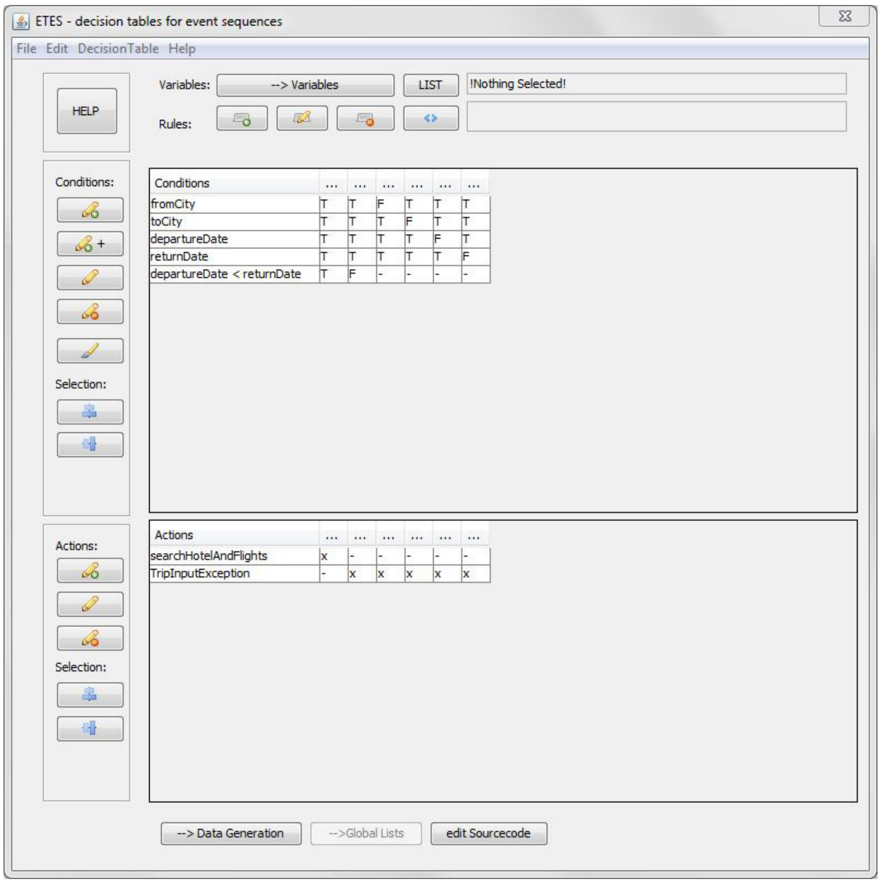


Figure 5.5: An ESG4WSC model in TSD – extracted from (Belli et al., 2013).

message can be provided in TSD and will be available within a CDATA section[3]. The negative test cases are also represented with special elements and attributes for sensitive events, faulty edges, and fault classes.

Finally, the TSD tool is able to measure a set of model metrics that can be used by the testers to support decisions on manual effort during the MBT process. Table 5.4 shows the proposed metrics and their descriptions. They are all based on defined elements of the ESG4WSC model, as defined in Section 5.2.2. We provide further information about the metrics in Chapter 6, Section 6.3.

[3]All text within a CDATA section is ignored by the XML parser.



Conditions	...	...	...	...	...	...
fromCity	T	T	F	T	T	T
toCity	T	T	T	F	T	T
departureDate	T	T	T	T	F	T
returnDate	T	T	T	T	T	F
departureDate < returnDate	T	F	-	-	-	-

Actions	...	...	...	...	...	...
searchHotelAndFlights	x	-	-	-	-	-
TripInputException	-	x	x	x	x	x

Figure 5.6: DTs in TSD – extracted from (Belli et al., 2013).

Table 5.4: Metrics for ESG4WSC.

Name	Description
NPubReqE	Number of public request events.
NPrivReqE	Number of private request events.
NReqE	Number of request events, summing public and private ones.
NPubRespE	Number of public response events.
NPrivRespE	Number of private response events.
NRespE	Number of response events, summing public and private ones.
NRefE	Number of refined events.
NGenE	Number of generic events.
NE	Number of events (sum of all previous metrics).
NEdges	Number of edges.
NREsgs	Number of refining ESGs.
NEsgsInPar	Number of ESGs in parallel.
NDTs	Number of decision tables associated with events.
NConst	Number of constraints in DTs.
NAct	Number of actions in DTs.
NRules	Number of rules in DTs.
NUnPrivRespE	Number of distinct private response events.
NUnPubReqE	Number of distinct public request events and, if it is the case, their rules. If the public request event has an associated DT, it counts the number of rules for this event, and one, otherwise.
NUnPubRespE	Number of distinct public response events.
NUnMsgE	Number of distinct message events which includes public request and response events and private request and response events.

```
<TestCase>
 <CompleteEventSequence>
  <Event label="TA:queryTrip" type="request" public="true" rule="R1" >
   <Param name="departureDate">31.12.2011</Param>
   <Param name="toCity">Sao Carlos</Param>
     ...
  </Event>
  <RefinedEvent>
   <CompleteEventSequence>
    <Event label="IS:login" type="request" public="false"/>
    <Event label="IS:login_Response" type="response" public="false" />
    <Event label="IS:search" type="request" public="false"/>
    <Event label="IS:searchResults_greaterEqThanOne" type="response" public="false" >
     <Message><![CDATA[ ... ]]></Message>
    </Event>
   </CompleteEventSequence>
   <CompleteEventSequence>
    <Event label="FL:search" type="request" public="false"/>
    <Event label="FL:searchResults_greaterEqThanOne" type="response" public="false" >
     <Message><![CDATA[ ... ]]></Message>
    </Event>
   </CompleteEventSequence>
  </RefinedEvent>
  <Event label="TA:queryTrip_Response" type="response" public="true" />
  <Event label="TA:book" type="request" public="true" rule="R1" >
   <Param> ... </Param>
  </Event>
  <RefinedEvent>
   <CompleteEventSequence>
    <Event label="FL:book" type="request" public="false"/>
    <Event label="FL:bookingSuccess" type="response" public="false" >
     <Message><![CDATA[ <soap:Envelope><soap:Body> ... </soap:Envelope> ]]></Message>
    </Event>
   </CompleteEventSequence>
   <CompleteEventSequence>
    <Event label="IS:login" type="request" public="false"/>
    <Event label="IS:login_Response" type="response" public="false" />
    <Event label="IS:search" type="request" public="false"/>
    <Event label="IS:searchResults_sameHotelPrice" type="response" public="false" >
     <Message><![CDATA[ ... ]]></Message>
    </Event>
    <Event label="IS:book" type="request" public="false"/>
    <Event label="IS:bookingSuccess" type="response" public="false" >
     <Message><![CDATA[ ... ]]></Message>
    </Event>
   </CompleteEventSequence>
  </RefinedEvent>
  <Event label="TA:bookingConfirmation" type="response" public="true" />
  <Event label="TA:getAllOptions" type="request" public="true" rule="R1" >
   <Param name="searchCode">[$validSearchCode$]</Param>
  </Event>
  <Event label="TA:TripInputException" type="response" public="true" />
 </CompleteEventSequence>
</TestCase>
```

Figure 5.7: An XML file for a test case – extracted from (Belli et al., 2013).

5.5.2 Test Execution

Mule-ESB (MuleSoft, 2012) was used as the infrastructure software that provides ESB capabilities (more details on the ESB in Chapter 2, Section 2.2). Initially, all services involved in the composition (including the composite service) are deployed in the bus, i.e., the entire communication (SOAP messages) passes through the ESB before reaching the destination service.

The test execution is supported by a tool named Event Runner for Test Execution (ERunTE), which is composed of three modules, a Web service (ERunTE-service), an ESB component (ERunTE-esbcomp), and an event runner (ERunTE-runner). ERunTE-service contains four main operations:

- `startObservation`: prepares the test execution and identifies the start of a new test case. The partner services whose messages will be modified are passed as parameters.
- `modifyMessage`: sets the response message for a certain service. This operation is useful to force an event and test a specific scenario. Besides the expected events (positive tests), this operation also provokes the fault classes "unexpected fault", "wrong XML schema", "wrong XML syntax", "right schema, wrong data", and "missing service". In the "missing service" class, this operation turns off the proxy of the service, i.e., it simulates an unavailable service.
- `modifyMessageWithTimeout`: is similar to the previous operation, but it is possible to set up a delay to answer the response. This operation is used to support the fault classes "longtime response" and "no response".
- `getAllMessages`: retrieves all messages that pass through the ESB after the last *startObservation* call.

ERunTE-service can be used directly in the test code. In the test environment, it has been integrated in ERunTE-runner. The second module, ERunTE-esbcomp, is the ESB component that implements the monitor and aggregator patterns (Section 2.2). This component is integrated with Mule-ESB and is able to interact with ERunTE-service. The component records all messages that pass through the ESB and also modifies messages according to operation `modifyMessage`. Figure 5.8 summarizes the architecture adopted to execute the tests in this case study.

The test cases are implemented using Java/JUnit and executed with ERunTE-runner. Two adaptors are necessary to execute a test sequence like the one presented in Figure 5.7: *`PublicEventAdaptor`* and *`MessageCheckingAdaptor`*. The former is responsible for invoking and checking the messages of the composition interface, i.e., the public request and response events. The latter is responsible for checking SOAP messages produced during the test case execution. The adaptors were developed as classes with specific Java annotations for allowing ERunTE-runner to map methods to events in the test model. ERunTE-runner uses these two adaptors and the test sequence in XML generated by TSD to execute the tests. It works in three phases:

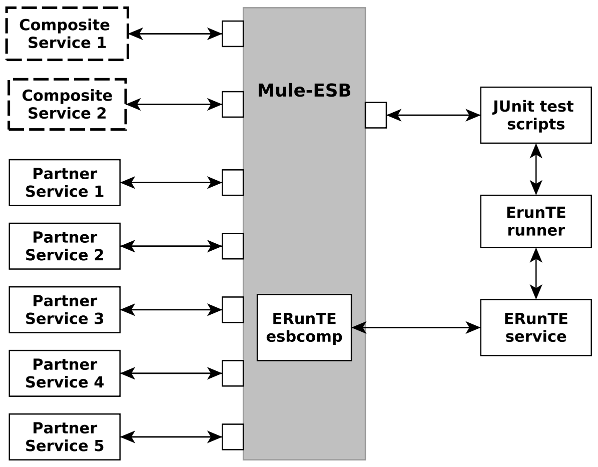


Figure 5.8: Architecture to execute the tests.

1. *Setting up private messages*: the runner parses the test sequence in XML and traverses the event sequence. It calls ERunTE-service to set up the messages for each private response. The message that must be returned is retrieved from the test case file, as presented in Figure 5.7 Line 31. The order is defined by counting previous response events of the same service. After this phase, the ERunTE tool has the configuration for executing the test case.

2. *Calling the public interface*: the test execution starts with requests for the service composition. Thus, `PublicEventAdaptor` is used to call the public events and check their responses. The test sequence is traversed and public events are executed by calling the respective methods in `PublicEventAdaptor`. If some response of the composition interface is different from the expected in the test sequence, the test case fails. Figure 5.9 presents a code snippet for `PublicEventAdaptor`. Line 03 presents annotation *@Event* for event “TA:queryTrip”. It defines that the marked method should be called for event “TA:queryTrip”. This event has an associated DT and this method (Lines 04-17) is specific for rule “R2”. Internal variables (starting with “S_”) are used to store temporary values used by the methods. All methods return a Boolean value to represent a successful/failed execution. For instance, the method in Lines 20-28 returns **true** if event “TA:queryTrip_Response” is correctly executed and **false**, otherwise.

3. *Checking messages*: the runner retrieves all SOAP messages produced during the test case execution using ERunTE-service. The messages are checked using `MessageCheckingAdaptor`. The sequence of events is also expected to be followed. If the number of messages, the messages, or the event order are not in accordance with the test sequence, the test case fails. Figure 5.10 presents a code snippet for `MessageCheckingAdaptor`. The methods are also marked with annotation *@Event*. The methods return a Boolean value and have a string as parameter. The string represents the message to be checked. The method returns **true** if the message represents the labeled

event and **false**, otherwise. In this case study, XPath (Berglund et al., 2010) expressions are used to check whether the messages are correct or not. Lines 04-13 and 16-25 check if a message is event "TA:queryTrip" and "TA:queryTrip_response", respectively.

```
01:public class PublicEventsAdaptor {
02:  ...
03:  @Event(label="TA:queryTrip", rule="R2")
04:  public boolean m01() {
05:    S_exception = null;
06:    TripSearchData tripSearchData = new TripSearchData();
07:    tripSearchData.setFromCity("Paderborn");
08:    tripSearchData.setToCity("Sao Paulo");
09:    tripSearchData.setDepartureDate( 2012, 07, 19 );
10:    tripSearchData.setReturnDate( 2012, 07, 13 );
11:    try {
12:      travelAgentService.queryTrip(tripSearchData);
13:    }catch (Exception e) {
14:      S_exception = e;
15:    }
16:    return true;
17:  }
18:  ...
19:  @Event(label="TA:queryTrip_Response")
20:  public boolean m11() {
21:    if(S_tripOptions == null)
22:      return false;
23:    if(S_tripOptions.getFlightInfos().size() != 5)
24:      return false;
25:    if(S_tripOptions.getHotelInfos().size() != 5)
26:      return false;
27:    return true;
28:  }
29:  ...
30:}
```

Figure 5.9: Sample code for *PublicEventAdaptor* – extracted from (Belli et al., 2013).

```
01:public class EventCheckerAdaptor {
02:  ...
03:  @Event(label="TA:queryTrip")
04:  public boolean isTA_queryTrip(String message) {
05:    try {
06:      NamespaceContext context = new NameSpaceContextMap("soap", "http://./soap/envelope/",
                                  "serv", "http://TravelAgent.service.triphandling.cs/");
07:      xpath.setNamespaceContext(context);
08:      Node node = xpath.evaluate( "/soap:Envelope/soap:Body/serv:queryTrip",
                           new InputSource(new StringReader(message)), XPathConstants.NODE);
09:      return node != null;
10:    }
11:    catch (XPathExpressionException e) { }
12:    return false;
13:  }
14:  ...
15:  @Event(label="TA:queryTrip_Response")
16:  public boolean isTA_queryTripResponse(String message) {
17:    try {
18:      NamespaceContext context = new NameSpaceContextMap("soap", "http://./soap/envelope/",
                                  "serv", "http://TravelAgent.service.triphandling.cs/");
19:      xpath.setNamespaceContext(context);
20:      Node node = xpath.evaluate("/soap:Envelope/soap:Body/serv:queryTripResponse",
                           new InputSource(new StringReader(message)), XPathConstants.NODE);
21:      return node != null;
22:    }
23:    catch (XPathExpressionException e) { }
24:    return false;
25:  }
26:  ...
27:}
```

Figure 5.10: Sample code for *MessageCheckingAdaptor* – extracted from (Belli et al., 2013).

Negative testing also requires special configurations of the test execution environment. ERunTE-esbcomp implements a configurable delay for fault classes "no response" and "longtime response". ERunTE-service has an operation to shut down service proxies, helping to reproduce fault class "missing service". For fault class "unexpected fault", possible unexpected fault messages have been identified and simulated. For example, SOAP-Faults thrown by Web service frameworks when exceptions are not correctly handled in the application. For fault classes "wrong XML syntax" and "wrong XML schema", ERunTE-runner makes small modifications in the original messages to reproduce these faults.

The test generation and execution is fully supported by the TSD and ERunTE tools, albeit improvements are required in the adaptor development. The version control between model and adaptors is needed and repeated code can be generated for the adaptors.

5.6 Final Remarks

In this chapter, we have proposed an event driven approach, named ESG4WSC, for MBT of service compositions. Test cases are generated based on an ESG4WSC model verifying desired scenarios (positive testing) and unexpected situations (negative testing). To support the test generation and execution, two tools have been developed, TSD and ERunTE.

ESG4WSC brings the benefits of black box-oriented MBT to service compositions, providing a holistic approach for positive and negative testing. Black-box tests can be generated by modeling an ESG4WSC and observing/modifying messages exchanged in the composition. Thus, faults are detected by observing the exchanged messages. The approach can be applied to many different scenarios. However, its strength stems from its potential to fit well for cases where WS-BPEL or WS-CDL specifications are not available. Thus, it is independent of the type of composition, either orchestration or choreography, and therefore allows to simultaneously performing the steps for implementation and testing of the SUT. Other testing approaches strictly require the availability of the artifacts, such as WS-BPEL codes and WS-CDL specifications.

Based on the literature review presented in Section 2.5, we can state that: *(i)* there is no comparable work that supports the holistic testing of composite services by modeling and testing not only the published composition interface, but also the internal communication with partner services; *(ii)* no other approach has solved the oracle problem using sensitive events for negative testing of service compositions; and *(iii)* it has been described how an ESB can support the test execution. In our case, the ESB is used to perform the observations and modifications of exchanged messages as well as to provoke the unexpected situations for negative testing.

When a testing approach is proposed, it is important to have evidences about its practical application. However, the research on service testing falls short on experimental studies as discussed in Section 2.5.1. In this context, the next chapter presents the experimental evaluation of the ESG4WSC approach.

CHAPTER

6

Evaluation of the Proposed Approach

6.1 Overview

The type of knowledge in software testing and, widely in software engineering, can be considered as of relatively low maturity (Juristo et al., 2004). In this context, the conduction of experimental studies has been a constantly researched topic in the last years (Kitchenham et al., 2002; Briand, 2007). This fact occurs not only for software testing in general, but also for specific areas as service testing (Bozkurt et al., 2012). As for existing approaches to test service-oriented applications, there is a need for more detailed experimental studies and industrial experiences using real-world systems, as discussed in Section 2.5.1.

This chapter presents three experimental studies conducted to evaluate the ESG4WSC approach introduced in Chapter 5. Section 6.2 summarizes a case study presented in paper *"A Holistic Approach to Model-based Testing of Web Service Compositions", Belli, F., Endo, A. T., Linschulte, M. and Simao, A.,* published in the Software: Practice and Experience journal (Belli et al., 2013). In this case study, we aim at evaluating the applicability of the aforementioned approach. The ESG4WSC was applied to generate positive and negative test suites for a large and complex application, named `xTripHandling`.

Section 6.3 reproduces a cost analysis that aims to evaluate the machine CPU time and the human effort during the steps of the approach. The two algorithms for generating positive test suites (proposed in Chapter 5) are compared in order to identify trade-offs concerning test generation and execution. Model metrics to support modeling and test concretization are also analyzed.

Section 6.4 partially extends an industrial experience described in technical report *"Using Models to Test Web Service-Oriented Applications: an Experience Report"*, *Endo, A. T.*, *Silveira,* *M. B., Rodrigues, E. M., Simao A., Oliveira, F. M., Zorzo, A. F.* (Endo et al., 2012). In this industrial experience, we aim at evaluating the proposed approach in the context of an IT company. Initially, we used a set of composite services specified in WS-BPEL to analyze the modeling and test generation steps. Then, we report the use of the ESG4WSC approach in an ongoing project, focusing on concretization and test execution.

Finally, Section 6.5 describes the lessons learned during the experimental evaluation, and Section 6.6 concludes by discussing the results and analyzing the limitations of the three studies.

6.2 Case Study: xTripHandling

This section describes a case study carried out to evaluate of the proposed approach for positive and negative testing of service compositions. It describes the application used as subject in the case study, as well as its configuration and results.

6.2.1 System Under Test

The case study was conducted using the `xTripHandling` application, which is based on different scenarios proposed in technical and research literature (Mei et al., 2009a; Sourceforge.net, 2012; NetBeans.org, 2009). The application was developed using SOA concepts and Web services and provides a set of facilities to query and book trips. It also includes facilities to book trains, rent cars, book sightseeing, and order maps. The application consists of eight services, of which six are single services and two are composite services. The single services are the following:

1. *ISELTA-hotel Service*: is a Web service provided by commercial system ISELTA that enables travel and touristic enterprises to create their individual search and service offering masks (Belli and Linschulte, 2010). It provides operations to query hotels and manage bookings.

2. *Airlines Service*: provides a set of operations to manage flight tickets, which are similar to ISELTA-hotel service.

3. *Map Service*: provides operations to locate places (e.g., airports, train stations) close to a city and order maps for certain cities.

4. *Car Rental Service*: provides operations to search and rent vehicles to be used in a predefined city.

5. *Train Service*: provides operations to check train lines between cities and buy train tickets.

6. *Sightseeing Service*: provides operations to list available cities in which the service operates and to buy tickets for sightseeing.

The composite services are the following:

1. *Travel Agent Service*: provides a set of facilities to query and book a trip. *Travel Agent Service* interacts with two services, *ISELTA-hotel* and *Airlines* services. It combines these two services, providing operations to search and book a travel involving flight and hotel reservation. As the flight ticket and hotel reservation are essential in any travel, a successful booking using this service guarantees hotel and flight reservations.

2. *Customer Service*: combines the services *Travel Agent, Airlines*, *Map*, *Car Rental*, *Sightseeing*, and *Train* to provide a centralized resource for customers to manage various aspects of a trip, including hotels, flights, maps, trains, cars, and sightseeing.

Figure 6.1 illustrates the services, their interfaces, and the interactions of composite services. The figure presents a summarized version of the WSDL interfaces. The dashed lines represent the interaction between composite services and partner services.

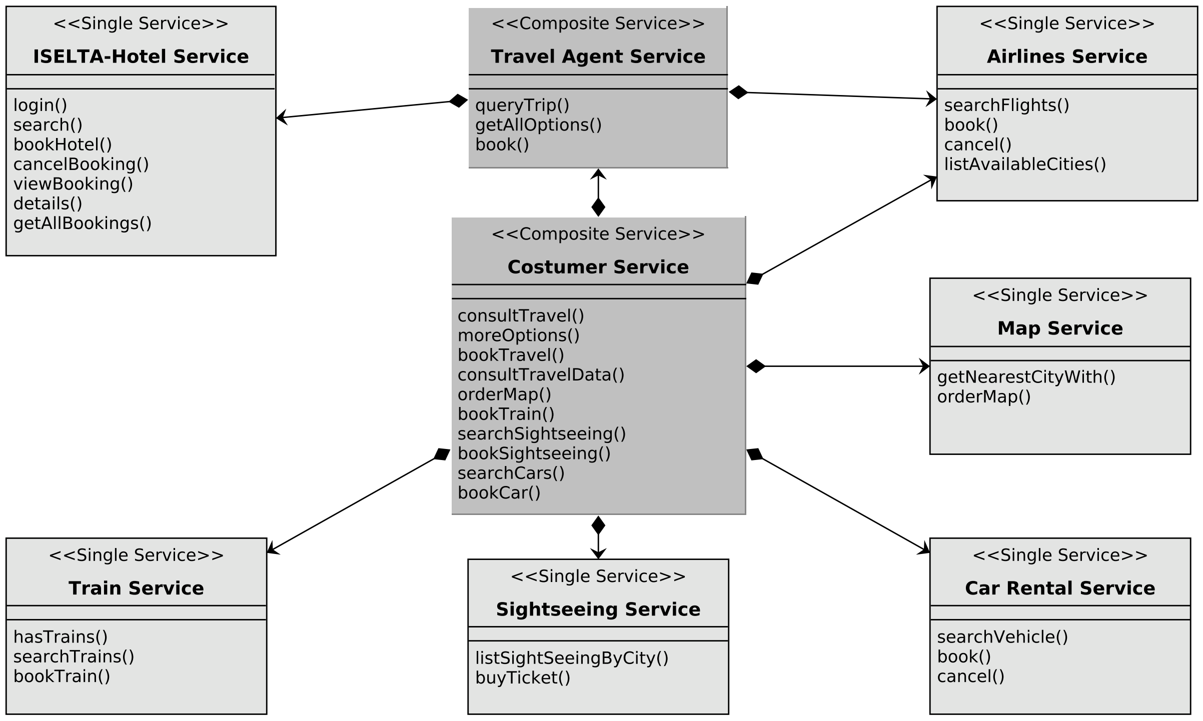


Figure 6.1: Service interfaces in `xTripHandling` – adapted from (Belli et al., 2013).

The complete workflow specification for the composite services (*Customer Service* and *Travel Agent Service*), as well as the interface descriptions (WSDL and Java interfaces), can be found on `http://www.labes.icmc.usp.br/~aendo/esg4wsc`.

6.2.2 Configuration and Results

The ESG4WSC approach was applied to test the composite services during the development of the `xTripHandling` application. The case study involved a developer and a tester. The developer described a functional specification which was used to implement the services. The specification and service interfaces were provided to the tester that created ESG4WSC models and associated DTs. Since the approach is black box-oriented, the tester had no access to the source code. Test cases were derived according to the ESG4WSC approach. The tester also performed the concretization and execution of the test cases.

The correction of faults in the case study was based on the following scheme: when the tester finds a fault (some test case fails), the testing process is interrupted. The tester analyzes the origin of the fault. If both the specification and the model are correct with respect to the test case, the tester concludes that the fault is in the implementation. *(i)* If the fault is in the specification, developer and tester update the specification and the tester updates the model based on the new version of the specification. *(ii)* If the fault is in the model, the tester updates the model to fix the mistake. In case of a change in the model, test cases are regenerated and the tester resumes the testing process. *(iii)* If the fault is in the implementation, the developer executes the test case and fixes the fault with the restriction that all previously executed test cases, including the failed test case, must also pass. This scheme is followed until all positive and negative test cases are successfully executed.

The ESG4WSC approach was applied to test the two composite services in `xTripHandling`, *Travel Agent Service* and *Customer Service*. First, *Travel Agent Service* was tested, since its partner services are just single ones. Then, the approach was applied to *Customer Service*. The testing process based on the established fault correction scheme had several iterations, producing different versions of models and test suites.

Table 6.1 summarizes the information about the ESG4WSC models. Lines 1-4 refer to the number of each type of event. Lines 5 and 6 show the total number of events and edges, respectively. Line 7 refers to the number of refining ESG4WSCs and those that are to be executed in parallel in Line 8. Lines 9-11 show the number of DTs, constraints, and rules, respectively.

Table 6.1: Test model information – extracted from (Belli et al., 2013).

		Travel Agent	*Customer Service*
1:	# request events	15	204
2:	# response events	34	449
3:	# generic events	1	13
4:	# refined events	2	34
5:	**# events (total)**	**52**	**700**
6:	**# edges (total)**	**75**	**947**
7:	# refining ESG4WSCs	4	68
8:	# ESGs in parallel	4	44
9:	# DTs	7	108
10:	# constraints	32	197
11:	# rules	46	300

Refining ESG4WSCs in *Travel Agent Service* were used exclusively to represent parallel execution, i.e., the four refining ESG4WSCs are in parallel. In *Customer Service*, the model contains 34 refined events and 68 refining ESG4WSCs, being 44 in parallel. Notice that 24 refining

ESG4WSCs are not in parallel and were used to modularize the model. Thus, refined events and refining ESG4WSCs were used not only to represent parallelism, but also to manage the complexity through hierarchy. For instance, after booking a basic trip (hotel + flight), the client can search and book a car. This workflow can be abstracted as a refined event “rentCar” and its details expressed in an associated refining ESG4WSC. Similar refined events were defined for maps, sightseeing, and trains, using the hierarchy of refining ESG4WSCs to organize the model.

In both models, since a request message can have several relevant instances of response messages, there are more response events than request events. For example, a search request can return one of the following responses: *(i)* a message with zero items, *(ii)* a message with one or more items, or *(iii)* an expected fault. Generic events facilitate the description of time constraints or changing points. DTs mainly supplement public request events for which input data must be generated. The constraints are defined over request parameters and rules test different combinations of these constraints. In addition, DTs were also used to prune extra edges in refined events with parallel execution.

Using the designed test models, the TSD tool generated test suites according to the holistic ESG4WSC approach. Table 6.2 summarizes the information about the test suites, divided into positive and negative testing. The number of executed events for each test suite is also provided. Positive test suites are divided by the length of covered ESs ($k = 2$, $k = 3$, and $k = 4$). Negative test suites are divided into public and private cases.

Table 6.2: Test suite information – extracted from (Belli et al., 2013).

	Travel Agent	*Customer Service*
Positive testing		
#test cases ($k = 2$)	25	1,054
#executed events ($k = 2$)	388	89,536
#test cases ($k = 3$)	33	998
#executed events ($k = 3$)	540	139,388
#test cases ($k = 4$)	49	20,537
#executed events ($k = 4$)	922	3,406,148
#test cases (total)	107 (25+33+49)	22,589 (1,054+998+20,537)
#executed events	1,850	3,635,072
Negative testing		
#PubFESs	181	6,535
#executed events PUBFESs	1,706	152,125
#PriFES (F_{NR})	10	95
#PriFES (F_{MS})	10	95
#PriFES (F_{UF})	10	95
#PriFES (F_{LR})	24	194
#PriFES (F_{WSc})	24	194
#PriFES (F_{WSy})	24	194
#PriFES (F_{WD})	24	194
#PriFESs	126	1,061
#executed events PriFESs	1,544	22,486
#test cases (total)	307 (181+126)	7,596 (6,535+1,061)
#executed events	3,250	174,611

Since the intended coverage is dependent on the model characteristics, the size difference observed in Table 6.1 is also observed in the number of test cases. For the *Travel Agent Service* model, 107 positive test cases and 307 negative test cases were generated. For the *Customer Service* model, $22,589$ positive test cases and $7,596$ negative test cases were generated.

In positive testing, the increase of k causes a higher number of executed events. However, these test suites do not need to be applied one-by-one. For instance, if the tester chooses test suites with $k = 4$, test suites with lengths 2 and 3 can be skipped since the test requirements for smaller lengths are contained in $k = 4$ as well (in terms of coverage).

In negative testing, the number of test cases for a given fault class is equal to the number of private request or response events (as shown in Table 5.3). That is why $\mathcal{F}_{\text{NR}}$, $\mathcal{F}_{\text{MS}}$, and $\mathcal{F}_{\text{UF}}$ have the same number of test cases. Similarly, $\mathcal{F}_{\text{LR}}$, $\mathcal{F}_{\text{WSc}}$, $\mathcal{F}_{\text{WSy}}$, and $\mathcal{F}_{\text{WD}}$ also have the same amount of test cases.

It is important to emphasize that the number of test cases is only used to show the computational effort in this study. Since the test suites are automatically generated from the model, the manual effort is mainly measured by the modeling and concretization steps.

Table 6.3 presents the information about faults detected using positive and negative test suites. Faults are also divided by the artifact, specification (Spec) or implementation (Impl). After several iterations, the tester found 18 faults in *Travel Agent Service*, 12 in the implementation and six in the specification. The six faults were related to some behavior not described in the specification and have been observed during the test case execution. Faults related to missing and unexpected messages have been detected for both specification and implementation.

Table 6.3: Detected faults information – extracted from (Belli et al., 2013).

Test Suites	*Travel Agent*		*Customer Service*	
	Spec	Impl	Spec	Impl
positive test suites				
k=2	4	11	16	12
k=3	0	0	0	1
k=4	0	0	0	0
Total	4	11	16	13
negative test suites				
PriFES ($\mathcal{F}_{\text{NR}}$)	1	0	1	0
PriFES ($\mathcal{F}_{\text{LR}}$)	0	0	0	0
PriFES ($\mathcal{F}_{\text{MS}}$)	0	0	0	0
PriFES ($\mathcal{F}_{\text{UF}}$)	0	0	0	0
PriFES ($\mathcal{F}_{\text{WSc}}$)	0	0	0	0
PriFES ($\mathcal{F}_{\text{WSy}}$)	0	0	0	3
PriFES ($\mathcal{F}_{\text{WD}}$)	0	0	0	0
PubFES (resp-req)	0	0	0	1
PubFES ([-req)	0	0	0	0
PubFES (req-req)	1	1	1	2
Total	2	1	2	6
positive and negative test suites				
Total	6	12	18	19

Customer Service presents a higher number of specification faults because its initial specification was very incomplete. The tester found 12 specification faults while designing the first version of the model and six other faults afterwards. Although *Customer Service* had a high number of executed test cases, the faults in the implementation were mainly identified during the first executed positive and negative test cases. In the end, 37 faults were detected using positive and negative test suites.

Table 6.3 also shows the order in which the test suites were executed. Thus, all positive test suites were first applied using the established fault correction scheme. As a consequence, this order obviously reduced the number of faults to be detected by subsequent test suites. Note that most of the faults were detected by the positive test suites covering sequences of length $k = 2$. This result corresponds to experimental results achieved in previous studies for testing graphical user interfaces (Belli et al., 2010). Although longer test sequences facilitate the detection of critical faults that can only be detected in specific contexts, test suites with length 2 detected most of the faults (Belli et al., 2010). The negative test suites uncovered less faults for both compositions due to the fact that several exceptions were properly handled in the implementation when the developer fixed the detected faults.

The faults in the implementation were mainly identified by testing different rules (from the DTs) and checking expected events and their order. The correction of these faults was not critical and was performed only in the implementation. The specification faults were identified by checking expected event sequences. The specification faults were more critical since the tester needed to modify the ESG4WSC and regenerate the test cases. Moreover, the specification and, consequently, the implementation were corrected as well. As the modeling task is a learning activity, faults were also introduced and identified in the test models. However, an accurate number of these faults is not available since they were directly and dynamically corrected by the tester.

6.3 Cost Analysis

This section presents a cost analysis of the ESG4WSC approach while considering the different steps of MBT. First, we compare the two algorithms that can be used to generate CESs that cover all EPs: *(i)* the Chinese postman problem presented in Section 5.3.2 and *(ii)* the event tree algorithm presented in Section 5.3.3. We aim to compare their performance with respect to the costs of generation and execution. Then, we analyze the model metrics introduced in Section 5.5.1, as well as their relationship with test modeling and concretization.

6.3.1 Test Generation and Execution

Setup To conduct this experiment, we adopted random models which were generated in accordance with the formal definitions of the ESG4WSC model described in Chapter 5, Section 5.2.2.

The generation of random models was performed as follows. First, the algorithm creates the number of events specified as input. Then, event pairs are randomly connected. Finally, the produced model is validated against Definition 3. If the model is not valid, it is discarded and a new model is generated.

ESG4WSC models were randomly generated with different sizes, ranging from 50 to 1000 events. For each model size (e.g., 250 events), 100 different models were generated and average values were calculated for the analyzed variables. Since the test generation step in the ESG4WSC approach is automated by TSD, we measured the CPU time to analyze the cost of test generation. The measured time represents how long the given algorithm takes to produce the test suite from the model used as input. In order to analyze the cost of test execution, the size of the test suite, that is, the number of events to be executed was used as the main measure. Moreover, we also collected additional measures of the generated test suites, such as the number of CESs (test cases) and the average length of test cases.

The results were computed using a computer with AMD Turion 64x2 2.0 GHz with 2 GB RAM running Windows XP. The two algorithms were implemented in the TSD tool. In the rest of this section, we refer to the Chinese postman problem based algorithm as *CPP* and the event tree algorithm as *ETA*.

Results Figure 6.2 shows how the generation time (in seconds) varies with respect to the number of events. Notice that *ETA* is much faster than *CPP* in all cases; this is more evident with large models. The *ETA* algorithm shows approximately a linear growth, taking less than one second to produce test suites for models until 750 events and below two seconds for larger models. In the *CPP* algorithm, the generation time increases in large models, mainly those with over 500 events, though it is below 20 seconds. However, for models with up to 400 events, *CPP* is able to produce test suites in less than one second.

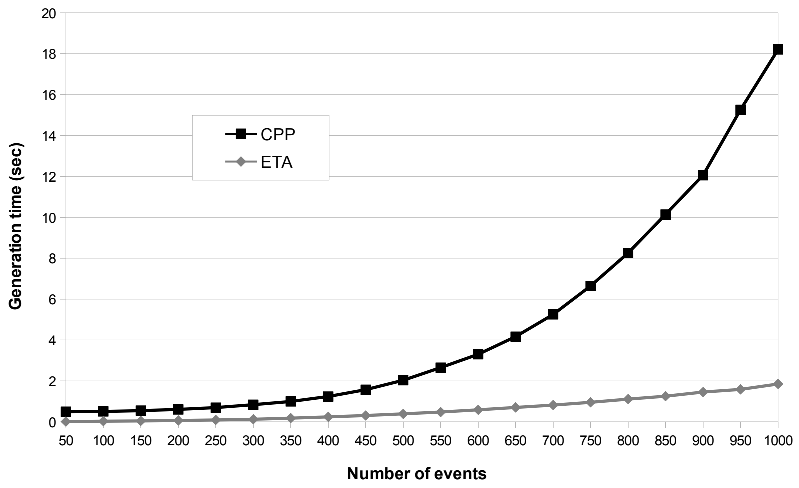


Figure 6.2: Generation time in seconds (sec) varying the number of events.

Figure 6.3 shows how the test suite size varies with respect to the number of events. Notice that *CPP* produces test suite lengths smaller than *ETA*, though both algorithms seem to generate test suites that increase linearly. The *CPP* algorithm produces a minimum-cost solution to cover each edge (Aho et al., 1995) and for the ESG4WSC models used in the experiment, it means around twice the number of events in the model.

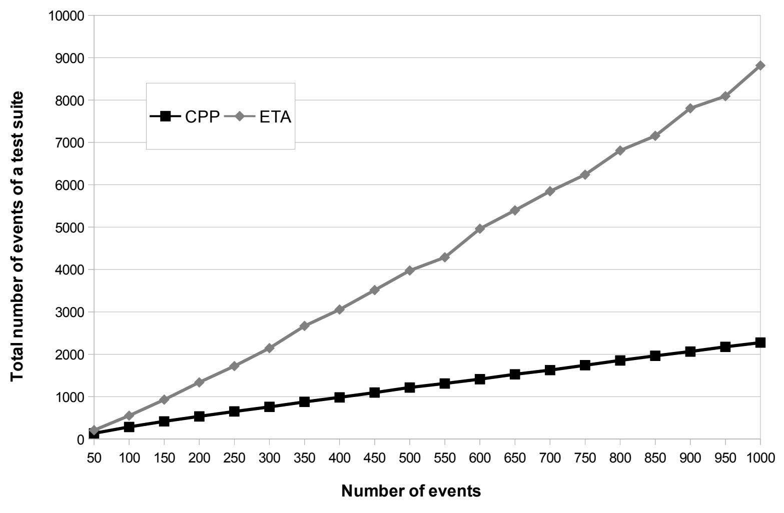


Figure 6.3: Test suite size varying the number of events.

Figure 6.4 shows how the number of CESs (test cases) varies with respect to the number of events. Notice that the number of CESs grows linearly in test suites generated by *ETA*, while *CPP* produces nearly constant number of CESs even in models with higher number of events. In models with over 250 events, the number of CESs varies between 30 and 39 test cases. The *ETA* algorithm produces test suites with a higher number of test cases, which increases proportionally in large models.

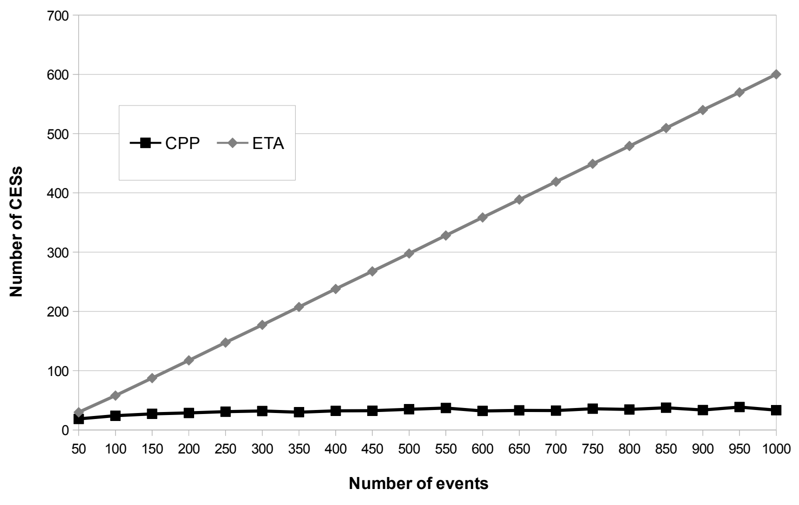


Figure 6.4: Number of CESs varying the number of events.

Figure 6.5 shows how the average test case length varies with respect to the number of events. Notice that *CPP* produces test suites with average test case lengths that are longer than *ETA*. The *ETA* algorithm generates test suites with shorter test cases and that have a slight growth in models with more events. In models with over 200 events, the average test case length varies between 11 and 15 events. The *CPP* algorithm produces average test case lengths that, albeit showing some variation, grow in models with more events. For instance, in models with 1000 events, *CPP* shows an average length of 82.04, while *ETA* shows an average length of 14.69.

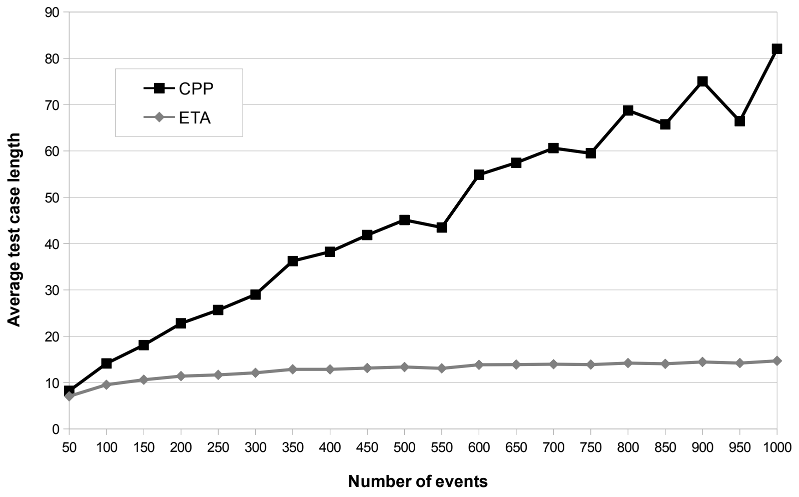


Figure 6.5: Average test case length varying the number of events.

6.3.2 Test Modeling and Concretization

To complement the previous analysis on automated parts of the ESG4WC approach, we herein discuss the two steps that require manual effort: modeling and concretization. We revisit the case study previously described in Section 6.2, collecting and analyzing data about the manual effort spent.

Table 6.4 shows the model metrics (Section 5.5.1) that give an overview of the two models designed in the `xTripHandling` case study. The two models have different size and complexity. On one hand, the model for *Travel Agent Service* is smaller since it involves two partner services. On the other hand, the model for *Customer Service* is considerably more complex and larger involving a great number of events and edges. Although the ESG4WSC model provides the feature of refining events and hiding complexity in a given layer, the *Customer Service* model required much more manual effort to handle. The initial modeling time was around eight hours for the *Travel Agent Service* and around 20 hours for *Customer Service*. The overall time spent by the tester to perform improvements on the models during the test sessions was not measured. We observed during the tests that the MBT process results in many iterations which hinders a detailed analysis of the modeling step. However, we roughly estimate that it took about the same amount of time to improve the model as the initial modeling time.

Table 6.4: Model metrics for the `xTripHandling` case study.

Metric Name	**Description**	*Travel Agent Service*	*Customer Service*
NPubReqE	N. of public request events.	5	64
NPrivReqE	N. of private request events.	10	57
NReqE	N. of publ. and priv. req. events.	15	121
NPubRespE	N. of public response events.	10	171
NPrivRespE	N. of private response events.	24	116
NRespE	N. of publ. and priv. resp. events.	34	287
NRefE	N. of refined events.	2	31
NGenE	N. of generic events.	1	17
NE	N. of all events.	52	456
NEdges	N. of edges.	95	812
NREsgs	N. of refining ESGs.	4	65
NEsgsInPar	N. of ESGs in parallel.	4	44
NDTs	N. of DTs associated with events.	7	72
NConst	N. of constraints in DTs.	32	145
NAct	N. of actions in DTs.	15	155
NRules	N. of rules in DTs.	46	212

To perform the concretization in the ESG4WSC approach, the tester deals with two artifacts, the test model and the adaptor code. The analyzed metrics for both model and adaptor code are shown in Tables 6.5 and 6.6, respectively. Table 6.5 shows the model metrics that are more related to the concretization step. These metrics differ from the previous ones by removing duplicate events from the final value. Table 6.6 shows source code metrics for the two required adaptors defined in Section 5.5: *`MessageCheckingAdaptor`* (MCA) and *`PublicEventAdaptor`* (PEA). The metrics are divided by the two composite services under test. For each implemented adaptor, it shows the total LoC, the number of methods, the average LoC per method, and the CC.

Table 6.5: Model metrics related to concretization.

Metric Name	**Description**	*Travel Agent Service*	*Customer Service*
NUnMsgE	N. of distinct message events.	34	85
NUnPubReqE	N. of distinct pub. request events.	14	36
NUnPubRespE	N. of distinct pub. response events.	6	17
NUnPrivRespE	N. of distinct priv. response events.	18	38

Table 6.6: Source code metrics for adaptors.

	Travel Agent Service		*Customer Service*	
Metric Name	MCA	PEA	MCA	PEA
number of methods	37	22	87	51
avg LoC per method	6.2	14.1	7.3	12.9
CC	1.7	2.3	1.9	3.2
total LoC	392	416	1004	893

The NUnMsgE metric can be used to count how many different SOAP messages are produced during the tests. This metric gives an idea on the number of XPath queries to be produced in order to validate the SOAP messages observed during a test case execution. Thus, NUnMsgE is related to the `MessageCheckingAdaptor` adaptors since they are responsible for checking whether a given message is the right event. Note that the number of methods (column MCA) in Table 6.6 is quite similar to the value of NUnMsgE in Table 6.5. On the other hand, the complexity of implementing methods of `MessageCheckingAdaptor` adaptors can only be measured by source code metrics. Using the current versions of the tools, each method that checks if a SOAP message is a given event has around 6.2-7.3 LoC and around 1.7-1.9 CC.

The NUnPubReqER metric can be used to show the number of pieces of code to be implemented in order to invoke the composite service. Moreover, the NUnPubRespE metric can be used to show the number of pieces of code to be implemented in order to perform the validation/assertion of the response sent back by the composite service. Thus, the NUnPubReqER and NUnPubRespE metrics are related to `PublicEventAdaptor` adaptors. Observe that the number of methods (column PEA) in Table 6.6 is quite similar to the sum of NUnPubReqER and NUnPubRespE in Table 6.5, i.e., 22-22 for *Travel Agent Service* and 51-53 for *Customer Service*. As in `MessageCheckingAdaptor` adaptors, the complexity of implementing methods of `PublicEventAdaptor` adaptors can only be measured by source code metrics. Using the current versions of the tools, each method has around 12.9-14.1 LoC and about 2.3-3.2 CC.

The association between a private response event and its respective SOAP message file (to be returned during the test execution) is performed through the TSD tool. Thus, the NUnPrivRespE metric can be used to show how many SOAP messages should be created by hand. It requires a manual effort to produce these messages and associate them using the tool. In the case study, the messages were generated using the SoapUI tool. Since these activities are carried out using TSD, this metric has no relation with any of the source code metrics presented.

6.4 Experience on Industrial Setting

This section reports an industrial experience with the ESG4WSC approach. This study was conducted in cooperation with a multinational IT company and with the Pontifícia Universidade Católica do Rio Grande do Sul. The company provided access to its applications, as well as technological support and documentation. The communication with the IT company happened through online means (e-mail, instant messaging) and meetings with the development team. All data presented herein was produced and collected using service-oriented applications developed and used internally by the company.

This study was divided into two parts. First, we focused on the modeling and test generation over WS-BPEL based composite services. Second, we analyzed the full application of the ESG4WSC approach in the context of the ABC project[1], emphasizing the concretization and test execution.

[1]For the sake of confidentiality, we replace the original labels by generic ones for all services along the text.

6.4.1 Part 1: Test Modeling and Generation

In this part, we used 23 composite services specified in WS-BPEL to evaluate the ESG4WSC's modeling capabilities. Then, we evaluated the test suites generated from these models. The sources of information about the services were the WS-BPEL specification itself, WSDL files, logs of the WS-BPEL engine, and talks to the developers.

Table 6.7 shows the test model's characteristics for each of the modeled services. Observe that all services are asynchronous (request-only) since the number of public response events is zero in all lines. The number of public request events is low, most of them with one or two requests. Only services `BCS-03` and `BCS-20` have more public requests. The number of private request and response events reflects the complexity of communication with the partner services. The overall complexity of the test models can be summarized by the number of events and edges. Clearly, `BCS-11` has the large test model with 447 events and 501 edges, followed by `BCS-22`, `BCS-09`, and `BCS-20`. In services `BCS-11`, `BCS-20`, and `BCS-22`, refined events and refining ESG4WSCs were adopted to deal with the complexity of many events and edges. ESG4WSCs in parallel were not needed in these models. Most of the branches in the models are caused by different types of responses, instead of input parameters. This is observed by the number of DTs and their elements (constraints, actions, and rules) that is low. `BCS-20` has the highest number of DTs which is consistent with its public request events; `BCS-10` has the highest number of constraints for one DT. Ten out of 23 services do not have associated DTs.

For the test models, positive and negative test suites were generated. Table 6.8 shows the test suite's information divided by the type of testing. For each type of test suite, the number of executed events is also shown as a measure of cost. All test suites were automatically generated using the TSD tool. The cost of positive test cases is highly dependent on number of events and edges in the model. `BCS-11` has the highest number of test cases (40) and `BCS-01`, `BCS-15`, and `BCS-17` has the lowest number of test cases (2).

For PubFESs, the cost is related to the number of public request and response events. `BCS-03` and `BCS-20` have the highest number of test cases and cost, 49 test cases executing 98 events and 25 test cases executing 222 events, respectively. Sixteen out of 23 services have only one test case.

The PriFESs are divided in accordance with the fault classes described in Table 5.3 (Section 5.4). The test suites for the fault classes NR, MS, and UF are dependent on the number of private request events. They have different costs because each fault class has its own characteristics during the execution. For instance, in class NR, the affected request event happens and no response is produced, while in class UF a response event is triggered. `BCS-11`, `BCS-09`, and `BCS-22` have the highest number of test cases.

The test suites for the fault classes LR, WSc, WSy, and WD are dependent on the number of private response events. `BCS-11`, `BCS-09`, and `BCS-22` also have the highest number of test cases. `BCS-03`, `BCS-08`, and `BCS-23` have no test case for these fault classes since their models do not contain private response events (as shown in Table 6.7).

Table 6.7: Test model information – adapted from (Endo et al., 2012).

Service Name	#Public Request Events	#Private Request Events	#Request Events	#Public Response Events	#Private Response Events	#Response Events	#Refined Events	#Generic Events	#Events	#Edges	#Refining ESG4WSCs	#ESGs in Parallel	#Decision Tables	#Constraints	#Actions (next events)	#Rules
BCS-01	1	3	4	0	2	2	0	2	8	10	0	0	1	1	2	2
BCS-02	1	8	9	0	7	7	0	0	16	25	0	0	1	2	3	3
BCS-03	7	7	14	0	0	0	0	0	14	21	0	0	0	0	0	0
BCS-04	2	15	17	0	13	13	0	0	30	42	0	0	1	1	2	2
BCS-05	1	8	9	0	16	16	0	0	25	39	0	0	1	1	2	2
BCS-06	2	15	17	0	14	14	0	1	32	45	0	0	1	1	2	2
BCS-07	1	9	10	0	7	7	0	0	17	26	0	0	1	3	4	4
BCS-08	1	2	3	0	0	0	0	0	3	6	0	0	1	2	3	3
BCS-09	1	38	39	0	33	33	0	0	72	87	0	0	1	1	2	2
BCS-10	1	26	27	0	13	13	0	2	42	56	0	0	1	14	14	14
BCS-11	1	235	236	0	181	181	11	19	447	501	11	0	1	4	4	4
BCS-12	1	24	25	0	22	22	0	5	52	63	0	0	0	0	0	0
BCS-13	2	5	7	0	4	4	0	0	11	15	0	0	0	0	0	0
BCS-14	1	5	6	0	5	5	0	0	11	16	0	0	0	0	0	0
BCS-15	1	2	3	0	1	1	0	2	6	8	0	0	1	1	2	2
BCS-16	2	6	8	0	11	11	0	0	19	30	0	0	0	0	0	0
BCS-17	1	4	5	0	3	3	0	0	8	10	0	0	0	0	0	0
BCS-18	1	7	8	0	9	9	0	0	17	24	0	0	0	0	0	0
BCS-19	1	4	5	0	4	4	0	0	9	12	0	0	0	0	0	0
BCS-20	6	26	32	0	13	13	2	6	53	65	2	0	6	6	12	12
BCS-21	1	13	14	0	14	14	0	4	32	44	0	0	0	0	0	0
BCS-22	1	33	34	0	29	29	2	8	73	86	2	0	0	0	0	0
BCS-23	2	3	5	0	0	0	0	0	5	9	0	0	1	2	3	3

The last two columns show the total number of PriFESs, including all seven fault classes. As the negative testing for private events produces test suites that cover all request and response events in combination with the fault classes, a high number of negative test cases is generated. For all models, its cost exceeds the cost of positive testing.

Table 6.8: Test suite information – adapted from (Endo et al., 2012).

Service Name	#Positive Test Cases	#Executed Events	#Public Faulty Event Sequences	#Executed Events	#Private Faulty Event Sequences (NR)	#Executed Events	#Private Faulty Event Sequences (MS)	#Executed Events	#Private Faulty Event Sequences (UF)	#Executed Events	#Private Faulty Event Sequences (LR)	#Executed Events	#Private Faulty Event Sequences (WSc)	#Executed Events	#Private Faulty Event Sequences (WSy)	#Executed Events	#Private Faulty Event Sequences (WD)	#Executed Events	Total # #Private Faulty Event Sequences	#Executed Events
BCS-01	2	7	1	2	3	12	3	9	3	15	2	6	2	8	2	8	2	8	18	68
BCS-02	8	47	1	2	8	36	8	28	8	44	7	33	7	33	7	33	7	33	53	242
BCS-03	7	14	49	98	7	14	7	7	7	21	0	0	0	0	0	0	0	0	70	140
BCS-04	10	80	6	28	15	103	15	88	15	118	13	54	13	67	13	67	13	67	103	592
BCS-05	14	97	1	2	8	46	8	38	8	54	16	80	16	96	16	96	16	96	89	508
BCS-06	11	85	6	28	15	103	15	88	15	118	14	56	14	70	14	70	14	70	107	603
BCS-07	9	39	1	2	9	30	9	21	9	39	7	18	7	25	7	25	7	25	56	185
BCS-08	3	5	1	2	2	4	2	2	2	6	0	0	0	0	0	0	0	0	7	14
BCS-09	15	159	1	2	38	327	38	289	38	365	33	210	33	243	33	243	33	243	247	1922
BCS-10	14	53	1	2	26	78	26	52	26	104	13	26	13	39	13	39	13	39	131	379
BCS-11	40	822	1	2	235	3923	235	3688	235	4158	181	2623	181	2804	181	2804	181	2804	1430	22806
BCS-12	10	173	1	2	24	285	24	261	24	309	22	243	22	265	22	265	22	265	161	1895
BCS-13	4	24	5	17	5	23	5	18	5	28	4	13	4	17	4	17	4	17	36	150
BCS-14	5	27	1	2	5	22	5	17	5	27	5	16	5	21	5	21	5	21	36	147
BCS-15	2	5	1	2	2	6	2	4	2	8	1	2	1	3	1	3	1	3	11	31
BCS-16	10	55	4	8	6	28	6	22	6	34	11	40	11	51	11	51	11	51	66	285
BCS-17	2	13	1	2	4	18	4	14	4	22	3	10	3	13	3	13	3	13	25	105
BCS-18	7	47	1	2	7	36	7	29	7	43	9	38	9	47	9	47	9	47	58	289
BCS-19	3	15	1	2	4	14	4	10	4	18	4	10	4	14	4	14	4	14	29	96
BCS-20	9	112	25	222	26	250	26	224	26	276	13	88	13	101	13	101	13	101	155	1363
BCS-21	11	140	1	2	13	103	13	90	13	116	14	103	14	117	14	117	14	117	96	765
BCS-22	8	132	1	2	33	402	33	369	33	435	29	310	29	339	29	339	29	339	216	2535
BCS-23	4	7	4	8	3	6	3	3	3	9	0	0	0	0	0	0	0	0	13	26

The cost of generating the test suites is low since it is automatically performed by the tool. For the largest model, the tool took less than 11 seconds to produce the positive and negative tests. We have provided the test suite information as an additional measure to predict the cost of execution.

During the modeling of the 23 services, limitations were identified on the ESG4WSC modeling technique and tool. We describe these limitations and practical solutions for each of them as follows.

Event branch: it happens when some event branch is solved by some event or input parameter (in a DT) that happens previously in the workflow. In Figure 6.6(a), events `resp01_1` and `resp01_2` are used to select the branch in event `req03`. Although this design is acceptable in the exploratory modeling, Figure 6.6(b) shows a solution to support test generation. The event sequence between the solving events (`resp01_1` and `resp01_2`) and the branch (`req03`) needs to be replicated. A drawback is that the replicated piece of model can be large and difficult to manipulate.

ForEach in parallel: the activity ForEach in parallel from WS-BPEL 2.0 and the flowN extension of Oracle BPEL engine introduce the possibility of executing n request events in parallel. As the parallelism is implicitly introduced and the number of threads is only decided in runtime, ESG4WSCs in parallel are not able to directly represent this case in the proposed model. Figure 6.7(a) shows an example so that the graph within the box can be executed n times in parallel. Figure 6.7(b) outlines a solution assuming that there will be two instances (threads). Thus, the

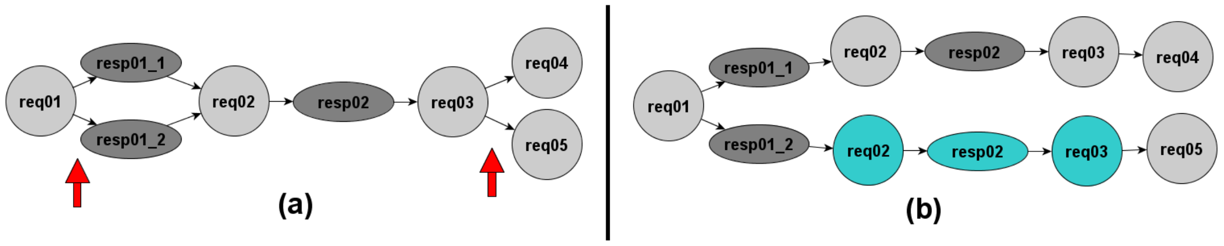


Figure 6.6: Model snippets for the event branch issue.

tester defines the number of instances before the test generation and replicates the ESG4WSCs in parallel in a refined event. The tester needs to know and define the number of instances (threads) in modeling time, which is a disadvantage.

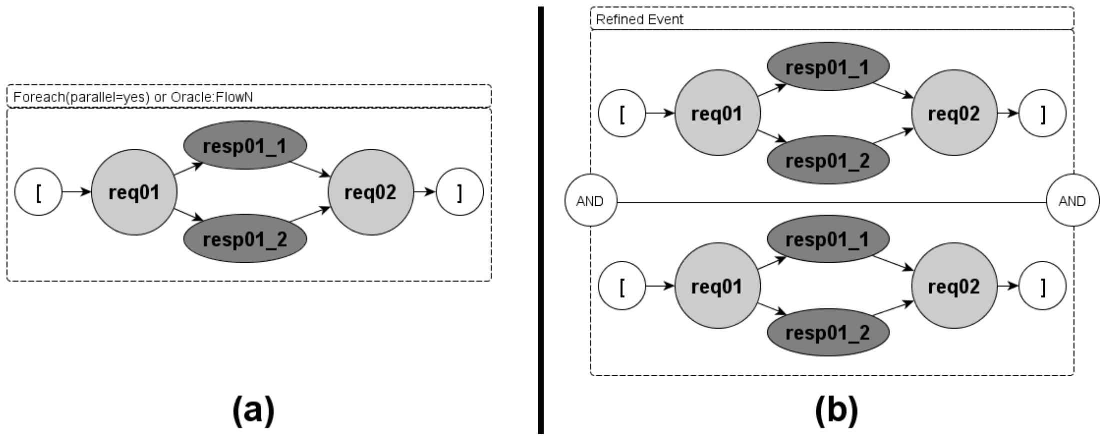


Figure 6.7: Model snippets for the forEach in parallel issue.

Private events within loops: this case is similar to the previous one, as the number of iterations in a loop is decided in runtime. Figure 6.8(a) shows a model snippet usually obtained during the exploratory modeling (events `req01` and `resp01_1` are within a loop). Figure 6.8(b) depicts a solution so that the loop is extended in three iterations. The tester should identify the number of iterations and repeat the instances in the model before generating the tests. The drawback here is also to have some previous knowledge on the runtime execution of the composite service.

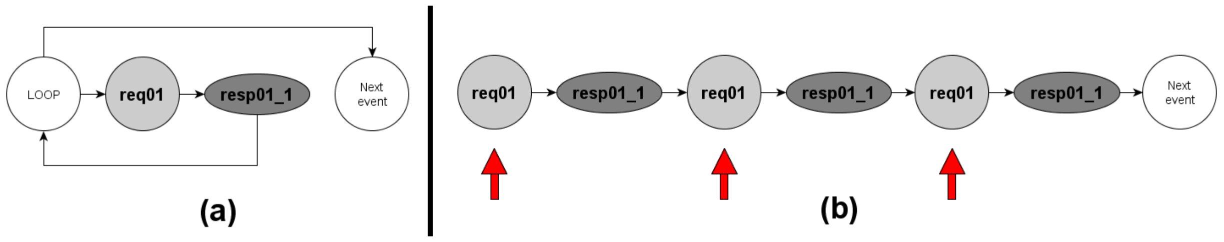


Figure 6.8: Model snippets for the loop issue.

Global/internal variables: WS-BPEL engines have the concept of global variables that are independent and assigned outside the scope of the composition. However, they can be referred within the WS-BPEL and define different branches. Internal variables are more common in composite services implemented in traditional programming languages, instead of WS-BPEL. These composite services tend to interact with databases and modify the workflow depending on internal variables. These global and internal variables cannot be represented in the ESG4WSC model. Figure 6.9(a) illustrates a case in which the branch after event `resp01_1` is solved by global variable `var`. A practical solution is to establish preconditions to the model or to the test cases. Figure 6.9(b) depicts the splitting in two models with preconditions (`var=true` and `var=false`). These preconditions have to be handled during the test execution which entails an extra effort from testers.

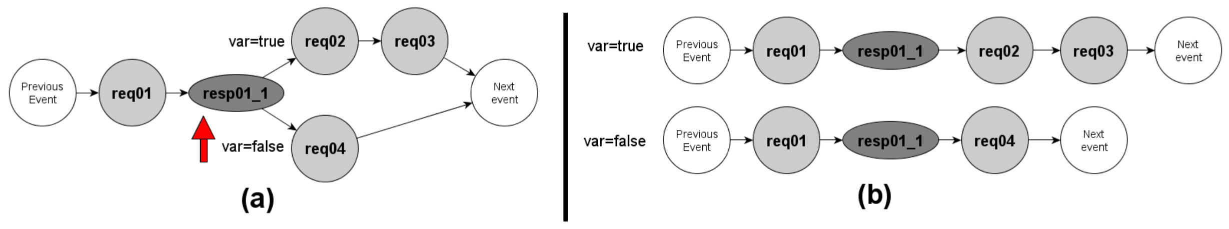


Figure 6.9: Model snippets for the variables issue.

6.4.2 Part 2: Test Concretization and Execution

In this part, we applied the ESG4WSC approach in the ABC application, specifically in its composite service *ABCService* (ABCS). This service interacts with three other services: *PartnerService01* (PS01), *PartnerService02* (PS02), and *PartnerService03* (PS03). Table 6.9 shows the tested services, whether they are composite or not, their total number of operations, and number of operations involved in the tests.

Table 6.9: Information about services – adapted from (Endo et al., 2012).

Service Name	composite service?	#operations	#involved operations
ABCService	yes	8	2
PartnerService01	no	26	2
PartnerService02	no	12	1
PartnerService03	no	10	1

Based on performance test scripts and on meetings with the development team, the test model shown in Figure 6.10 was designed. It focuses on the flow of messages triggered by operations operation01 and operation02 of *ABCService*. This model was augmented with SOAP messages for the public request events and private response events.

The four services involved in the tests were deployed in Mule-ESB. As services *PartnerService01*, *PartnerService02*, and *PartnerService03* have some security issues, we used SoapUI to mock them. A simple modification was performed in *ABCService*, the original service endpoints were changed to the ones provided by the ESB. Thus, all messages produced during the tests passed through the bus and were controlled by the module *esbcomp* of the ERunTE tool.

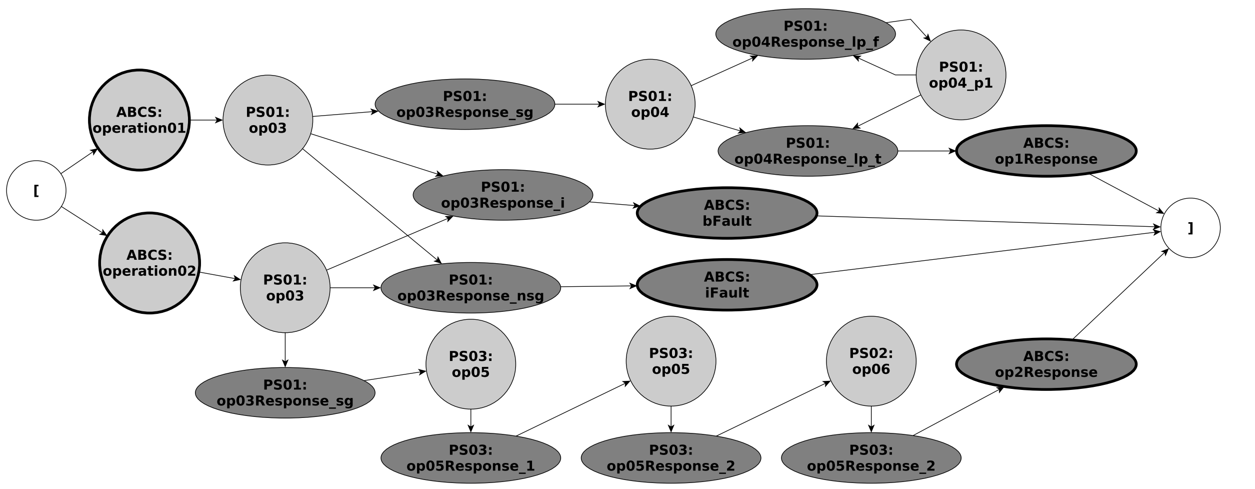


Figure 6.10: ESG4WSC model for the ABC application – adapted from (Endo et al., 2012).

Using the model in Figure 6.10, test cases were generated using the TSD tool. Table 6.10 summarizes the test suites generated for positive and negative testing. In total, 68 test cases were generated.

Table 6.10: Number of positive and negative test cases – adapted from (Endo et al., 2012).

Positive testing		
Test Suite	**#Test cases**	**Execution time**
#CESs	7	≈ 13s

Negative testing		
Test Suite	**#Test cases**	**Execution time**
#PubFESs	4	≈ 5s
#PriFESs (NR)	7	≈ 75s
#PriFESs (MS)	7	≈ 515s
#PriFESs (UF)	7	≈ 13s
#PriFESs (LR)	9	≈ 96s
#PriFESs (WSc)	9	≈ 12s
#PriFESs (WSy)	9	≈ 5s
#PriFESs (WD)	9	≈ 12s

The next step was the concretization of the tests. By doing so, two adaptors were implemented, *PublicEventAdaptor* and *MessageCheckingAdaptor*. Moreover, a couple of additional classes were implemented to configure and run the tests. Modules *runner* and *service* of ERunTE were used to support the test execution with the developed adaptors. Table 6.11 shows the number of LoC and the CC for the two adaptors and the entire project. The metrics were collected using Eclipse Metrics (SourceForge.net, 2005).

All test cases were successfully executed; the approximate execution time is also presented in Table 6.10. The test suite for fault class MS took more time than the others (around 515 seconds). This happened due to a limitation in the ERunTE tool that requires more time to simulate a missing service. The test suites for fault classes NR and LR also took more time since the ERunTE tool simulates timeouts for these PriFESs.

Table 6.11: Code metrics for the test project (adaptors, setup code) – adapted from (Endo et al., 2012).

	LoC	**average CC**
`PublicEventAdaptor`	94	1.5
`MessageCheckingAdaptor`	231	1.8
Entire test project	900	1.6

During the tests, no fault was detected in the SUT. This can be explained by the application's stability. The application has been released for more than two years and many cycles of testing/ maintenance were performed.

6.5 Lessons Learned

In the case study described in Section 6.2, the tester designed the entire models before implementing the adaptors for test execution. This strategy fits for cases in which the development phase is ongoing and the implementation is not yet available. However, it takes some time to have test cases executing in the implementation. If the implementation has a preliminary version, the test model and adaptors can be built partially and iteratively to obtain some executable test cases sooner. We followed this strategy in the industrial experience (Section 6.4) since the `ABC` application had an up-and-running version.

Depending on the modeling strategy, the number of test cases can increase very rapidly since the applied methods intend to cover event sequences of a given length in the ESG4WSC model. Eliminating irrelevant edges was a strategy used in the case study (Section 6.2) to reduce the test cases. For instance, for the *Customer Service* model, several edges to and from special vertices '[' and ']' were removed. This simple change eliminated a considerable number of test cases.

The current version of the ESG4WSC approach addresses seven fault classes for negative testing of private events (Section 5.4.2). Additional classes can be defined and implemented using the current infrastructure. Notice that covering all fault classes generates a high number of negative test cases (as shown in Tables 6.2 and 6.8). If the cost of test execution is critical in the current project, a subset of the negative test suite can be selected. Based on the case study experience, a strategy is to concentrate on the fault classes "longtime response" and "unexpected fault". They are usually enough to test the service composition robustness, since the fault correction for those classes indirectly handles other fault classes.

The evaluation of negative test results should be performed carefully. The information used in negative testing (undesired situations) is usually misleading, scarce, and even missing. For instance, the "longtime response" class can expose unplanned issues that must be handled by the composition, such as timeouts in the implementation, missing specification, and incomplete workflows. This fact hinders an accurate and automatic evaluation of test sequences. Therefore, a mechanism was presented to handle this issue by using *sensitive events* (Section 5.4.2). This strategy avoids false positives, but faults can be missed by the test cases. Thus, it is recommended that the tester inspects a subset of each fault class to avoid false negatives.

During the first part of the industrial experience (Section 6.4), we noticed that the ESG4WSC modeling was more effective when divided into two steps. This configuration of steps was intuitively performed during the modeling of the first four services. After a phase of identification, all WS-BPEL based composite services were modeled in two steps and therefore with two model versions. The two steps are detailed as follows:

1. *Exploratory modeling:* an initial ESG4WSC model was designed, identifying request and response events. The order among them was also modeled. During this step, the global communication is prioritized and branches and DTs are put aside. Generic events and comments were used to recall that these issues need to be handled in future.

2. *Test-driven modeling:* using the model designed in the previous step, a more detailed analysis was conducted to identify and model DTs (constraints, rules, and actions). Generic events and comments were removed and branches along the model were solved. The goal of this step is to set up a model that is adequate to generate test cases. During this step, we identified the limitations and solutions described at the end of Section 6.4.1.

6.6 Discussion of Results and Limitations

The case study in Section 6.2 demonstrated that the approach is applicable to a non-trivial service-oriented application. Numerous faults were revealed, not only in the implementation but also in the specification. Thus, the approach helped to keep implementation and specification synchronized. Moreover, the specification has been set up first and the implementation and test model creation has been done in tandem by two different people. Hence, the tester does not need to wait for the implementation to set up a model and derive tests since the proposed approach is not based on artifacts like WS-BPEL or WS-CDL as in (Mei et al., 2009a; Hou et al., 2009; Bentakouk et al., 2009).

Orchestration and choreography have not been distinguished since the approach can be applied in both contexts. The only restriction is that the tester should have control over messages exchanged by the partner services. The use of the ERunTe tool requires that all messages pass through an ESB, as described in Section 5.5.2. Although ESBs may not be part of the service-oriented application under test, deploying the services in an ESB is a simple task.

The proposed approach assumes that ESG4WSCs in a refined event are independent. This enables a simple way to model some parallelism and, during the experimental studies, this was enough. It is possible that more complex scenarios occur in service compositions and the tester might also want to test combinations of message interleaving. Although the approach can be adapted to test these scenarios, it is recommended to use specific modeling and testing techniques for concurrent programs (Hoare, 2004; Lei and Carver, 2006).

The studies were conducted to evaluate the approach with regard to its applicability to the development of a service-oriented application, involving composite services of very different sizes. Although it has been noticed that the approach could systematically detect faults (as in Section 6.2), further experimental research is necessary considering alternative application areas and sizes. Moreover, investigating in which scenarios MBT is not cost-effective remains an open topic, e.g., when the costs of maintaining a model and generating tests are higher than the costs of maintaining a traditional test suite.

In Chapter 5, we presented the target fault classes for both positive and negative testing. During the case study, the results evinced that these faults can be revealed by the approach. Besides, different faults were also identified in the specification as observed in the MBT literature (Pretschner et al., 2005; Utting and Legeard, 2006). However, we do not compare the proposed approach with other techniques, e.g., structural testing. In other domains, experimental results on this topic have shown that MBT approaches tends to complement the structural testing technique (Mouchawrab et al., 2007). Further research on this comparison will be carried out in future work.

The high number of test cases and events executed might be a limitation in some contexts. Thus, a smaller but still effective test suite is desirable. In Section 6.5, we discussed a strategy to reduce the test suite by modifying the model. Another strategy would be to modify the test case generation algorithms. Intuitively, this seems to be a promising candidate for further research since most of the faults were detected by the first test cases executed.

While using MBT, very often the users assume that the underlying model is correct from the beginning. This assumption is a potential threat to the validity since it cannot hold in practice. From a practical point of view, this is a critical assumption since a (formal) model set up on an (informal) specification regularly needs some time to reflect the specification correctly. The tester should always consider that a mistake might be made in the test model and not only in the specification or in the implementation when a fault is detected.

Requirements evolution is also one of the main benefits of MBT (Utting and Legeard, 2006) and is also shown in the case study (Section 6.2). This is particularly relevant for dynamical and loosely coupled environments based on SOAs (Josuttis, 2007; Papazoglou and Heuvel, 2007). As the model is usually smaller and easier to maintain than a large test suite, it is faster to modify the model and regenerate the test suite when the requirements (specification) change (Utting and Legeard, 2006). If no event is added to test model, the cost of regenerating the test suite is equivalent to the cost of executing the test generation algorithms. This process can be systematically controlled by tracking the changes between the specification and the tests. Nevertheless, the requirement-test traceability was not handled in this dissertation.

The ESG4WSC approach, and MBT approaches in general, can be divided into four main steps: *(i)* modeling, *(ii)* test case generation, *(iii)* concretization, and *(iv)* test execution (Chapter 2, Section 2.4). Each step has its own cost which may include human effort and/or machine CPU time.

Test model information gives an idea on the human effort that would be spent since the tester is supposed to design it manually. Although the modeling effort is directly related to the application's size and complexity, we observed in the industrial experience (Section 6.4) that other factors may also influence it. The restricted access to sources of information and unclear test purposes may increase the cost during the modeling. In the performed studies, the TSD tool was essential since the models were constantly manipulated. Large models remain complex to manipulate, motivating the investigation of different reduced models instead of only one. The scope of the presented data on test modeling for service-oriented applications is limited and more experimental evaluation is required. Future work is also necessary to deal with limitations in the ESG4WSC modeling technique described at the end of Section 6.4.1.

Concerning the test generation, we have proposed two different algorithms for positive test generation (Section 5.3). Based on the experiment we conducted with random models (Section 6.3), CPP and ETA have both advantages and disadvantages. They have shown different performance with respect to the test suite size (which impacts on the test execution). Furthermore, test suites are generated all the time and a fast algorithm for test generation is essential. The results have shown that with models until 500 events (relatively large), the generation time is acceptable for both algorithms (below two seconds). While ETA executes faster, CPP produces shorter test suites. Another key difference is the quantity and length of test cases generated by each algorithm. ETA generates many short test cases, where CPP generates a few, but longer, test cases. The use of random models, albeit posing a limitation on extrapolation of the results to real projects, allows to investigating the relationship between the model size and the test suites.

Although test concretization is essential in the proposed approach (also in MBT), there is a lack of investigation into the cost of this step. Using the ESG4WSC approach, concretization metrics and source code metrics have shown that this step demands a reasonable effort (Sections 6.3.2 and 6.4.2). However, the current effort on concretization can be reduced by further improvements in the tools. We also note that the proposed model metrics can be helpful to estimate manual effort and support decisions by the testers.

During the second part of the industrial experience (Section 6.4.2), discussions with the development team suggested that most of the functionalities tested by the ESG4WSC approach were likely covered by previous tests. Nevertheless, we noticed a lack of automated solutions for testing Web services. Our impressions are that the MBT approach can be useful in scenarios, similar to the presented ones, that have complex workflows and mocking different services and sequences of messages are too complex and error-prone. However, more robust and automated tools would be essential to a large scale adoption.

We observed that there is still room for improvements in the approach automation. Concerning the tools' evolution, the main goal is to increase the automation and reduce the manual effort by, for instance, reducing the LoC necessary to develop adaptors. There are a couple of options that requires further investigation and experimentation. The generation of SOAP messages that are associated with private responses may be eased by integrating the SoapUI tool (Eviware, 2012).

In the adaptors development, `MessageCheckingAdaptor` may use the XML schema in the WSDL files to support automatic verification of messages. Most of the written code can be generated automatically. Moreover, XPath queries that currently are evaluated in the adaptor code can be included directly in the model (and in the XML test cases as well). Thus, ERunTE-runner would be in charge of reading XPath queries and evaluated them, working as a test oracle. ESB configurations may also be automatically performed and integrated with the development environment. Finally, the `PublicEventAdaptor` adaptors can be evolved for automatic driver construction and input data generation.

6.7 Final Remarks

This chapter has presented three experimental studies that were conducted to evaluate the ESG4WSC proposed in Chapter 5. Section 6.2 has described a case study so that the ESG4WSC approach was employed during the development of the `xTripHandling` application. The results of this case study suggested that the approach scales well with larger compositions. Moreover, faults were detected not only in the SUT, but also in the specification of a non-trivial service-oriented application for managing trips.

Section 6.3 has described the results obtained by carrying out a cost analysis of the ESG4WSC approach. As for test generation and execution, advantages and disadvantages for the two algorithms have been identified using random models. *ETA* is faster and produces shorter test cases, which favor scenarios with bigger models and that require test cases easier to debug. *CPP* generates smaller test suites with few test cases, being adequate for scenarios where the test execution cost should be minimized. As for modeling, the obtained experience has supported ESG4WSC as an intuitive technique to test composite services. Regarding test concretization, testers devoted a reasonable effort, but straightforward use, in test concretization as observed in the source code metrics.

Section 6.4 has described an experience report on applying the proposed approach in a set of real-world applications of a multinational IT company. The results on modeling and test generation of 23 WS-BPEL based composite services have been described. We have also provided more details on the concretization and test execution for the `ABC` application. The experience reported has provided preliminary evidences that the ESG4WSC approach is applicable to test service-oriented applications in real and less controlled scenarios within an IT corporation. From the results, we have analyzed issues that impact the approach and tools, discussing how they can be overcome.

The lessons learned, discussion of results, and limitations of these three experimental studies are presented in Sections 6.5 and 6.6. All in all, we have recognized the threats and limitations of each study and general conclusions cannot be drawn from the results presented in this chapter. However, the three studies have provided evidences that at least MBT, more specifically the

ESG4WSC approach, is worth to be applied in service-oriented applications from both academic and industrial points of view. We can also conclude that the automation capability is also promising based on the use of the supporting tools.

The next chapter concludes this dissertation, summarizing the main contributions, discussing general limitations, and mentioning future work.

CHAPTER 7

Conclusion

Software testing is a key factor to successfully implement projects that develop applications using SOA and Web services. Nevertheless, these service-oriented applications pose challenges that cannot be overcome by the use of traditional software testing techniques. Service testing has been widely researched over the past years, though the systematic and formal testing of this class of software and its appropriate automated support have been still motivating topics.

In this context, this dissertation has contributed with theoretical and experimental studies to advance the service testing area by applying MBT. The results have provided a positive answer to the general research question proposed in Chapter 1, i.e., *"Is MBT applicable to test service-oriented applications so that test cases are generated to verify the SUT formally in a holistic way?"*.

The contributions that support answering such a question are revisited in Section 7.1. Section 7.2 summarizes the limitations of the work and discusses how they can be overcome.

7.1 Revisiting the Dissertation Contributions

This section revisits the achievements of this dissertation as follows.

Experimental comparison of FSM-based test methods: we conducted an experimental study to compare five methods, namely W, HSI, SPY, H, and P, capable of generating test suites from FSMs, and the results are provided in Chapter 3. For each method, we analyzed the test suite characteristics regarding number of resets and test case length (Section 3.5.1). The overall cost of each method, i.e., the test suite length was analyzed in Section 3.5.2. We applied mutation testing to simulate a domain so that the fault detection ratio could be evaluated (Section 3.5.3). Results of the correlations among the analyzed dimensions: number of resets, test case length, test suite length, and fault detection ratio are provided (Section 3.5.4).

MBT process for service-oriented applications: we introduced a testing process for service-oriented applications, revisiting the MBT literature. By doing so, a set of steps, tools, and artifacts was identified and described in Section 4.3. We carried out an exploratory study, described in Section 4.4, to provide an initial evaluation of the process. This study consisted of *(i)* the development of a tool called JStateModelBasedTest (Section 4.4.1) and *(ii)* a case study with two applications: `ThirdPartyCall-SOA` and `QualiPSo-Factory` (Section 4.4.2).

Holistic testing of service compositions: we proposed an MBT approach, named ESG4WSC, to test service compositions in a holistic and formal way. The approach, presented in Chapter 5, involves a modeling technique that represents the communication of a composite service with its partner services, formally defined in Section 5.2.2. We also proposed algorithms to generate positive and negative test suites from an ESG4WSC model. Positive test suites cover expected (or desirable) situations in the test model (more details in Section 5.3) and negative test suites cover public and private events to verify unexpected (or undesirable) situations in the test model (more details in Section 5.4).

Mechanisms to support the test automation: prototype tools were implemented and evaluated along the development of this doctoral work. As for the ESG4WSC approach, we cooperated with researchers from Paderborn Universität to develop the tools described in Section 5.5 of Chapter 5. To deal with modeling and test generation, the TSD tool was augmented to support definitions and algorithms of the ESG4WSC approach (Section 5.5.1). Regarding concretization and test execution, we developed the ERunTE tool, whose main task is the integration with an ESB to monitor and control the message flow (Section 5.5.2).

Experimental evaluations of the proposed approach: we evaluated the ESG4WSC approach and its supporting tools in the three experimental studies reported in Chapter 6. Initially, the approach was employed in the `xTripHandling` application. The results provided evidences that the approach can be applied to larger and complex compositions and faults can be revealed in both specification and SUT (more details in Section 6.2). A cost analysis was also conducted and preliminary results identified trade-offs between the two algorithms for positive test generation and suggested reasonable manual effort on modeling and concretization (more details in Section 6.3). Finally, an experience with real-world applications within an IT corporation was reported. The study identified limitations in the modeling technique and provided evidences that the approach is feasible in real and less controlled contexts (more details in Section 6.4).

Table 7.1 connects the reported achievements with three types of contributions: *(i)* theoretical definitions, *(ii)* experimental studies, and *(iii)* supporting tools (Maldonado, 1997). The table also provides the section in which the contribution was described and associated publication.

Table 7.1: Classification and structure of the dissertation contributions.

Chapter	**Description**	**Location**	***(i) theoretical definitions***	***(ii) experimental studies***	***(iii) supporting tools***	**Publication**
3	Experimental results from the comparison among FSM-based test methods w.r.t. test suite characteristics	Section 3.5.1		✓		(Endo and Simao, 2013)
	Experimental results from the comparison among FSM-based test methods w.r.t. the overall test suite length	Section 3.5.2		✓		
	Experimental results from the comparison among FSM-based test methods w.r.t. fault detection ratio	Section 3.5.3		✓		
	Correlations of analyzed variables when comparing FSM-based test methods	Section 3.5.4		✓		
4	MBT process for service-oriented applications	Section 4.3	✓			(Endo and Simao, 2011)
	Exploratory study – JStateModelBasedTest tool	Section 4.4.1			✓	
	Exploratory study – case study	Section 4.4.2		✓		
5	Proposal of the ESG4WSC approach – model	Section 5.2.2	✓			(Belli et al., 2013)
	Proposal of the ESG4WSC approach – positive test case generation	Section 5.3	✓			
	Proposal of the ESG4WSC approach – negative test case generation	Section 5.4	✓			
	Tool support for modeling and test generation (TSD)	Section 5.5.1			✓	
	Tool support for concretization and test execution (ERunTE)	Section 5.5.2			✓	
6	Case study: `xTripHandling`	Section 6.2		✓		(Belli et al., 2013)
	Cost analysis	Section 6.3		✓		
	Industrial experience	Section 6.4		✓		(Endo et al., 2012)

7.2 Limitations and Future Directions

This section describes limitations of the dissertation contributions and how they can be dealt with in the future. Notice that specific limitations have already been discussed in previous chapters, therefore we herein concentrate on broader limitations to be overcome as well as possible improvements that can be made during short- and medium-term research.

Comparison of FSM-based test methods: five methods able to generate n-complete test suites were considered for the comparison of the FSM-based test methods described in Chapter 3. The study does not take into account all methods existing in the literature, though replications can include others, like DS (Hennie, 1964), UIOv (Vuong et al., 1989), and Wp (Fujiwara et al., 1991). The adoption of random FSMs is another limitation that might be overcome by replications using real-world models as subjects. The experimental investigation into test methods for partial and nondeterministic machines also requires further studies.

Limitations of the ESG4WSC approach: during the experience with real-world applications shown in Section 6.4, limitations on the modeling technique were identified. In future work, elements of the ESG4WSC model defined in Chapter 5 should be revisited and extended to handle these limitations. Concerning the test generation, a high number of test cases was observed for positive and negative testing. In this case, the tester might need to either execute only a subset of the test cases or order the test case execution. Further investigations into strategies for test suite minimization and prioritization can shed some light on this topic.

Experimental studies: the ESG4WSC approach was evaluated by three experimental studies described in Chapter 6. Their results provide encouraging evidences, but the scope was limited and opportunities for future experiments are manifold. First, an experiment can be conducted to compare the proposed approach with other approaches, like structural testing. In these experiments, the most interesting dimensions are the effort devoted by the testers (cost) and the fault detection ratio (effectiveness). It is also worth conducting controlled experiments with human subjects to assess the manual efforts involved, such as modeling time, ease of use and learning curve, and mistakes made.

Tool support: the tools described in Chapters 4 and 5 are prototypes and several improvements, such as version control of test models and generated test suites in TSD and mechanisms to reduce the development effort of adaptors in ERunTE may be incorporated to increase the level of automation. The overall approach automation and its MBT steps can benefit from increasing the synergy between tools and service artifacts specified in Web service standards. Another extension to be assessed in the future is the integration of TSD and ERunTE with IDEs (e.g., Eclipse) since parts of the testing process are performed within those environments.

7.2.1 Possible extensions

Some of the possible extensions to the contributions of this dissertation include:

- Research into the automatic generation of partial event-driven models out of WS-BPEL specifications, reducing the initial effort to produce test models and keeping an updating traceability between models and SUTs.

- Opportunities for supporting performance testing. For instance, the *esbcomp* (integrated with the ESB) module of ERunTE could be extended to capture metrics used to evaluate and monitor performance, such as response time and throughput. Extensions could also be proposed in the modeling technique to introduce performance testing information that guides the generation of model-based performance tests.

- Improvements in the test modeling capabilities. FSMs and ESGs have been applied independently along this research project. We plan to investigate how to evolve an ESG4WSC model to a state machine that explicitly represents not only events, but also states, coping with the stateful behavior of composite services. This may also require a considerable increase in the automation power of the presented tools.

- Other topics to be exploited, such as online testing and dynamic compositions. Extensions can be proposed to generate on-the-fly (online) tests based on observed outputs of the composition. Although dynamic service compositions have not been widely adopted by practitioners yet, the ESG4WSC can also be investigated in the context of dynamically assembled services.

Bibliography

AALST, W. M. P. Formalization and verification of event-driven process chains. *Information & Software Technology*, v. 41, n. 10, p. 639–650, 1999.

ABRIAL, J.-R.; HALLERSTEDE, S. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundamenta Informaticae*, v. 77, n. 1-2, p. 1–28, 2007.

ACTIVEVOS Activevos overview. Available on: `http://www.activevos.com/products/activevos/overview`. Last access: 13/02/2013, 2013.

AHO, A. V.; DAHBURA, A. T.; LEE, D.; UYAR, M. U. Conformance testing methodologies and architectures for osi protocols. chap. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours, Los Alamitos, CA, USA: IEEE Computer Society Press, p. 427–438, 1995.

AMMANN, P.; OFFUTT, J. *Introduction to software testing*. New York, NY, USA: Cambridge University Press, 2008.

ANDREWS, A. A.; OFFUTT, J.; ALEXANDER, R. T. Testing web applications by modeling with FSMs. *Software and System Modeling*, v. 4, n. 3, p. 326–345, 2005a.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, St. Louis, MO, USA: ACM, 2005b, p. 402–411.

ANDRIKOPOULOS, V.; BUCCHIARONE, A.; NITTO, E.; KAZHAMIAKIN, R.; LANE, S.; MAZZA, V.; RICHARDSON, I. Service engineering. In: *Service Research Challenges and Solutions for the Future Internet*, Springer Berlin Heidelberg, p. 271–337, 2010.

APACHE.ORG Apache jUDDI. Available on: `http://ws.apache.org/juddi/index.html`. Last access: 13/02/2013, 2013.

ARCURI, A. Longer is better: On the role of test sequence length in software testing. In: *International Conference on Software Testing, Verification, and Validation (ICST)*, Paris, France: IEEE Computer Society, 2010, p. 469–478.

ASF Apache ode. Available on: `http://ode.apache.org/`. Last access: 13/02/2013, 2010.

ASF Apache maven project. Available on: `http://maven.apache.org`. Last access: 13/02/2013, 2011.

AYDAL, E. G.; PAIGE, R. F.; UTTING, M.; WOODCOCK, J. Putting formal specifications under the magnifying glass: Model-based testing for validation. In: *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, Denver, CO, USA: IEEE Computer Society, 2009, p. 131–140.

BALDONI, M.; BAROGLIO, C.; MARTELLI, A.; PATTI, V.; SCHIFANELLA, C. Verifying the conformance of web services to global interaction protocols: A first step. In: *International Workshop on Web Services and Formal Methods (WS-FM)*, Versailles, France: Springer, 2005, p. 257–271.

BELLI, F.; BUDNIK, C. J. Minimal spanning set for coverage testing of interactive systems. In: *First International Colloquium on Theoretical Aspects and Computing (ICTAC)*, Guiyang, China: Springer Verlag, 2004, p. 220–234.

BELLI, F.; BUDNIK, C. J.; WHITE, L. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification & Reliability*, v. 16, n. 1, p. 3–32, 2006.

BELLI, F.; ENDO, A. T.; LINSCHULTE, M.; SIMAO, A. Model-based testing of web service compositions. In: *The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011)*, Irvine, CA, USA, 2011a, p. 181–192.

BELLI, F.; ENDO, A. T.; LINSCHULTE, M.; SIMAO, A. A holistic approach to model-based testing of web service compositions. *Software: Practice and Experience*, p. n/a–n/a, 2013. Available on: `http://dx.doi.org/10.1002/spe.2161`

BELLI, F.; GÜLER, N.; LINSCHULTE, M. Are longer test sequences always better? - a reliability theoretical analysis. In: *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, Singapore, Singapore, 2010, p. 78–85.

BELLI, F.; GÜLER, N.; LINSCHULTE, M. Does "depth" really matter? on the role of model refinement for testing and reliability. In: *IEEE 35th Annual Computer Software and Applications Conference (COMPSAC)*, Munich, Germany, 2011b, p. 630–639.

BELLI, F.; LINSCHULTE, M. Event-driven modeling and testing of web services. In: *IEEE International Computer Software and Applications Conference (COMPSAC)*, Turku, Finland, 2008, p. 1168–1173.

BELLI, F.; LINSCHULTE, M. Event-driven modeling and testing of real-time web services. *Service Oriented Computing and Applications Journal*, v. 4, n. 1, p. 3–15, 2010.

BENHARREF, A.; DSSOULI, R.; GLITHO, R.; SERHANI, M. A. Towards the testing of composed web services in 3rd generation networks. In: *IFIP International Conference on Testing of Communicating Systems (TESTCOM)*, New York City, USA, 2006, p. 118–133.

BENTAKOUK, L.; POIZAT, P.; ZAÏDI, F. A formal framework for service orchestration testing based on symbolic transition systems. In: *International Conference on Testing of Software and Communication Systems (TESTCOM)*, Eindhoven, the Netherlands: Springer-Verlag, 2009, p. 16–32.

BERGLUND, A.; BOAG, S.; CHAMBERLIN, D.; FERNANDEZ, M. F.; KAY, M.; ROBIE, J.; SIMEON, J. XML path language (XPath) 2.0 (second edition). Available on: `http://www.w3.org/TR/xpath20`. Last access: 13/02/2013, 2010.

BERTOLINO, A.; ANGELIS, G. D.; FRANTZEN, L.; POLINI, A. Model-based generation of testbeds for web services. In: *IFIP International Conference on Testing of Communicating Systems (TESTCOM)*, Tokyo, Japan, 2008, p. 266–282 (*Lecture Notes in Computer Science*, v.5047).

BERTOLINO, A.; FRANTZEN, L.; POLINI, A.; TRETMANS, J. Audition of web services for testing conformance to open specified protocols. In: *Architecting Systems with Trustworthy Components International Seminar*, Dagstuhl Castle, Germany, 2004, p. 1–25.

BINDER, R. V. *Testing object-oriented systems: Models, patterns, and tools*, v. 1. Addison Wesley Longman, Inc., 1999.

BLACKBURN, M.; BUSSER, R.; NAUMAN, A. *Why model-based test automation is different and what you should know to get started.* Technical Report, Software Productivity Consortium, 2004.

BOAG, S.; CHAMBERLIN, D.; FERNANDEZ, M. F.; FLORESCU, D.; ROBIE, J.; SIMEON, J. XQuery 1.0: An XML query language (second edition). Available on: `http://www.w3.org/TR/xquery/`. Last access: 13/02/2013, 2010.

BOUQUET, F.; DEBRICON, S.; LEGEARD, B.; NICOLET, J.-B. Extending the unified process with model-based testing. In: *MoDeVa'06, 3rd Int. Workshop on Model Development, Validation and Verification*, Genova, Italy, 2006, p. 2–15.

BOURHFIR, C.; DSSOULI, R.; E.ABOULHAMID; N.RICO Automatic executable test case generation for EFSM specified protocols. In: *International Workshop on Testing of Communicating Systems (IWTCS'97)*, Cheju Island, Korea, 1997, p. 75–90.

BOZKURT, M.; HARMAN, M.; HASSOUN, Y. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, p. n/a–n/a, 2012. Available on: `http://dx.doi.org/10.1002/stvr.1470`

BRIAND, L. C. A critical analysis of empirical research in software testing. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Madrid, Spain: IEEE Computer Society, 2007, p. 1–8.

BROY, M.; JONSSON, B.; KATOEN, J.-P.; LEUCKER, M.; PRETSCHNER, A. *Model-based testing of reactive systems: advanced lectures*. 1st ed. Springer, 2005.

BRUNO, M.; CANFORA, G.; PENTA, M. D.; ESPOSITO, G.; MAZZA, V. Using test cases as contract to ensure service compliance across releases. In: *International Conference on Service-Oriented Computing (ICSOC)*, Amsterdam, The Netherlands, 2005, p. 87–100.

BUCCHIARONE, A.; MELGRATTI, H.; SEVERONI, F. Testing service composition. In: *8th Argentine Symposium on Software Engineering (ASSE'07)*, Mar del Plata, Argentina, 2007.

BUDD, T. A. Mutation analysis: Ideas, example, problems and prospects. *Computer Program Testing*, p. 129–148, 1981.

CANFORA, G.; DI PENTA, M. SOA: Testing and self-checking. In: *International Workshop on Web Services - Modeling and Testing (WS-MaTE)*, Palermo, Italy, 2006a, p. 3–12.

CANFORA, G.; DI PENTA, M. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, v. 8, n. 2, p. 10–17, 2006b.

CANFORA, G.; DI PENTA, M. Service-oriented architectures testing: A survey. In: *Software Engineering: International Summer Schools (ISSSE)*, Berlin, Heidelberg: Springer-Verlag, 2009, p. 78–105.

CAPELLARI, M. L.; GIMENES, I. M. S.; SIMAO, A.; ENDO, A. T. Towards incremental fsm-based testing of software product lines. In: *XI Simposio Brasileiro de Qualidade de Software (SBQS 2012)*, Fortaleza, Brazil, 2012, p. 9–23.

CAVALLI, A.; CAO, T.-D.; MALLOULI, W.; MARTINS, E.; SADOVYKH, A.; SALVA, S.; ZAIDI, F. Webmov: A dedicated framework for the modelling and testing of web services composition. In: *IEEE International Conference on Web Services (ICWS)*, Miami, FL, USA, 2010, p. 377–384.

CERAMI, E. *Web services essentials*. 1st ed. O'Reilly, 2002.

CHAKRABARTI, S. K.; RODRIQUEZ, R. Connectedness testing of restful web-services. In: *India software engineering conference (ISEC)*, New York, NY, USA: ACM, 2010, p. 143–152.

CHAN, K. S.; BISHOP, J.; STEYN, J.; BARESI, L.; GUINEA, S. Service-oriented computing - icsoc 2007 workshops. chap. A Fault Taxonomy for Web Service Composition, Berlin, Heidelberg: Springer-Verlag, p. 363–375, 2009.

CHAN, W.; CHEUNG, S.; LEUNG, K. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Service Research*, v. 4, n. 2, p. 61–81, 2007.

CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, v. 4, n. 3, p. 178–187, 1978.

CORRADINI, F.; ANGELIS, F.; POLINI, A.; POLZONETTI, A. Improving trust in composite eservices via run-time participants testing. In: *International conference on Electronic Government (EGOV)*, Torino, Italy: Springer-Verlag, 2008, p. 279–290.

CURBERA, F.; DUFTLER, M.; KHALAF, R.; NAGY, W.; MUKHI, N.; WEERAWARANA, S. Unraveling the web services: an introduction to soap, wsdl, and uddi. *Internet Computing*, v. 6, n. 2, p. 86–93, 2002.

DALAL, S. R.; JAIN, A.; KARUNANITHI, N.; LEATON, J. M.; LOTT, C. M.; PATTON, G. C.; HOROWITZ, B. M. Model-based testing in practice. In: *International conference on Software engineering (ICSE)*, Los Angeles, USA: ACM, 1999, p. 285–294.

DE ANGELIS, F.; POLINI, A.; ANGELIS, G. A counter-example testing approach for orchestrated services. In: *Third International Conference on Software Testing, Verification and Validation (ICST)*, Paris, France, 2010, p. 373 –382.

DEMILLO, R. A. *Software testing and evaluation.* The Benjamim/Commings Publishing Company, Inc, 1978.

DEUGD, S. D.; CARROLL, R.; KELLY, K.; MILLETT, B.; RICKER, J. Soda: Service oriented device architecture. *IEEE Pervasive Computing*, v. 5, n. 3, p. 94–96, c3, 2006.

DI NITTO, E.; GHEZZI, C.; METZGER, A.; PAPAZOGLOU, M.; POHL, K. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering Journal*, v. 15, n. 3-4, p. 313–341, 2008.

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering Journal*, v. 10, n. 4, p. 405–435, 2005.

DO, H.; ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, v. 32, n. 9, p. 733 –752, 2006.

DONG, R.; WEI, Z.; LUO, X.; LIU, F. Testing conformance of bpel business process based on model checking. *Journal of Software*, v. 5, n. 9, 2010.

DOROFEEVA, R.; EL-FAKIH, K.; MAAG, S.; R.CAVALLI, A.; YEVTUSHENKO, N. Experimental evaluation of FSM-based testing methods. In: *International Conference on Software Engineering and Formal Methods (SEFM)*, Koblenz, Germany, 2005a, p. 23–32.

DOROFEEVA, R.; EL-FAKIH, K.; MAAG, S.; R.CAVALLI, A.; YEVTUSHENKO, N. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, v. 52, n. 12, p. 1286–1297, 2010.

DOROFEEVA, R.; EL-FAKIH, K.; YEVTUSHENKO, N. An improved conformance testing method. In: *IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, Taipei, Taiwan: Springer, 2005b, p. 204–218 (*Lecture Notes in Computer Science*, v.3731).

DRANIDIS, D.; KOURTESIS, D.; RAMOLLARI, E. Formal verification of web service behavioural conformance through testing. *Annals of Mathematics, Computing & Teleinformatics*, v. 1, n. 5, p. 36–43, 2007.

DRANIDIS, D.; METZGER, A.; KOURTESIS, D. Enabling proactive adaptation through just-in-time testing of conversational services. In: DI NITTO, E.; YAHYAPOUR, R., eds. *Towards a Service-Based Internet*, v. 6481 de *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 63–75, 2010.

DURELLI, V. H. S.; ENDO, A. T.; SIMAO, A.; DELAMARO, M. E. Towards envisaging software testing in a pervasive computing world. In: *XXVI Brazilian Symposium on Software Engineering (SBES 2012) - Special track on Grand Challenges in Softwar-System Engineering*, Natal, Brazil, 2012, p. 201–205.

EL-FAR, I. K.; WHITTAKER, J. A. Model-based software testing. In: *Encyclopedia on Software Engineering*, Wiley, 2001, p. 825–837.

ELER, M. M.; DELAMARO, M. E.; MALDONADO, J. C.; MASIERO, P. C. Built-in structural testing of web services. In: *XXIV Brazilian Symposium on Software Engineering (SBES)*, Salvador, Brazil, 2010, p. 70–79.

ELER, M. M.; ENDO, A. T.; MASIERO, P. C.; DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R.; CHAIM, M. L.; BEDER, D. M. JaBUTiService: A web service for structural testing of java programs. In: *Proceedings of the 33rd Annual IEEE Software Engineering Workshop (SEW 2009)*, Skovde, Sweden, 2009, p. 69–76.

ENDO, A. T.; LINSCHULTE, M.; SIMÃO, A. S.; SOUZA, S. R. S. Event- and coverage-based testing of web services. In: *Workshop on Model-Based Verification & Validation From Research to Practice (MVV) - in conjunction with the Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, Singapore, Singapore, 2010, p. 1–8.

ENDO, A. T.; SILVEIRA, M. B.; RODRIGUES, E. M.; A., S.; OLIVEIRA, F. M.; ZORZO, A. F. *Using models to test web service-oriented applications: an experience report*. Technical Report 67, Pontifícia Universidade Católica do Rio Grande do Sul (FACIM-PUCRS), Porto Alegre, RS, Brazil, 2012.

ENDO, A. T.; SIMÃO, A. S.; SOUZA, S. R. S.; SOUZA, P. S. L. Web services composition testing: a strategy based on structural testing of parallel programs. In: *Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART)*, Windsor, United Kingdom, 2008, p. 3–12.

ENDO, A. T.; SIMAO, A. *Formal testing approaches for service-oriented architectures and web services: a systematic review*. Technical Report 348, Universidade de São Paulo (USP), São Carlos, SP, Brazil, 2010a.

ENDO, A. T.; SIMAO, A. A systematic review on formal testing approaches for web services. In: *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, Natal, Brazil, 2010b, p. 89–98.

ENDO, A. T.; SIMAO, A. Model-based testing of service-oriented applications via state models. In: *IEEE International Conference on Services Computing (SCC)*, Washington, DC, USA, 2011, p. 432–439.

ENDO, A. T.; SIMAO, A. Experimental comparison of test case generation methods for finite state machines. In: *The 8th Workshop on Advances in Model Based Testing (A-MOST 2012)*, Montreal, Canada, 2012a, p. 549–558.

ENDO, A. T.; SIMAO, A. *An experimental study on test suite characteristics, cost, and effectiveness of fsm-based testing methods*. Technical Report 378, Universidade de São Paulo (USP), São Carlos, SP, Brazil, 2012b.

ENDO, A. T.; SIMAO, A. Experiments with FSMs: LabES. 2012c.
Available on: `http://www.labes.icmc.usp.br/~aendo/fsm-experiments`

ENDO, A. T.; SIMAO, A. Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. *Information and Software Technology*, v. 55, n. 6, p. 1045–1062, 2013.
Available on: `http://dx.doi.org/10.1016/j.infsof.2013.01.001`

ERL, T. *Service-oriented architecture: Concepts, technology, and design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

ESCOBEDO, J.; GASTON, C.; LE GALL, P.; CAVALLI, A. Testing web service orchestrators in context: A symbolic approach. In: *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, Pisa, Italy, 2010, p. 257–267.

ESTRELLA, J. C.; ENDO, A. T.; TOYOHARA, R. K. T.; SANTANA, R. H.; SANTANA, M. J.; BRUSCHI, S. M. A performance evaluation study for web services attachments. In: *IEEE International Conference on Web Services*, Los Angeles, CA, USA, 2009, p. 799–806.

EVIWARE soapUI. Available on: `http://www.soapui.org/`. Last access: 13/02/2013, 2012.

FARCHI, E.; HARTMAN, A.; PINTER, S. S. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, v. 41, n. 1, p. 89–110, 2002.

FENG, X.; PARNAS, D. L.; TSE, T.; O'CALLAGHAN, T. A comparison of tabular expression-based testing strategies. *IEEE Transactions on Software Engineering*, v. 37, n. 5, p. 616–634, 2011.

FRANTZEN, L.; LAS NIEVES HUERTA, M.; KISS, Z. G.; WALLET, T. On-the-fly model-based testing of web services with jambition. In: *International Workshop on Web Services and Formal Methods (WS-FM)*, Milan, Italy: Springer-Verlag, 2008, p. 143–157.

FRANTZEN, L.; TRETMANS, J.; VRIES, R. Towards model-based testing of web services. In: POLINI, A., ed. *International Workshop on Web Services - Modeling and Testing – WS-MaTe 2006*, Palermo, Italy, 2006, p. 67–82.

FUJIWARA, S.; BOCHMANN, G. V.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, v. 17, n. 6, p. 591–603, 1991.

GAO, H.; LI, Y. Generating quantitative test cases for probabilistic timed web service composition. In: *IEEE Asia Pacific Services Computing Conference (APSCC)*, Jeju, South Korea, 2011, p. 275–283.

GARCÍA-FANJUL, J.; RIVA, C.; TUYA, J. Generation of conformance test suites for compositions of web services using model checking. In: *Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, Windsor, UK, 2006, p. 127–130.

GILL, A. *Introduction to the theory of finite-state machines*. McGraw-Hill, 1962.

GRIESKAMP, W.; KICILLOF, N.; STOBIE, K.; BRABERMAN, V. A. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, v. 21, n. 1, p. 55–71, 2011.

HARTMAN, A.; KATARA, M.; OLVOVSKY, S. Choosing a test modeling language: a survey. In: *HVC'06: Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, Haifa, Israel, 2007, p. 204–218.

HECKEL, R.; MARIANI, L. Automatic conformance testing of web services. In: *International Conference on Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, Edinburgh, Scotland, 2005, p. 34–48 (*Lecture Notes in Computer Science*,).

HENNIE, F. C. Fault detecting experiments for sequential circuits. In: *Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, 1964, p. 95 –110.

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LÜTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, v. 41, n. 2, p. 1–76, 2009.

HIERONS, R. M.; URAL, H. Generating a checking sequence with a minimum number of reset transitions. *Automated Software Engineering Journal*, v. 17, n. 3, p. 217–250, 2010.

HOARE, C. *Communicating sequential processes*. Prentice Hall International, 2004.
Available on: `http://www.usingcsp.com/`

HOU, S.-S.; ZHANG, L.; LAN, Q.; MEI, H.; SUN, J.-S. Generating effective test sequences for BPEL testing. In: *International Conference on Quality Software (QSIC)*, Jeju, Korea, 2009, p. 331–340.

HUHNS, M. N.; SINGH, M. P. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, v. 9, n. 1, p. 75–81, 2005.

HUMMER, W.; RAZ, O.; SHEHORY, O.; LEITNER, P.; DUSTDAR, S. Test coverage of data-centric dynamic compositions in service-based systems. In: *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, Berlin, Germany, 2011, p. 40–49.

HUMMER, W.; RAZ, O.; SHEHORY, O.; LEITNER, P.; DUSTDAR, S. Testing of data-centric and event-based dynamic service compositions. *Software Testing, Verification and Reliability*, 2013.

IEEE *IEEE standard glossary of Software Engineering terminology*. Standard 620.12, IEEE, 1990.

ILIEVA, S.; MANOVA, D.; MANOVA, I.; BARTOLINI, C.; BERTOLINO, A.; LONETTI, F. An automated approach to robustness testing of BPEL orchestrations. In: *The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011)*, Irvine, CA, USA, 2011, p. 193–203.

IPATE, F. Bounded sequence testing from deterministic finite state machines. *Theorical Computer Science*, v. 411, n. 16-18, p. 1770–1784, 2010.

JIA, Y.; HARMAN, M. Constructing subtle faults using higher order mutation testing. In: *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Beijing, China, 2008, p. 249–258.

JORDAN, D.; EVDEMON, J.; ALVES, A.; ARKIN, A.; ASKARY, S.; BARRETO, C.; BLOCH, B.; CURBERA, F.; FORD, M.; GOLAND, Y.; GUÍZAR, A.; KARTHA, N.; LIU, C. K.; KHALAF, R.; KONIG, D.; MARIN, M.; MEHTA, V.; THATTE, S.; RIJN, D.; YENDLURI, P.; YIU, A. OASIS web services business process execution language (WSBPEL) v2.0. Available on: `http://docs.oasis-open.org/wsbpel/2.0/`. Last access: 13/02/2013, 2007.

JOSUTTIS, N. *SOA in practice: The art of distributed system design*. O'Reilly Media, Inc., 2007.

JUNIT Junit.org resources for test driven development. Available on: `http://junit.sourceforge.net/`. Last access: 13/02/2013, 2011.

JURISTO, N.; MORENO, A. M.; VEGAS, S. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering Journal*, v. 9, n. 1-2, p. 7–44, 2004.

KAMPENES, V. B.; DYBÅ, T.; HANNAY, J. E.; SJØBERG, D. I. K. Systematic review: A systematic review of effect size in software engineering experiments. *Information & Software Technology*, v. 49, n. 11-12, p. 1073–1086, 2007.

KARAM, M.; SAFA, H.; ARTAIL, H. An abstract workflow-based framework for testing composed web services. In: *IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, Amman, Jordan, 2007, p. 901–908.

KATTEPUR, A.; SEN, S.; BAUDRY, B.; BENVENISTE, A.; JARD, C. Pairwise testing of dynamic composite services. In: *The 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Waikiki, Honolulu, HI, USA: ACM, 2011, p. 138–147.

KAVANTZAS, N.; BURDETT, D.; RITZINGER, G.; FLETCHER, T.; LAFON, Y.; BARRETO, C. Web services choreography description language version 1.0. Available on: `http://www.w3.org/TR/ws-cdl-10/`. Last access: 13/02/2013, 2005.

KAZHAMIAKIN, R.; PISTORE, M.; SANTUARI, L. Analysis of communication models in web service compositions. In: *WWW '06: Proceedings of the 15th international conference on World Wide Web*, Edinburgh, Scotland: ACM Press, 2006, p. 267–276.

KEUM, C.; KANG, S.; KO, I.-Y.; BAIK, J.; CHOI, Y.-I. Generating test cases for web services using extended finite state machine. In: *IFIP International Conference on Testing of Communicating Systems (TESTCOM)*, New York, NY, USA: Springer, 2006, p. 103–117.

KITCHENHAM, B. A.; PFLEEGER, S. L.; PICKARD, L. M.; JONES, P. W.; HOAGLIN, D. C.; EMAM, K. E.; ROSENBERG, J. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, v. 28, n. 8, p. 721–734, 2002.

KOURTESIS, D.; RAMOLLARI, E.; DRANIDIS, D.; PARASKAKIS, I. Increased reliability in soa environments through registry-based conformance testing of web services. *Production Planning & Control: The Management of Operations*, v. 21, n. 2, p. 130–144, 2010.

LALLALI, M.; ZAIDI, F.; CAVALLI, A.; HWANG, I. Automatic timed test case generation for web services composition. In: *European Conference on Web Services (ECOWS)*, Dublin, Ireland, 2008, p. 53–62.

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, v. 84, n. 8, p. 1090–1123, 1996.

LEI, Y.; CARVER, R. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, v. 32, n. 6, p. 382 –403, 2006.

LI, L.; CHOU, W.; GUO, W. Control flow analysis and coverage driven testing for web services. In: *IEEE International Conference on Web Services (ICWS)*, Beijing, China, 2008a, p. 473–480.

LI, Z. J.; TAN, H. F.; LIU, H. H.; ZHU, J.; MITSUMORI, N. M. Business-process-driven gray-box soa testing. *IBM Systems Journal*, v. 47, n. 3, p. 457–472, 2008b.

LIU, C.-H.; CHEN, S.-L.; LI, X.-Y. A ws-bpel based structural testing approach for web service compositions. In: *IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, Jhongli, Taiwan, 2008, p. 135–141.

LOHMANN, M.; MARIANI, L.; HECKEL, R. A model-driven approach to discovery, testing and monitoring of web services. In: *Test and Analysis of Web Services*, 2007, p. 173–204.

LUO, G.; PETRENKO, A.; BOCHMANN, G. Selecting test sequences for partially-specified nondeterministic finite state machines. In: *IWPTS '94: 7th IFIP WG 6.1 international workshop on Protocol test systems*, London, UK: Chapman & Hall, Ltd., 1995, p. 95–110.

MA, C.; DU, C.; ZHANG, T.; HU, F.; CAI, X. WSDL-based automated test data generation for web service. In: *International Conference on Computer Science and Software Engineering (CSSE)*, Wuhan, China, 2008, p. 731–737.

MACKENZIE, C. M.; LASKEY, K.; MCCABE, F.; BROWN, P. F.; METZ, R.; HAMILTON, B. A. OASIS reference model for service oriented architecture 1.0. Available on: `http://docs.oasis-open.org/soa-rm/v1.0/`. Last access: 13/02/2013, 2006.

MALDONADO, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. Doctoral Dissertation, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MALDONADO, J. C. Critérios de teste de software: Aspectos teóricos, empíricos e de automatização. ICMC-USP, 1997.

MATHUR, A. P. *Foundations of software testing: Fundamental algorithms and techniques*. Addison-Wesley Professional, 2008.

MCMILLAN, K. L. *Symbolic model checking: an approach to the state explosion problem*. Doctoral Dissertation, Pittsburgh, PA, USA, 1992.

MEI, L.; CHAN, W.; TSE, T. Data flow testing of service-oriented workflow applications. In: *International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008, p. 371–380.

MEI, L.; CHAN, W. K.; TSE, T. H. Data flow testing of service choreography. In: *Symposium on the Foundations of Software Engineering (FSE)*, Amsterdam, The Netherlands, 2009a, p. 151–160.

MEI, L.; CHAN, W. K.; TSE, T. H.; KUO, F.-C. An empirical study of the use of frankl-weyuker data flow testing criteria to test bpel web services. In: *IEEE International Computer Software and Applications Conference (COMPSAC)*, Seattle,Washington, USA, 2009b, p. 81–88.

MEI, L.; ZHANG, Z.; CHAN, W. K.; TSE, T. H. Test case prioritization for regression testing of service-oriented business applications. In: *International Conference on World Wide Web (WWW)*, Madrid, Spain: ACM, 2009c, p. 901–910.

MILANOVIC, N.; MALEK, M. Current solutions for web service composition. *IEEE Internet Computing*, v. 8, n. 6, p. 51–59, 2004.

MOORE, E. F. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematics Series*, , n. 34, p. 129–153, 1956.

MORALES, G.; MAAG, S.; CAVALLI, A.; MALLOULI, W.; OCA, E.; WEHBI, B. Timed extended invariants for the passive testing of web services. In: *IEEE International Conference on Web Services (ICWS)*, Miami, FL, USA, 2010, p. 592 –599.

MOUCHAWRAB, S.; BRIAND, L. C.; LABICHE, Y. Assessing, comparing, and combining statechart- based testing and structural testing: An experiment. In: *ESEM '07: Proceedings* *of the First International Symposium on Empirical Software Engineering and Measurement*, Washington, DC, USA: IEEE Computer Society, 2007, p. 41–50.

MULESOFT Mule ESB: Open source ESB and integration platform. Available on: `http://www.mulesoft.org/`. Last access: 13/02/2013, 2012.

MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. *The art of software testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

NETBEANS.ORG Netbeans SOA project home. Available on: `http://soa.netbeans.org/`. Last access: 13/02/2013, 2009.

NEWCOMER, E. *Understanding web services: XML, WSDL, SOAP and UDDI*. 1st ed. Addison Wesley, 2002.

NGUYEN, H. N.; POIZAT, P.; ZAÏDI, F. Passive conformance testing of service choreographies. In: *the 27th Annual ACM Symposium on Applied Computing (SAC 2012)*, Trento, Italy: ACM, 2012, p. 1528–1535.

NI, Y.; HOU, S.; ZHANG, L.; ZHU, J.; LI, Z.; LAN, Q.; MEI, H.; SUN, J. Effective message-sequence generation for testing bpel programs. *IEEE Transactions on Services Computing*, v. PP, n. 99, p. n/a–n/a, 2011.

OASIS UDDI specifications tc. Available on: `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`. Last access: 13/02/2013, 2004.

OASIS Oasis web services security (wss) tc. Available on: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`. Last access: 13/02/2013, 2006.

OGF Open grid forum – web services agreement specification (ws-agreement). Available on: `http://www.ogf.org/documents/GFD.107.pdf`. Last access: 13/02/2013, 2007.

OPEN SERVICE ACCESS Parlay x web services. Available on: `http://www.3gpp.org/ftp/Specs/html-info/29199-01.htm`. Last access: 13/02/2013, 2009.

ORACLE Oracle BPEL process manager. Available on: `http://www.oracle.com/technetwork/middleware/bpel/overview/index.html`. Last access: 13/02/2013, 2013.

PALACIOS, M.; GARCÍA-FANJUL, J.; TUYA, J. Testing in service oriented architectures with dynamic binding: A mapping study. *Information and Software Technology*, v. 53, n. 3, p. 171–189, 2011.

PAPAZOGLOU, M. P.; HEUVEL, W.-J. Service oriented architectures: approaches, technologies and research issues. *The International Journal on Very Large Databases (VLDB)*, v. 16, n. 3, p. 389–415, 2007.

PARADKAR, A.; SINHA, A.; WILLIAMS, C.; JOHNSON, R.; OUTTERSON, S.; SHRIVER, C.; LIANG, C. Automated functional conformance test generation for semantic web services. In: *IEEE International Conference on Web Services (ICWS)*, Salt Lake City, Utah, USA, 2007, p. 110–117.

PARK, Y.; JUNG, W.; LEE, B.; WU, C. Automatic discovery of web services based on dynamic black-box testing. In: *IEEE International Computer Software and Applications Conference (COMPSAC)*, Seattle,Washington, USA, 2009, p. 107–114.

PEDROSA, L. L. C.; MOURA, A. V. Incremental testing of finite state machines. *Software Testing, Verification and Reliability*, p. n/a–n/a, 2012.

PELTZ, C. Web services orchestration and choreography. *Computer*, v. 36, p. 46 – 52, 2003.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: *The 12th international conference on Evaluation and Assessment in Software Engineering (EASE)*, Bari, Italy, 2008, p. 68–77.

PRESSMAN, R. S. *Software engineering: A practitioner's approach*. 6th ed. McGraw-Hill, 2005.

PRETSCHNER, A.; PHILIPPS, J. Methodological issues in model-based testing. In: *Model-Based Testing of Reactive Systems*, Lecture Notes in Computer Science, 2004, p. 281–291 (*Lecture Notes in Computer Science*,).

PRETSCHNER, A.; PRENNINGER, W.; WAGNER, S.; KÜHNEL, C.; BAUMGARTNER, M.; SOSTAWA, B.; ZÖLCH, R.; STAUNER, T. One evaluation of model-based testing and its automation. In: *The 27th International Conference on Software Engineering (ICSE)*, St Louis, USA: ACM, 2005, p. 392–401.

QUALIPSO Qualipso factory - next generation forge. Available on: `http://qualipso.gforge.inria.fr`. Last access: 13/02/2013, 2010.

RABHI, I. Robustness testing of web services composition. In: *IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on High Performance Computing and Communication*, Liverpool, United Kingdom, 2012, p. 631–638.

RAMAMOORTHY, C.; BASTANI, F. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering*, v. 8, n. 4, p. 354–371, 1982.

RAMOLLARI, E.; KOURTESIS, D.; DRANIDIS, D.; SIMONS, A. Leveraging semantic web service descriptions for validation by automated functional testing. In: *European Semantic Web Conference (ESWC)*, Heraklion, Greece, 2009, p. 593–607.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985.

REMY, S. L.; BLAKE, M. B. Distributed service-oriented robotics. *IEEE Internet Computing*, v. 15, n. 2, p. 70–74, 2011.

RUSLI, H. M.; PUTEH, M.; IBRAHIM, S.; TABATABAEI, S. G. H. A comparative evaluation of state-of-the-art web service composition testing approaches. In: *Proceedings of the 6th International Workshop on Automation of Software Test (AST)*, Waikiki, Honolulu, Hawaii, USA: ACM, 2011, p. 29–35.

RUSSELL, S.; NORVIG, P. *Artificial intelligence: A modern approach*, chap. Constraints Satisfaction Problems. 2nd edition ed Prentice-Hall, Englewood Cliffs, NJ, p. 137–160, 2003.

RUTH, M.; OH, S.; LOUP, A.; HORTON, B.; GALLET, O.; MATA, M.; TU, S. Towards automatic regression test selection for web services. In: *IEEE International Computer Software and Applications Conference (COMPSAC)*, Beijing, China, 2007, p. 729–736.

SCHMIDT, M.-T.; HUTCHISON, B.; LAMBROS, P.; PHIPPEN, R. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, v. 44, p. 781–797, 2005.

SHARMA, M.; CHANDRA B., S. Automatic generation of test suites from decision table - theory and implementation. In: *The 2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, Nice, France: IEEE Computer Society, 2010, p. 459–464.

SHEN, L.; LI, F.; REN, S.; MU, Y. Dynamic composition of web service based on coordination model. In: CHANG, K.-C.; WANG, W.; CHEN, L.; ELLIS, C.; HSU, C.-H.; TSOI, A.; WANG, H., eds. *Advances in Web and Network Technologies, and Information Management*, v. 4537 de *Lecture Notes in Computer Science*, Vienna, Austria: Springer Berlin Heidelberg, p. 317–327, 2007.

SIMAO, A.; ENDO, A. T. Teste baseado em modelos. In: *Minicurso no Congresso Brasileiro de Software: Teoria e Pratica*, available on: `http://wiki.dcc.ufba.br/CBSOFT/ShortCourseMC10Pt`. Last access: 13/02/2013, 2010.

SIMAO, A.; PETRENKO, A. Checking completeness of tests for finite state machines. *IEEE Transactions on Computers*, v. 59, p. 1023–1032, 2010a.

SIMAO, A.; PETRENKO, A. Fault coverage-driven incremental test generation. *Computer Journal*, v. 53, p. 1508–1522, 2010b.

SIMAO, A.; PETRENKO, A.; MALDONADO, J. C. Comparing finite state machine test coverage criteria. *IET Software*, v. 3, n. 2, p. 91–105, 2009a.

SIMAO, A.; PETRENKO, A.; YEVTUSHENKO, N. Generating reduced tests for FSMs with extra states. In: *IFIP WG 6.1 International Conference on Testing of Software and Communication Systems (TESTCOM)*, Eindhoven, The Netherlands, 2009b, p. 129–145.

SINHA, A.; SMIDTS, C. HOTTest: A model-based test design technique for enhanced testing of domain-specific applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 15, n. 3, p. 242–278, 2006.

SOURCEFORGE.NET Eclipse metrics 1.3.6. Available on: `http://metrics.sourceforge.net`. Last access: 13/02/2013, 2005.

SOURCEFORGE.NET WS-CDL eclipse (with examples). Available on: `http://sourceforge.net/projects/wscdl-eclipse/`. Last access: 13/02/2013, 2012.

TRETMANS, J. *A formal approach to conformance testing*. Doctoral Dissertation, University of Twente, Enschede, Netherlands, 1992.

TSAI, W.; WEI, X.; CHEN, Y.; PAUL, R. A robust testing framework for verifying web services by completeness and consistency analysis. In: *IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, Beijing, China, 2005a, p. 151–158.

TSAI, W.; WEI, X.; CHEN, Y.; XIAO, B.; PAUL, R.; HUANG, H. Developing and assuring trustworthy web services. In: *International Symposium on Autonomous Decentralized Systems (ISADS)*, Chengdu, China, 2005b, p. 43–50.

TSAI, W. T.; GAO, J.; WEI, X.; CHEN, Y. Testability of software in service-oriented architecture. In: *The 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Chicago, USA: IEEE Computer Society, 2006, p. 163–170.

TSAI, W.-T.; WEI, X.; CHEN, Y.; PAUL, R.; XIAO, B. Swiss cheese test case generation for web services testing. *Transactions on Information and Systems*, v. E88-D, n. 12, p. 2691–2698, 2005c.

UTTING, M.; LEGEARD, B. *Practical model-based testing: A tools approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. *A taxonomy of model-based testing*. Technical Report, Hamilton, New Zealand, 2006.

VASILEVSKII, M. P. Failure diagnosis of automata. *Cybernetics and Systems Analysis*, v. 9, p. 653–665, 1973.

VINCENZI, A. M. R. *Orientação a objeto: Definição e análise de recursos de teste e validação*. Doctoral Dissertation, ICMC/USP, São Carlos, SP, 2004.

VUONG, S. T.; CHAN, W. Y. L.; ITO, M. R. The uiov-method for protocol test sequence generation. In: *IFIP Int. Workshop Protocol Test Systems*, Berlin, Germany, 1989.

W3C XSL transformations (xslt). Available on: `http://www.w3.org/TR/xslt`. Last access: 13/02/2013, 1999.

W3C Web services description language (WSDL). Available on: `http://www.w3.org/TR/wsdl`. Last access: 13/02/2013, 2001.

W3C Web services activity statement. Available on: `http://www.w3.org/2002/ws/Activity`. Last access: 13/02/2013, 2002.

W3C Extensible markup language (XML). Available on: `http://www.w3.org/XML/`. Last access: 13/02/2013, 2003.

W3C OWL-S: Semantic markup for web services. Available on: `http://www.w3.org/Submission/OWL-S`. Last access: 13/02/2013, 2004a.

W3C SOAP specifications. Available on: `http://www.w3.org/TR/soap/`. Last access: 13/02/2013, 2004b.

W3C Web services addressing (ws-addressing). Available on: `http://www.w3.org/Submission/ws-addressing/`. Last access: 13/02/2013, 2004c.

W3C Xml schema. Available on: `http://www.w3.org/XML/Schema`. Last access: 13/02/2013, 2004d.

W3C Web services description language (WSDL) version 2.0. Available on: `http://www.w3.org/TR/wsdl20`. Last access: 13/02/2013, 2007.

WIECZOREK, S.; KOZYURA, V.; ROTH, A.; LEUSCHEL, M.; BENDISPOSTO, J.; PLAGGE, D.; SCHIEFERDECKER, I. Applying model checking to generate model-based integration tests from choreography models. In: *International Conference on Testing of Software and Communication Systems (TESTCOM)*, Eindhoven, The Netherlands, 2009, p. 179–194.

WIECZOREK, S.; STEFANESCU, A.; ROTH, A. Model-driven service integration testing - a case study. In: *Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, Porto, Portugal, 2010, p. 292–297.

YAN, J.; LI, Z.; YUAN, Y.; SUN, W.; ZHANG, J. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In: *International Symposium on Software Reliability Engineering (ISSRE)*, Raleigh, North Carolina, USA, 2006, p. 75–84.

YUAN, X.; COHEN, M. B.; MEMON, A. M. Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, v. 37, n. 4, p. 559–574, 2011.

ZANDER, J.; SCHIEFERDECKER, I.; MOSTERMAN, P. *Model-based testing for embedded systems*. Computational Analysis, Synthesis, and Design of Dynamic Systems. Taylor & Francis, 2011.

ZHENG, Y.; ZHOU, J.; KRAUSE, P. An automatic test case generation framework for web services. *Journal of Software*, v. 2, n. 3, p. 64–77, 2007.

ZHOU, L.; PING, J.; XIAO, H.; WANG, Z.; PU, G.; DING, Z. Automatically testing web services choreography with assertions. In: *The 12th international conference on Formal engineering methods and software engineering (ICFEM)*, Shanghai, China, 2010, p. 138–154.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, v. 29, n. 4, p. 366–427, 1997.

Appendix A

Algorithms

This appendix shows the formal descriptions of algorithms proposed in Chapter 5. Table A.1 shows notations used in the following algorithms.

Table A.1: Notations used in the algorithms.

Symbol	*Meaning*	*Example*
$\langle\,\rangle$	the empty sequence	
$\langle a\rangle$	the sequence containing only a	
$\langle a,b,c\rangle$	the sequence with three events, a then b, then c	
$\alpha(s)$	the first event of sequence s	$\alpha(\langle a,b,c\rangle)=a$
$\omega(s)$	the last event of sequence s	$\omega(\langle a,b,c\rangle)=c$
$s[i]$	the ith event of sequence s	$\langle a,b,c\rangle\,[2]=b$
$s[i..j]$	the sequence from i to j	$\langle a,b,c\rangle\,[1..2]=\langle a,b\rangle$
$s\|\|t$	sequence s in parallel to sequence t	
$s\oplus t$	concatenate s and t	$\langle a,b\rangle\oplus\langle c\rangle=\langle a,b,c\rangle$
$N^{+}(v)$	successors of a vertex v in an ESG4WSC/ESG4WS	
$N^{-}(v)$	predecessors of a vertex v in an ESG4WSC/ESG4WS	

Algorithm 1 gives a formal description for the algorithm to generate CESs that cover all event pairs (positive test cases) from an ESG4WSC. This algorithm is described in Section 5.3.2.

Algorithm 1: generateCESs()

```
function : generateCESs()
input    : an ESG4WSC
output   : CES
1  foreach $re \in V_{refined}$ do
2      $resCES \leftarrow \emptyset$;
3      foreach $esg \in re$ do
4          $CES = generateCESs(esg)$;
5          if $resCES$==$\emptyset$ then
6              $resCES = resCES \cup \{(re \times CES)\}$;
7          else
8              foreach $ces_1 \in \{ces|(re, ces) \in resCES\}$ do
9                  foreach $ces_2 \in CES$ do
10                     $resCES = resCES \cup \{(re, (ces_1||ces_2))\}$;
11                 $resCES = resCES \setminus \{(re, ces_1)\}$;
12     if $f(re) \neq \varepsilon$ then
13         DT$_{seq} = f(re)$;
14         foreach $ces \in \{ces|(re, ces) \in resCES\}$ do
15             $v = getAllowedSuccessor(ces, DT_{seq})$;
16             $E := E \cup \{(re, v)\}$;
               // store a Mapping,i.e.,$Map \subseteq E \times ces$
17             $Map := Map \cup \{((re, v), ces)\}$;
18             $resCES := resCES \setminus \{(re, ces)\}$;
   // add multiple edges for each dataset to be tested
19 foreach $DT_{input,public} \in V$ do
20     foreach $a \in A$ do
21         $E := E \setminus \{(DT_{input}, a)\}$;
22     foreach $(C_{true}, C_{false}, E_x) \in R$ do
23         $E := E \cup \{(DT_{input}, E_x)\}$;
24 $CES = solveCPP(ESG4WSC)$;
25 foreach $ces \in CES$ do
26     for $i = 1$ to $\#ces$ do
27         if $ces[i] \in V_{refined}$ then
28             if $|\{ces|((ces[i], ces[i+1]), ces) \in Map]\}| > 0$ then
29                 $new = es$ with $es \in \{ces|((ces[i], ces[i+1]), ces) \in Map\}$;
30                 $Map = Map \setminus \{((ces[i], ces[i+1]), ces)\}$;
31                 $ces = ces[1..(i-1)] \oplus new \oplus ces[(i+1)..\#ces]$;
32             else if $|\{ces|(ces[i], ces) \in resCES\}| > 0$ then
33                 $new = es$ with $es \in \{ces|(ces[i], ces) \in resCES\}$;
34                 $resCES = resCES \setminus \{(ces[i], ces)\}$;
35                 $ces = ces[1..(i-1)] \oplus new \oplus ces[(i+1)..\#ces]$;
36 return $CES$;
```

Algorithm 2 gives a formal description for the algorithm that generates a PES to reach a given event. Examples of PESs generated using this algorithm can be found in Sections 5.4.1 and 5.4.2.

Algorithm 2: Algorithm to generate a partial event sequence that covers an event e_i.

```
function : generatePES()
input    : an ESG4WSC, a non-refining event $e_i$
output   : PES

1  $re = \epsilon$;
2  if $e_i \in V$ then
3      $e_k = e_i$;
4  else
5      Select $re \in V_{refined}$ such that $e_i$ is within $re$;
6      $e_k = re$;
7  $pes = getShortestPath(\Xi, e_k)$;
   // solve the refined events in pes
8  foreach $re_i \in \{re_i \mid re_i \in V_{refined} \setminus \{re\}$ and $re_i \in pes\}$ do
       // this procedure also handles DTs
9      $pes = solveCESforRefinedEvent(pes, re_i)$;
10 if $re \neq \epsilon$ then
11     $par = \{\}$;
12     foreach $esg4wsc \in re$ do
13         if $e_i \in esg4wsc$ then
               // recursive call
14             $pes_i = generatePES(esg4wsc, e_i)$;
15         else
16             $pes_i = getShortestCES(esg4wsc)$;
17         $par = par||pes_i$;
18     $pes = replace(pes, re, par)$;
19 return $pes$;
```

Algorithm 3 gives a formal description for the algorithm that translates an ESG4WSC to an ESG4WS and produces faulty event pairs from the latter. An example of this transformation and faulty edges can be found in Section 5.4.1.

Algorithm 3: Algorithm to transform an ESG4WSC to an ESG4WS and obtain the faulty edges.

```
function : transform()
input    : an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$
output   : an $ESG4WS = (V', E', \Xi', \Gamma')$, faulty edges FE

$ESG4WS = (V', E', \Xi', \Gamma')$;
foreach $v \in V$ do
    if ($v \notin V_{req}$ AND $v \notin V_{resp}$) OR $v$ is private then
        foreach $pre \in N^{-}(v)$ do //predecessors of v
            foreach $post \in N^{+}(v)$ do //successors of v
                if $(pre, post) \notin E$ then
                    $E := E \cup \{(pre, post)\}$;
                $E := E \setminus \{(v, post)\}$;
            $E := E \setminus \{(pre, v)\}$;
        $V := V \setminus v$;
        if $v \in \Xi$ then $\Xi := \Xi \setminus \{v\}$;
        if $v \in \Gamma$ then $\Gamma := \Gamma \setminus \{v\}$;
$V' := V; E' := E; \Xi' := \Xi; \Gamma' := \Gamma$;
$FE := \{\}$;
foreach $v_{req} \in V'_{req}$ do
    foreach $v_{resp} \in V'_{resp}$ do
        if $(v_{resp}, v_{req}) \notin E'$ then
            $FE := FE \cup \{(v_{resp}, v_{req})\}$;
    foreach $v_{req2} \in V'_{req}$ do
        if $v_{req2} \neq v_{req}$ AND $(v_{req2}, v_{req}) \notin E'$ then
            $FE := FE \cup \{(v_{req2}, v_{req})\}$;
    if $v_{req} \notin \Xi'$ then
        $FE := FE \cup \{([, v_{req})\}$;
return ESG4WS, FE;
```

Algorithm 4 gives a formal description for the algorithm to generate PubFESs from an ESG4WSC. This algorithm is described along with an example in Section 5.4.1.

Algorithm 4: Algorithm to generate PubFESs.

```
function : generatePubFESs()
input    : an ESG4WSC, faulty edges FE
output   : the test suite PubFES

1 PubFES = {};
2 foreach $(e_i, e_j) \in FE$ do
3     $pes_i = generatePES(\text{ESG4WSC}, e_i)$;
4     $pes_i = pes_i \oplus e_j$;
5     PubFES = PubFES $\cup\{(pes_i; \digamma)\}$;
6 return PubFES;
```

Algorithm 5 gives a formal description for the algorithm to generate PriFESs from an ESG4WSC. This algorithm is described along with an example in Section 5.4.2.

Algorithm 5: Algorithm to generate PriFESs.

```
function : generatePriFESs()
input    : an ESG4WSC, set of sensitive events $s$
output   : the test suite PriFES

1  PriFES = {};
2  Let $V_{REQ}$ be the union of sets $V_{req}$ for the ESG4WSC and their refining ESG4WSCs;
3  Let $V_{RESP}$ be the union of sets $V_{resp}$ for the ESG4WSC and their refining ESG4WSCs;
4  foreach private event $e_i \in V_{REQ}$ do
5      $pes_i = generatePES(\text{ESG4WSC}, e_i)$;
6      $fes_1 = (pes_i; \digamma_{\text{NR}}; s)$;
7      $fes_2 = (pes_i; \digamma_{\text{MS}}; s)$;
8      $fes_3 = (pes_i; \digamma_{\text{UF}}; s)$;
9      PriFES = PriFES $\cup\{fes_1, fes_2, fes_3\}$;
10 foreach private event $e_j \in V_{RESP}$ do
11     $pes_j = generatePES(\text{ESG4WSC}, e_j)$;
12     $fes_1 = (pes_j; \digamma_{\text{LR}}; s)$;
13     $fes_2 = (pes_j; \digamma_{\text{WSc}}; s)$;
14     $fes_3 = (pes_j; \digamma_{\text{WSy}}; s)$;
15     $fes_4 = (pes_j; \digamma_{\text{WD}}; s)$;
16     PriFES = PriFES $\cup\{fes_1, fes_2, fes_3, fes_4\}$;
17 return PriFES;
```