Universidade Estadual de Campinas
Instituto de Computação

Lucas Carvalho Leal

Self-adaptive model-based online testing for dynamic SOA

Sistema de teste auto-adaptativo baseado em modelo para SOA dinâmico

CAMPINAS
2017

Lucas Carvalho Leal

Self-adaptive model-based online testing for dynamic SOA

Sistema de teste auto-adaptativo baseado em modelo para SOA dinâmico

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientadora: Profa. Dra. Eliane Martins
Co-supervisor/Coorientador: Dr. Andrea Ceccarelli

Este exemplar corresponde à versão final da Dissertação defendida por Lucas Carvalho Leal e orientada pela Profa. Dra. Eliane Martins.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CAPES

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

L473s

Leal, Lucas Carvalho, 1988-
Self-adaptive model-based online testing for dynamic SOA / Lucas Carvalho Leal. – Campinas, SP : [s.n.], 2017.

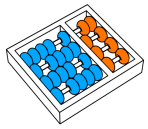
Orientador: Eliane Martins.
Coorientador: Andrea Ceccarelli.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Teste baseado em modelos. 3. Sistemas de computação adaptativos. 4. Arquitetura orientada a serviços (Computação). I. Martins, Eliane, 1955-. II. Ceccarelli, Andrea. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Sistema de teste auto-adaptativo baseado em modelo para SOA dinâmico
Palavras-chave em inglês:
Software engineering
Model-based testing
Adaptive computing systems
Service-oriented architecture (Computer science)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Leonardo Montecchi
Baldoino Fonseca dos Santos Neto
Data de defesa: 14-12-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Lucas Carvalho Leal

Self-adaptive model-based online testing for dynamic SOA

Sistema de teste auto-adaptativo baseado em modelo para SOA dinâmico

Banca Examinadora:

- Profa. Dra. Eliane Martins
  Instituto de computação - UNICAMP
- Prof. Dr. Baldoíno Fonseca dos Santos Neto Membro
  Instituto de computação - UFAL
- Prof. Dr. Leonardo Montecchi
  Instituto de computação - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 14 de dezembro de 2017

Agradecimentos

Agradeço aos meus orientadores, minha familia e namorada, meus amigos, aos funcionários do IC, e mais imporante, ao professor de física da Unicamp, que eu conheci durante meu estágio na Procter & Gamble, que falou alguma coisa coisa em alemão enquanto nos despediamos que eu nunca entendi (mas de alguma maneira eu sei que era sobre fugir daquele lugar).

Thanks to my advisors, my family and girlfriend, my friends, the Unicamp's IC staff, and most important, the Unicamp's physics professor, that I've met during my internship days at Proctor & Gamble, that said something in german during his farewell wish, which I never really understood (but somehow I knew that it was about running away from that place).

Resumo

Arquitetura orientada a serviços (SOA) é um padrão de design popular para implementação de serviços web devido à interoperabilidade, escalabilidade e reuso de soluções de software que promove. Os serviços que usam essa arquitetura precisam operar em um ambiente altamente dinâmico, entretanto quanto mais a complexidade desses serviços cresce menos os métodos tradicionais de validação se mostram viáveis.

Aplicações baseadas em arquitetura orientada a serviços podem evoluir e mudar durante a execução. Por conta disso testes offline não asseguram completamente o comportamento correto de um sistema em tempo de execução. Por essa razão, a necessidade de tecnicas diferentes para validar o comportamento adequado de uma aplicação SOA durante o seu ciclo de vida são necessárias, por isso testes online executados durante o funcionamento serão usados nesse projeto.

O objetivo do projeto é de aplicar técnicas de testes baseados em modelos para gerar e executar casos de testes relevantes em aplicações SOA durante seu tempo de execução. Para alcançar esse objetivo uma estrura de teste online autoadaptativa baseada em modelos foi idealizada.

Testes baseados em modelos podem ser gerados de maneira offline ou online. Nos testes offline, os casos de teste são gerados antes do sistema entrar em execução. Já nos testes online, os casos de teste são gerados e aplicados concomitantemente, e as saídas produzidas pela aplicação em teste definem o próximo passo a ser realizado. Quando uma evolução é detectada em um serviço monitorado uma atualização no modelo da aplicação alvo é executada, seguido pela geração e execução de casos de testes online.

Mais precisamente, quatro componentes foram integrados em um circuito autoadaptativo: um serviço de monitoramento, um serviço de criação de modelos, um serviço de geração de casos de teste baseado em modelos e um serviço de teste. As caracteristicas da estrutura de teste foram testadas em três cenários que foram executados em uma aplicação SOA orquestrada por BPEL, chamada jSeduite.

Este trabalho é um esforço para entender as restrições e limitações de teste de software para aplicações SOA, e apresenta análises e soluções para alguns dos problemas encontrados durante a pesquisa.

Abstract

Service Oriented Architecture (SOA) is a popular design pattern to build web services because of the interoperability, scalability, and reuse of software solutions that it promotes. The services using this architecture need to operate in a highly dynamic environment, but as the complexity of these services grows, traditional validation processes become less feasible. SOA applications can evolve and change during their execution, and offline tests do not completely assure the correct behavior of the system during its execution. Therefore there is a need of techniques to validate the proper behaviour of SOA applications during the SOA lifecycle. Because of that, in this project online testing will be used.

The project goal is to employ model-based testing techniques to generate and execute relevant test cases to SOA applications during runtime. In order to achieve this goal a self-adaptive model-based online testing framework was designed.

Tests based on models can be generated offline and online. Offline test are generated before the system execution. Online tests are generated and performed concomitantly, and the output produced by the application under test defines the next step to be performed. when our solution detects that a monitored service evolves, the model of the target service is updated, and online test case generation and execution is performed.

More specifically, four components were integrated in a self-adaptive loop: a monitoring service, a model generator service, a model based testing service and a testing platform. The testing framework had its features tested in three scenarios that were performed in a SOA application orchestrated by BPEL, called jSeduite.

This work is an effort to understand the constraints and limitations of the software testing on SOA applications, and present analysis and solutions to some of the problems found during the research.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Context and motivation

The definition of service-oriented architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains [30]. As a design pattern to build web services, SOA objectives are to promote interoperability, scalability, and reuse of software solutions. Usually services designed following SOA guidelines work with an interface that enables the communication of the services with custumers. Services work under agreements and contracts that make clear what one can or can't perform and how it should be requested [37][32]. The arising of services using this architecture created the possibility for more complex and distributed systems to bloom. As these systems grow in number and become part of more critical systems of society, assuring the quality of the component services became a necessity.

Service-oriented applications are composed progressively by third-party services available on the Internet. Thus, solutions to assure trust in such services, despite their evolutionary aspect, can be useful. As these services are not under control of the users, changes on them sometimes cannot be predicted. Despite third-party services being tested during the design and implementation stages of an SOA application, the unpredictability of the updates and the unavailability of the services, create the need to monitor and test SOA applications during runtime in order to detect possible flaws [19].

Model-based testing could bring adaptability and agility to test creation, more specifically to aid in situations where software development evolves fast and the requirements of the system under development are up to changes [2]. By applying online tests generated by a model, a SOA application is able to execute tests on the services after its deployment on production (some tests are not quickly performed). Such tests could be generated aiming for different goals inside the SOA application.

Online test is a concept that emerged around the idea of monitoring a system's behavior during runtime, using dynamic analysis and self-test techniques [5]. Online testing means that the composition or the individual services of a SOA application are systematically tested in parallel to its normal use and operation.

SOA is commonly used as an enterprise solution, it's a way to cut development and maintenance costs [39]. The use of mechanisms for execution of online tests over third-party services raises the confidence of the SOA application, avoiding problems caused by

hardware or software in both ends and protecting users from possible consequences of unexpected problems.

1.2 Objectives

The objective of this research is to employ model based testing techniques to generate and execute relevant test cases to service oriented architecture applications (SOA applications) during runtime.

As the use of SOA architecture becomes popular among web services, offline tests performed before deployment do not assure the proper behavior of a service during runtime [12]. Online tests are often avoided because of its intrusiveness. Therefore the use of methods to perform tests with minimum interference on the systems under test (SUT) are necessary.

Online tests can be used to perform procedures for checking if a SOA application meets requirements and specification during runtime, as well as being used to determine whether a service complies with the requirements in case of detected modifications in its proper behavior. This way, allowing the detection of problems before they affect the business process and the integrity of the service, triggering safety measures and reducing the time needed to stabilize the application.

This document chapters are organized as following: the second chapter is a quick review about software testing and model based testing; the third chapter contains a more profound review about test creation with UML State Machines; the fourth chapter presents the basic fundamentals of service oriented architecture; the fifth chapter is related to SOA testing and some related works; the sixth chapter present the SAMBA framework description and implementation; the seventh chapter is the experimental evaluation and test results of the framework; the eighth chapter presents the conclusions of the project.

Chapter 2

About software testing

2.1 Software testing

Software testing is the process of evaluating a software artifact, in order to assure that it meets the requirements provided by its stakeholders. Test objectives are: failure detection, that is observe differences between the behavior of a system under test (SUT) and the intended implementation behavior, defined by its requirements [44].

Software testing goes beyond the previous short description. Test gathers a variety of activities, actors, techniques and many complex challenges. The complexity of software applications and its integration with crucial systems of society is growing and spreading fast, ensuring that these systems behave according to what they were designed is extremely important, difficult and expensive [5].

A test case could be defined in software engineering as a set of inputs, conditions, and expected outputs, required to achieve a specific objective, which a tester will use to validate the behavior of a software artifact [5][29][1]. Since models are an abstract description of a system, the test cases generated from it are called abstract test cases.

2.1.1 Testing techniques

Test cases can be executed with different testing techniques [5]. Usually they are distinguished between Black-Box (functional tests) and White-Box (structural tests). Black-Box test is the technique to develop test cases without information about the source code of the SUT, usually focusing on the interface, giving inputs and validating outputs. White-Box method is the investigation of the internal logic and structure of the SUT. Test cases are more complex to be generated and can detect more specific defects. Sometimes it is possible to classify tests as Grey-Box, when tests are developed with limited knowledge of SUT's source code, a situation between Black-Box and White-Box, enabling a bigger variety of test cases compared with the black-box method [10].

During the life cycle of a software artifact, it is submitted to many tests, each one of them targeting a different objective. Following the taxonomy presented by Luo et al. [25] and Canfora et al. [9], depending on the level of the development phase and nature software tests can be classified as:

- Unit testing - the lowest level of software testing. It covers the basic units of a software, e.g. class methods.
- Integration Testing - it is when two or more tested units are combined. The test aims at the interfaces between the components.
- System testing - Aims at the quality of the entire system. System test is in general based on the functional/requirement specification. Non-functional quality attributes, such as reliability, security, and maintainability, can also be included.
- Acceptance testing - The purpose is rather to give confidence that the system is working than to find errors. Usually done when the system is delivered from the developers to the customers.
- Regression testing - Tests which are executed to guarantee that some features already implemented were not affected or lost during the evolution process.

2.1.2 Testing process

The testing process can be, in short, composed of three activities: creation, execution, and validation [26].

Test creation can be done manually or automatically. In the first method, the test cases are created by software engineers, the quality of these tests depends on the experience of the testing team and its knowledge on the SUT. Automatic test case generation is the process of extraction of test cases from documentation, requirements or model representation of the behaviour of the SUT. Test execution and validation is usually either online or offline. Generated test cases may be executed manually or automatically.

2.2 Model-based testing

Model-based testing (MBT) is a variant of Black-Box testing. It is an application of model-based design, that relies on models that reflects the behavior of a system under test (SUT) and/or its environment. Its objective is to generate test cases from behavioral models or from models combination, and execute them on the system on which the models were based [2].

The use of models for generation of test cases is supported by the fact that, manual test case generation strongly depends on the experience of the test engineers and their understanding about the behavior of the system and its environment.

Utting et al. proposes a generecic process for MBT, as the following figure 2.1 shows.

The basic process of MBT is: the automatic generation of abstract tests from an abstract model of the SUT; the generation of concrete tests cases from abstract tests; and the manual or automated execution of generated concrete test cases [44].

There are many taxonomies to classify MBT approaches, Utting et al. proposed a taxonomy with six possible dimensions, grouped in three different categories [44]. The process exhibited in the figure 2.1 can be linked to some of these dimensions. These

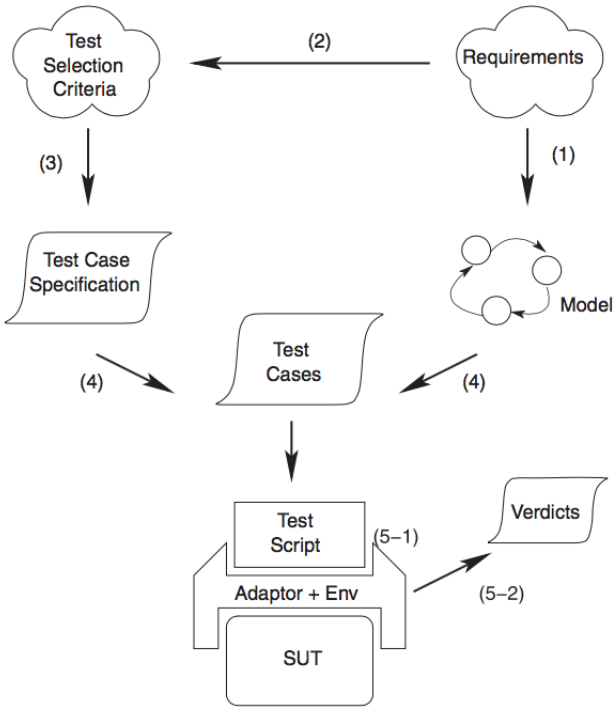


Figure 2.1: Utting et al. Process of model-based testing

dimensions are independent from each other in some way e.g., in some cases the selection of a characteristic could lead to a limitation on the possible combinations for the desired approach.

Starting from the requirements in the figure 2.1. The First step is related to the category "model specification", and covers three dimensions: model scope, model characteristic and model paradigm.

- **Model scope** - it is a binary decision: does the model specify only the inputs or does it specify the expected input–output behaviour of the SUT? Input-only is easier to specify, however the generated test cases are incapable of verifying the correct behavior of the SUT. Input-output is more complex and able to predict the output of the SUT for a determined input, by matching the outputs it can be used to verify if the SUT is behaving as it should.

- **Model characteristics** - model characteristics relate to timing, determinism, and the continuous or event-discrete nature of the model. These characteristics are chosen based on what kind of SUT is being tested. The selection of these characteristics is important since it impacts the choice of modelling paradigm, test case generation technology, and tests execution method (online or offline).

- **Model paradigm** - this dimension is related to which paradigm and notation are used to describe the SUT model. There are many different modelling notations. Each one of them describe the model in a different way and different representations of a SUT could generate distinct interpretations of its behavior.

The second, third, and forth steps of the process presented in the figure 2.1, are related to the category "Test Generation". The second and third step, are related to the dimension "test selection criteria". The forth step is related to the dimension "Technology".

- **Test selection criteria** - defines the techniques that are used to control the generation of tests cases. It guides the automatic test case generation to produce a test suite that fulfills the test policy defined for the SUT, and later are converted to test case specifications. It is an important feature between the MBT tools available. There is no best criteria, it is up to the test engineer to choose which will better meet the project's goals.
- **Technology** - the automation potential of MBT is one of the most appealing characteristics it has. With a defined test model and some test case specifications, test cases can be generated using many different techniques. Usually tools for test generation use several techniques to achieve their goal at once, since some techniques are complementary.

The fifth step is related to the category "Test Execution", it defines the timing of test case generation and test execution, and has only one dimension with only two possible options.

- **Online** - means that test generation algorithms can react to the SUT outputs. Test cases are generated at the same time they are tested in the SUT. The model is used to predict the SUT's behavior and to validates it.
- **Offline** - in this approach, test cases are generated before they are run. The model is used to generate sets of test cases that are run when necessary to validate the SUT's behavior.

Chapter 3

Tests based on UML State Machine

Model-based testing relies on explicit behavior models that describe the intended behaviour of a system and sometimes of its environment. The necessity to validate the model implies that it must be simpler than the SUT, or easier to check, modify and maintain. The model must be precise to serve as a basis for the generation of meaningful test cases [43].

The UML 2.0 is a set of notations, as the state machines, some of them can be classified as transition-based notations. The UML state machine (a.k.a. UML statechart) is an enhancement of the mathematical concept of finite-state machines (FSM) or finite-state automata. FSM is a mathematical model used to design software and logic circuits for computational purpose, it is an abstract machine with states and triggers which represent the logic behavior of a system [4].

3.1 UML State Machine

By using a state machine, it is possible to model the behavior of an individual object. A state machine describes the sequences of states and the behavior of an object during its lifetime in reaction to inputs and its current outputs.

Two types of state machines defined in UML are: behavioral state machine and protocol state machine [36]. A simple example of a UML state machine is displayed in the figure 3.1.

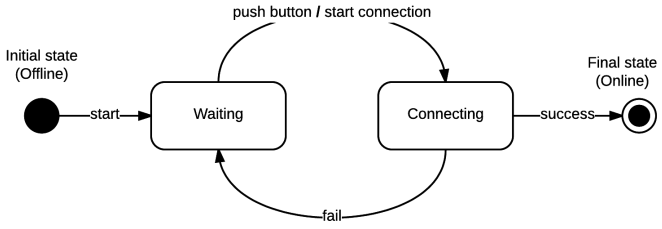


Figure 3.1: Simple UML state machine example

State machines are able to model the behavior of any modeling element, that could be a class, a use case, a component, or an entire system. The UML provides the graphical

notation to represent the states, transitions, events and actions of an object and they all possess its proper definitions [36]:

- **States -** is defined as a condition or a situation during the life cycle of an object which satisfies some criteria. In the figure 3.1 states are presented by the two rounded boxes labelled "waiting" and "connecting".
- **Initial and Final States -** two special states that indicates the starting place and the enclosing state for the state machine, respectively. In the figure 3.1, the filled black circle is the initial (offline) state, and the the outlined black circle is the final (online) state .
- **Transitions -** are the relationship between two states indicating that an object in the respective first state will enter the second state after performing a certain action. It is represented in the figure 3.1 as the bottom arrow labelled "fail" in the waiting and connection states.
- **Event trigger -** is an occurrence of an event that has a location in time and space, it is an event that can trigger a state transition. It is represented in the figure 3.1 by the top arrow, as the "push button" before the forward slash.
- **Action -** it is an executable atomic computation, which means that it cannot be interrupted by an event. It is represented in the figure 3.1 by the top arrow, as the "start connection" after the forward slash.

Behavioural State Machine

A behavioral state machine is used to specify discrete behavior (life cycle) of various model elements (classes, instances, subsystems, or components) through finite state transitions. The state machine formalism used is an object-based variant of Harel statecharts [13].

The behavior is modeled as a graph of state nodes connected by transitions. Transitions are triggered by the completion of series of key events. During the traversal of the model, the state machine could also execute some activities [40].

A Behavioral state machine could be owned by a behaviored classifier (called its context). The context defines which signals and triggers are important for the state machine, and which operations and attributes are available in activities of the state machine.

Without a context classifier, a state machine may use triggers that are independent of receptions or operations of a classifier, for example, signal triggers or call triggers based on template parameters of the parametrized state machine [40].

Protocol State Machine

A protocol state machine is a specialization or a subset of a behavioral state machine. It is used to describe a usage protocol or a life cycle of a classifier and to express complex protocols (e.g. communication protocols). It determines which operations of the classifier could be called in which state and with which conditions, this way specifying the call sequences, just like a graphical visualization of a protocol. Protocol state machine is

always defined in the context of a classifier. A classifier may have several protocol state machines [41].

3.2 Tools for online testing

There are several solutions for test case generation from models, all of them targeting a specific characteristic of the SUTs, different input models, test case generation algorithm, and much more [44]. The MIT has a website [27] dedicated to list some of these tools with short description of the tools. In the following are presented some free tools used for test case generation considered for this project so far.

3.2.1 GraphWalker

It's a Model-Based testing tool built in Java. Works reading models like finite-state diagrams, or directed graphs. The test generated from the models can be run, either offline or online [31].

State machines are modeled with GraphML (a language to describe graphs), the tool generates offline and online test sequences from a model called GraphML, an example of the model is presented in the figure 3.2. There are seven built-in coverage (stopping) criteria for test generation. It can be integrated with a Java test harness or called with SOAP (as a web service).

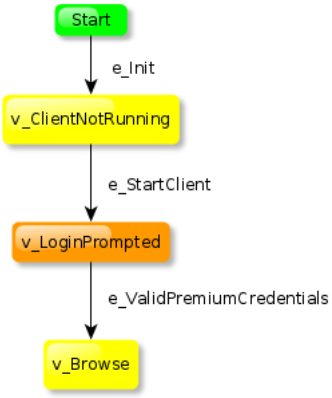


Figure 3.2: GraphML, model used by GraphWalker[27]

3.2.2 ModelJunit

It is a Java library that extends JUnit enabling it to support model-based testing. Models are written in Java and works as extended finite state machines. The models directly interact with a Java SUT or adapter, generated tests from these models could have different coverage metrics. ModelJUnit is an open source tool [42].

3.2.3 PolarSys (former TopCased)

It is model-driven development and testing environment for critical hw/sw embedded systems. It accepts different types of models (SysML, UML, OCL), and is an open source tool for development of embedded systems. In July 2015, Topcase's servers were shutdown and the community of developers is now working on a new project, called PolarSys.

PolarSys, like TopCased, is an open source solution, a plugin for the Eclipse IDE. PolarSys is an eclipse industry working group to collaborate on the creation and support of open source tools for the development of embedded systems.

It provides an open-source package dedicated to embedded systems development. Among the solutions present in this package is the eclipse Titan project.

With model-based testing, the test cases generated from the model are necessarily abstract, which means that the tests does not have low level information about the system. The Titan project goal is to generate executable test cases from abstract test cases generated from a model [34].

Chapter 4

Service oriented architecture

SOA is a means of organizing solutions that promotes reuse, growth and interoperability. It is a paradigm that enables one system to aggregate others capabilities, that can be owned by it or others, in order to execute a determined function. This chapter will present some information about SOA, presenting its concept, definition of service compositions: orchestrations and choreographs, and an example of a language use to describe a service composition. SOA is a very complex subject, this chapter is a quick introduction designed to enable the understanding by any reader of the next chapters, which will present some components of SOA passive of testing and the SAMBA Online test Framework.

4.1 Concept

Service Oriented Architecture (SOA) was born around the year 2000, in the same period that the concept of the web 2.0 started to be disseminated. As new devices started to grant access to the Internet, like Personal Digital Assistants (PDAs) and the first smartphones, the need to fragment the web content so it could fit these new devices raised up [16]. By 2005, the term Web 2.0 was more a buzzword than a real definition. The concept of Web as a Service started to be accepted as the web community slowly developed better understanding of the integration between sites, one example of it is the banner ad services [17].

SOA is a paradigm for organizing and utilizing distributed services that may be under the control of different ownership domains. It provides means to offer, discover, interact and use capabilities to achieve a goal with measurable expectations and preconditions [30]. Services are usually described in Web service description language (WSDL), which is a XML format to describe a service as a set of endpoints operating on messages that carry procedure-oriented or document-oriented information [45].

Complex SOA applications usually rely on several services to perform all of its functions. In order to describe the business process and facilitate the orchestration of synchronous (client-server) and asynchronous (P2P) Web Services some modeling languages were created. Among these is the Web Services Business Process Execution Language (WS-BPEL), which is an open standard, developed to be interoperable and portable across many environments [45]. WS-BPEL defines a grammar and a model for describing

the flow of a business process between the SOA application and its partners. WS-BPEL defines the multiple interactions among partner services, coordinating them to achieve a business goal [38].

Differently from Object Oriented Programming paradigms, which focus on packaging data exchange with operations, the focus of Service Oriented Architecture is the task that needs to be executed.

4.2 Service composition

Web Service (WS) Composition tries to solve the problems of business process integration. A composition uses the capabilities offered by different WSes in order to achieve a goal [46].

In simple words, a composition creates a new WS from the integration of existing WSes utilizing a composition pattern. There are two main composition patterns [33]: Web service Orchestration and Web Service Choreography.

4.2.1 Orchestration

Can be resumed as an executable business process that can interact with both proprietary and third party Web services (WS). The orchestration needs to be executed (controlled) by one of the participants of the SOA composition, which is usually an web service that belongs to the SOA composition owner.

The orchestrator describes the data flow and sequence of the WS in order to achieve a business goal. The WSes in this approach doesn't have information or are not aware of the other WSes that are involved during the business process. The industry is trying to establish a uniform Web Service Orchestrator (WSO) description language specification, such as Yet Another Workflow Language (YAWL), Web Application Description Language (WADL), and Business Process Execution Language (BPEL).

The most important elements of a WSO language are the following [46]:

- basic constructs, called atomic activities, to model invocations, listeners and replies for the WSes, and other inner basic functions;
- information and variables exchanged between WSes;
- control flows called structural activities to orchestrate activities;
- other inner transaction processing mechanisms, such as exception definitions and throwing mechanisms, event definitions and response mechanisms.

4.2.2 Choreography

A Web Service Choreography (WSC) is a protocol (or a contract) among Web Services. A WSC is a collaborative effort of the involved parties to deliver the expected behavior and outputs as requested.

The most important elements of a WSC are the following [46]:

- partners within a WSC, including the role of a partner acting as and relationships among partners;
- local states exchanged among the interacting WSes;
- interaction points and interaction behaviors defined as the core contents in a WSC;
- sequence definitions of interaction behaviors;
- other inner transaction processing mechanisms, such as exception definitions and throwing mechanisms, event definitions and response mechanisms.

4.3 WS-BPEL

The Web Service Business Processes Execution Language is an "Executable business processes model of the actual behavior of a composition interaction; and abstract business processes that are partially specified processes which are not intended to be executed" [38]. An abstraction process does not have full information about the operational procedures that are executed, its role is to describe the possible use cases of the composition. BPEL was developed to be used as behavior model for both executable and abstract process.

BPEL is an XML based language which has its own syntax to describe the data flow between web services. Web service description language (WSDL) is used as base to describe capabilities of the WSes in the process flows. As the Business process is defined, it can be exposed as a WS again using WSDL. BPEL also defines exception handling mechanism of operations and transactions on operations [38].

The example in the figure4.1 is retrieved from the WS-BPEL standard documentation [38], and it describes the process to deal with purchase orders. The figure 4.2 is just to illustrate the basic structure and some concepts of the language, and it is related with the figure 4.1.

The light and dark gray boxes in the figure 4.1 are the input and the output ports of the service, these ports store the operations used to invoke a respective service function. Callbacks of the operations are represented by small white rectangles with arrows. The figure 4.2 exhibits a snippet of a BPEL file, which contains the declaration of the services used, the service name, and two ports used to invoke functions of the invoice and the shipping services.

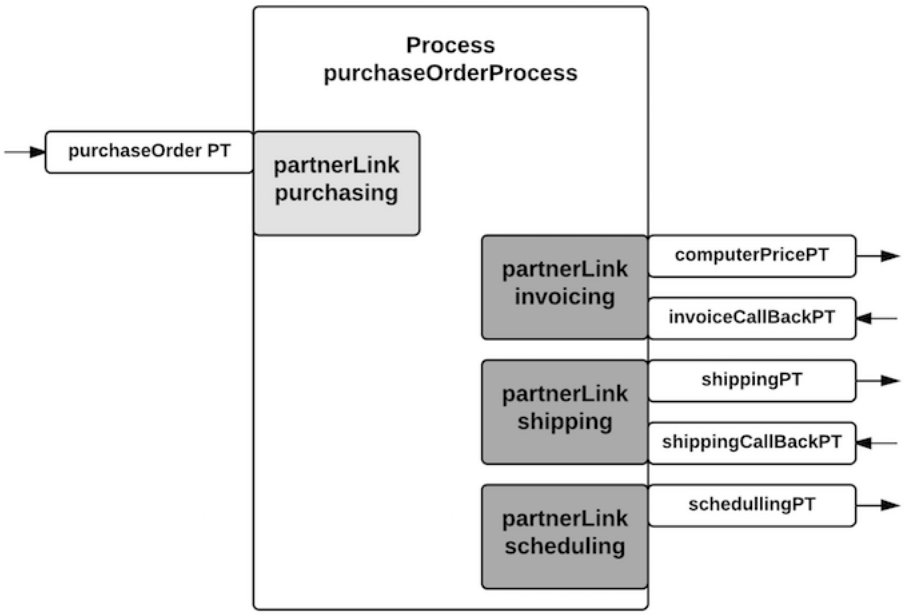


Figure 4.1: Purchase order service illustration

```
<wdsl:definitions
targetNamespace="http://manufacturing.org/wsdl/purchase"
    xmlns:sns="http://manufacturing.org/xsd/purchase"
    xmlns:pos="http://manufacturing.org/wsdl/purchase"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <wsdl:types>
        <xsd:schema>
            <xsd:import namespace="http://manufacturing.org/xsd/purchase"
            schemaLocation="http://manufacturing.org/xsd/purchase.xsd" />
        </xsd:schema>
    </wsdl:types>

    <!-- portType supported by the invoice services -->

    <wsdl:portType name="computePricePT">
        <wsdl:operation name="initiatePriceCalculation">
            <wsdl:input message="pos:POMessage" />
            </wsdl:operation>
            <wsdl:operation name="sendShippingPrice">
            <wsdl:input message="pos:shippingInfoMessage" />
        </wsdl:operation>
    </wsdl:portType>

    <!-- portType supported by the shipping service -->

    <wsdl:portType name="shippingPT">
        <wsdl:operation name="requestShipping">
            <wsdl:input message="pos:shippingRequestMessage" />
            <wsdl:output message="pos:shippingInfoMessage" />
            <wsdl:fault name="cannotCompleteOrder"
             message="pos:orderFaultType" />
        </wsdl:operation>
    </wsdl:portType>
```

Figure 4.2: BPEL file example code (snippet)

Chapter 5

SOA Testing: challenges and related works

Service oriented architecture is a design pattern composed of many independent services, which follows well defined standards. The services that compose a SOA application can be described, published, categorized, discovered and dynamically assembled, bound an unbound at any time [22].

Because of SOA application's characteristics many established and well known tools and methods are not applicable and don't work with services. From the perspective of users and service integrators, services are accessible by interfaces, without the proper information about the services, white-box testing methods based on data flow knowledge and code structure became not feasible. Mutation-test approaches, which require inserting the source code with errors, are also affected [9].

SOA testing efforts aim in the elements to be tested, which aspects should be tested for (or against), and the artifacts generated during the testing process. Testing handbooks list non-executable artifacts (strategies for testing the completeness, correctness, and accuracy of requirements, architecture, design, documentation, and more), and executable artifacts (which include unit, integration, and systems-level testing of executable software) [28]. In the following sections more information about SOA testing characteristics, types and challenges will be presented.

5.1 About SOA Testing

SOA implementation testing does not differ from testing traditional systems. SOA testing addresses: [28]

- functionality: Functional capabilities are tested to determine whether requirements and other expectations will be (or are being) met. Verification of correct availability of services is a example of functional test.
- non-functional attributes: The characteristics that are needed to determine the quality attributes, must be tested. Some of the quality attributes considered important are presented in the table 5.1, descriptions of the implications from SOA testing are also presented.

- conformance: Testing verify if specific guidelines and standards that the organization has chosen to adopt are being respected. There are many standards and guidelines available, W3C [45] and OASIS [30] are examples of organization that work with web services.

Quality Attribute	Implications of SOA on the quality Attribute
Availability	Services may not be under control of consumer their availability must be guaranteed. To ensure the availability multiple identical instances of a service could be deployed, however that could lead to a need to test consistency between the instances. Diverse instances of a service complicates testing as each implementation must be tested.
Modifiability	A service implementation can be changed as long as the service interface is unchanged and prior functionality is maintained. In some cases testers do not have the service source code, which makes testing complicated. Newer versions of a service may not be backwards compatible with preview tested service versions.
Interoperability	Published service interfaces and common protocols are used to make services interoperable, however, semantic interoperability has minimum support. Difficulty to invoke services across differing infrastructures, because the same features can be implemented in may different ways in SOA infrastructures.
Performance	Web services has negative effect on run-time performance of SOA for two reasons: Overhead caused by the XML handling (serialization, deserialization, validation, and transformation); larger messages that require more bandwidth.
Adaptability	Dynamic environment forces tests to balance between the desire for full testing and the need for rapid deployment, regression testing and automated test suites techniques become critical.
Reliability	Failures that derive from the distributed nature of SOA systems need to be tested. There are standards to help message-level reliability, but service-level reliability depends on the implementation.
Security	Testing need to ensure that security is maintained even when information crosses untrusted boundaries, since service consumer reuse existing services data can flow across untrusted networks.

Table 5.1: non-functional attributes quality attribute list

5.1.1 Artifacts to be Tested

Functional, non-functional, and conformance characteristics should be considered in the context of the artifacts available for testing, specifically SOA infrastructure, web services, and the different classes of service consumers.

SOA infrastructure

The SOA infrastructure typically offers capabilities and services that register and discover other services, manage metadata, deliver messages and provide security. It can be composed by commercial products, custom-developed software, open source software, or any combination of these components. An SOA infrastructure stack typically consists of [28]:

- an enterprise service bus (ESB) application server
- a registry
- web servers
- databases and data stores

There is huge variation of SOA infrastructure capabilities available, the core elements of infrastructure stack, development tools, testing, integration, software engineering activities and capabilities can differ depending of the commercial vendor providing it.

Web Services

Individual services and composites of services are included in the test of web services. An individual WS usually provides coarse-grained, business-level capability. The WS elements that are passive of testing are the service interface, service implementation, the service level agreement (SLA), message format and payload. As previously mentioned, WS orchestration and choreography are two types of composite services used in SOA environments [46].

End-to-End Threads

The combination of humans, services, applications, back-end applications, and databases that uses network infrastructure and the SOA to achieve a business goal is called End-to-end thread. End-to-end threads include services with other interacting components (human, functional, infrastructural), and the operating environment [28].

5.1.2 Perspectives, Roles, and Responsibilities

There are many different roles played by individuals and organizations during the development of a SOA implementation. Each role performs an important part in SOA testing, if there is a problem during the definitions of what to expect from each specific role related to test, many problems can appear, hence affecting the resolution. Its is important to determine roles and imagine the issues the product would have in the context of the role [28]. The table 5.2 lists several roles that individuals and organizations play in developing an SOA implementation. Defined expectations of each role avoid problems when identifying the source of a problem, it also helps identify the role of each individual service provider in relation to the composition or end-to-end threads, which simplify the processes of setting up the test environment and the result analysis[22][9].

Role	Action	Recommendation for SOA Testing
Service developer	Implements services internals and their interfaces using existing components and packing it as a service or forming a service implementation.	Define guidelines for test before the service is ready for use.
Service provider	Whoever provides services. The service provider might not be the service developer, a service developer could also be a service provider.	Establish the responsibilities of testing and customer support allow for distinction between service providers and developers.
Service consumer	Whoever uses a service (individual or composite) directly	Developed methods to identify use cases to generate tests that can cover the requirements of performance expected for users when customer support is requested.
Service integrator	Uses existing services to create new services.	Develop guidelines for testing composite services of any type, critical composite services and composites employing a large number of composition mechanisms should have especial attention.
Infrastructure provider	Provides the necessary middleware infrastructure for the SOA (e.g., Enterprise Service Bus, data bases).	Guidelines for the processes of verification/testing of infrastructure capabilities updated. Determine the type and level of testing support provided by the infrastructure provider to the service provider or integrator
Third-party service tester/certifier	Validates and certifies whether a service works as expected.	Third-party testing/certification activities must have their focus, expectations and limitations identified.
End user	Uses the application, which relies on services	Forms of testing performed should accord to SOA roles and have policies to delineate it.

Table 5.2: List of roles that individuals and organizations play in SOA implementations

5.2 Functional SOA Testing Levels

There are two types of WS composition: orchestration and choreography. In orchestrations there is a central node that works as business logic controller, deciding the execution order of the components. Choreography involves multiple parties and sources, all components in the process are autonomous and control their own agenda. Both orchestrations and choreographies have their own characteristics and it affects the testing process differ-

ently [22] [7].

- **Orchestration Testing** - The tests can stress the components of the orchestration, the operation sequence or the business logic. White-box or black-box techniques can be applied depending on how much information the tester has about the orchestrator and its components [22].

- **Choreography Testing** - The flexibility of service choreographies, and the difficulty to monitor its components makes it more challenging for testing than orchestrations [9]. Unit tests can be applied, since it is possible to consider a choreography as a service unit. Integration tests usually rely on model-based techniques to validated the proper behavior of the composition or its components [8].

Depending the of role and context of the tester different techniques are used, service providers and integrators can use techniques common in subsystem or component testing to perform functional testing. As services are published their specification is shared and used to enable service discovery. The information shared in the service specification includes, for example, a WSDL reference to the service's interfaces. This information is used by integrators to generate test cases using black-box strategies[7].

The different levels of information available on the services, enables different and more sophisticated test strategies to be used. There are different test levels and techniques to be applied to to SOA, information about this is presented next.

5.2.1 Testing SOA Infrastructure Capabilities

The capabilities are provided by the SOA infrastructure, services, usually used by the business processes, which rely on them to exist. Some common capabilities that should be tested are [28]:

- **Service Registration Mechanism** - The goal is to ensure that only validated services become part of the SOA registry. Testing targets the tools and options provided by the infrastructure to register a new service. It also needs to take into account the version control of services already registered.

- **Service and Resource Discovery Mechanism** - Tests must ensure that infrastructure resources and registered services can be found by consumers.

- **Subscription and Notification Mechanisms** - Service consumers must be informed of new version of a service is registered with the infrastructure. Notifications of service level violations are also required by service providers and consumers.

- **Service Usage Monitoring** - Service providers use the usage data to identify the effects caused by changes in a service.

- **Service Virtualization** - Creates and manages virtual endpoints for web services, it allows a service to be dynamically associated with physical endpoints.

5.2.2 Testing Web Service Functionality

The initial step in testing services is to determine whether each service meets its functional requirements. Components testing methods are usually applied to test functionalities of SOA services, however it presents challenges, some services might not be owned by the user, and there are many other aspects of testing related to the interaction of a service and the other components that need to be covered.

Stubs are a common software strategy to "mock" the behavior of unavailable components, useful when there is a need to test services that need to interact with other services that are not yet available. Mocking is a common practice in service testing, many open source and commercial implementations (e.g., apache synapse, soapUI) help service developers to mock services and perform unit tests. The limitations to this approach are:

- Testing result depends on the fidelity of the mock service to the real service. Complex services are difficult to mock, so there are practical limitations of the types of services that can be mocked.
- While development is not finished behaviors and interfaces of services are not finalized, mocks are helpful because of this, however, this leads to uncertainty on test results.
- QoS cannot be mocked. QoS test depends directly on environment and components, mocks would not produce reliable results.

5.2.3 Fault Injection

Fault injection is used to check if a system can detect a fault an recover from it, the faults are made by making deliberated mistakes in the implementation of the service. this test technique can be applied to both individual WS and compositions [28], it focus on determining that services do not reach unexpected states or produce unintended behavior.

A common technique to fault injection on services is called fuzzing, which generates invalid input data, the results are then analyzed to check if the web service was compromised. Some examples of types of injected faults are a perturbation in the message (e.g., different data types in SOAP or REST messages), timing delays on compositions (e.g., forcing timeouts), and malformed service interface (e.g., WSDL with errors). Reliability, security and performance could also be tested using fault injection.

5.2.4 Regression Testing

Regression testing focus on the retest of software whenever it is necessary to guarantee that any feature of the service has not been compromised during the evolution of any of its service components [28].

In SOA environment the service provider is the one responsible for controlling the evolution of the services, this makes the testing more complicated. Service users and systems integrators need to be aware of the release strategy in order to perform this kind of test, however the service provider might not be aware of all the users of his service,

these users might not be notified about the future changes and updates of the service's interface or implementation [9].

The lack of control of the integrator over the components is what differentiates SOAs from component-based systems and other distributed systems. Since the the integrator has no knowledge of the internal functional of the services it relies, it might never notice during the service lifetime that changes happened, since in many cases it only has access to the interfaces of the services it uses [9].

Regression testing should be triggered whenever changes happen in any of these cases [22]:

- Life-cycle management of components
- Service composition/bindings
- Deployment configurations
- Back-end systems or data sources
- Service's functional behavior
- Service's non-functional behavior
- Requestor/responder application
- Infrastructure and/or services invoked
- Back-end systems upgrades to Service Contracts (WSDL)

5.2.5 End-to-End Testing

End-to-end testing refers to testing of the entire business process, all the services and capabilities involved in executing a specific goal or task. A service might rely in several different components, some of them might be out of the service owner control, therefore, the execution of tests on business flows involves everything, from data input from humans, to invocations to external services [22].

End-to-end testing should consider [28]:

- **Decentralized ownership and lack of centralized control** - there is a need to create an environment (through negotiation) for testing, tests must be coordinated with multiple service providers, owners and back-end systems.
- **Long-running business activities** - SOA implementations usually interact with other business processes and workflows.
- **Loosely Coupled Elements** - The advantage is that changes in a single entity have minimal impact on the other interfacing entities, but testing becomes an issue, test compositions need to be done through simple interactions rather than complex ones.

- **Complexity** - The information about specific services invoked, attachments included and parameters is embedded in protocols meant to machine-to-machine interaction.
- **Regression testing** - Maintaining awareness of changes at any service requires agreements regarding what, when, the types of changes that must be notified and are allowed to occur, and how to notify them.

5.3 SOA Testing Challenges

Testing a SOA system is challenging, some of the limitations to perform tests are [9]:

- Limited access to information about service's code and infrastructure by users.
- Lack of control caused by the independent infrastructure and evolution mechanism of components under control of the service provider.
- Adaptiveness and dynamicity limit the testers ability to determine the web services that are involved in the workflow execution.
- Costs related to tests, cause by the use of services, disruption on a service (caused by massive testing), and possible testing side effects to some systems (e.g., safety systems and market systems).

Also each structural components of SOA system present different challenges to test, these challenges are present next [28].

5.3.1 SOA Infrastructure Testing Challenges

There are five factors that impact tests of SOA infrastructures: limited technical information about components, complex configuration of an infrastructure, rapid release cycles for components in an infrastructure, lack of a uniform view, and variation across an infrastructure [28].

5.3.2 Service Challenges

It is necessary to classify services in two classes in order to address properly the challenges related to them: individual WS and composite WS. In the perspective of individual WS there are four challenges to test: Lack of source and binary code, unknown contexts and environments, unanticipated demand and impact on quality of service, and standards conformance. In the perspective of composite WS, which includes end-to-end threads, there are eight documented challenges: Service (component) availability, multiple provider coordination, common semantic data model, lack of a common fault model, transaction management, test side effects, independent evolution of participating services, lack of testing tool support [28].

5.3.3 Environment for Testing

There is one challenge related to environment for testing, the difference between test and deployment environments. The test environment should allow developers test the integration of services and service compositions before the actual deployment, and allow testers to design, implement, perform, manage the whole testing process and collect information about the functional and quality aspects of SOA service, infrastructure, and end-to-end threads [28].

5.4 Related works

Since this work will cover a service composition that is described by BPEL, which means that it has a coordinator responsible for handling the control-flow and the data flow of third party services, it is classified as an orchestration. There are many published frameworks, tools and techniques related to test of web services. As presented by Bucchiarone et al. [8], service composition test is mainly divided into two complementary views: orchestration and choreography.

Tests in a web service composition could be executed offline and online. In general, offline tests are executed during the development of the composition. Since the goal of this work is to generate and execute tests during runtime, the approach will rely on model-based online testing to achieve its objective.

Since several techniques and tools have been developed to generate and execute tests on web services, some publications were selected to be used as references for this project.

The articles selected tackle the problem of third-party services that are tested during runtime of a composition and can evolve in ways not predicted by its costumers. In order to detect failures, third-party services should be tested periodically during the composition life cycle. The following projects present solutions for this problem and some possible applications for it that go beyond its main objective.

5.4.1 Enabling proactive adaptation through just-in-time testing of conversational services [19]

The article claims that failures detected should be used to start the process of adaptation of the service-based application (SBA) to guarantee and sustain its expected functionality and quality.

The challenge in the proposed solution is how to trigger adaptations if conversational services are employed in the composition. A conversational service is one that only accepts specific invocations sequences of operations. Some operations may have preconditions which depend on the state of the service. A collection of acceptable invocation sequences is called a *protocol* or choreography of the service. Unexpected behavior during an execution of a protocol could be costly and should be avoided.

The paper presents an automated technique that determines whether adaptations are necessary when there is a conversational service, thus avoiding expensive compensation actions. Just-in-time online tests of the relevant operation sequences of the constituent

service are performed. Just-in-time means that before a service is invoked for the first time on the composition, it is tested for detection of potential deviations from its specific protocol.

The generation of test cases is grounded in the formal theory of Stream X-machines (SXMs). To ensure that the tests will be executed in feasible cost and time, test cases are generated using a reduced model of the composition.

The researchers introduced the technique and declared that it could enable compositions to proactively trigger adaptations on SOA applications, presenting results by comparing the length and sizes of tests sets generated using complete SXM and reduced SXM. As future work, the use of adaptation strategies in case of changes in the composition or context are considered, as an extension of the proposed technique.

5.4.2 Online testing framework for web services [10]

The paper points out that conformance testing is a commonly used activity to improve the system reliability, and that it is very time-consuming and requires a great deal of experience on the target systems. The use of a specification, like BPEL, could provide a support for automating the test process.

By using the information about the composite web service, it is possible to code it in a different language following the specifications given and then this new representation of the composition is tested. This differs from Black-Box tests approach, in which no knowledge of the inner logic of the services is available. The tests are generated from the information in the specification, but without information of the services internal structure. Because of this the approach was called *gray-box*.

Using the gray-box approach, the paper presents a framework that automatically generates and executes online tests for a composite of web services described in BPEL. The BPEL is converted to a timed extended finite machine (TSFM) model, which is able to represent time constraints and data variables.

Using the model, the paper proposes an online testing algorithm which is responsible for generating the test cases, and for the execution and determination of the results of the tests on distributed testing architecture.

In addition to the theoretical framework, a prototype tool called WSOTF was developed. The framework has limitations: test cases cannot be selected because the algorithm randomly selects it. Future works will include a memory to improve the random selection of tests. In the case of flow activity, if the service invokes many actions on parallel and validates the timing constraints, these timing constraints are maybe not validated because the framework works (with a flow activity) as a sequence.

5.4.3 A testing service for lifelong validation of dynamic SOA [12]

The paper proposes a SOA testing approach based on a composite service that is able to trace SOA evolution and automatically test the various services according to specific testing policies. The architecture of the testing service is described and a concrete implementation focused on robustness testing, which checks if a system can function correctly

if invalid inputs of stressful environments, is presented.

An issue related to SOA applications is verification and validation (V&V) of SOAs. The service provider most of the time neither is aware of the existence of other services nor has control over it. One other problem is that SOA evolves during in-service, that means that services may be modified or new services can enter/exit the system during run-time. Integration of third party services remain a necessity, consequently, the provider may not be able to perform traditional (offline) V&V (e.g., testing) on the entire set of services.

The paper focuses on lifelong testing of SOAs. This approach, based on online service discovery and testing actions that are applied according to specific algorithms, is implemented via the introduction in the SOA of a testing service. This service becomes part of the architecture and is managed by the providers that require the continuous evaluation of the composition.

The proposed architecture of the testing service is a composite web service that uses basic information given by some providers about the services known, owned or managed. This information is used to build an initial description of the SOA and then to automatically discover the other services that operate within it. This information is kept up-to-date by detecting services that evolve, enter or exit the architecture.

The services listed in the SOA description are periodically tested by the testing service. An important feature is that the testing service is able to operate when the system is offline and also at runtime. The testing service is not designed for any specific test. The paper also presents an implementation focusing on robustness, which uses the API of the wsrbench tool, an on-line tool for robustness benchmarking.

5.4.4 PLeTS - A Software Product Line for Model-Based Testing Tools [14]

The paper claims that with the popularization of model-based testing (MBT), which contain several advantages compared to other testing techniques, like the reduction of system requirement interpretation errors by developers, engineers and testers. Many tools developed exploit the advantages of MBT like Conformiq Qtronic and AGEDIS, both tools generate test cases from different modeling languages, these tools have a lot of characteristics in common, despite the research group and companies that developed them. The author proposes the use of software product lines in order to reduce the problems related to development, integration and evolution of MBT tools.

In order to define the requirements necessary to develop the the PLeTs a systematic mapping of publications was made to track the MBT tools available, as well as their core features and characteristics. A partnership with a software development company, specialized in software test technologies, was made in order to uncover some requirements and validate the common features of the tools found. The requirements for the PLeTs were defined as:

- Derived tools must support automatic test case generation based on models.
- Tools must be able to be integrated with external test execution tools.

- Minimum manual intervention
- Should not rely on proprietary tools or environment.

The model of characteristics of PLeTs is composed of four basic features, each basic feature represents a step on the MBT process. The features are: *Parser*, it extracts the information from specialized models; *TestCaseGenerator*, that could be specialized for many types of tests, test case generation algorithms and more; *ScriptGenerator*, optional feature that enables the tool to generate scripts that could be used to guide test execution on testing tools; *Executor* Optional feature that enables the tools generated to execute and start the test case execution automatically.

The PLeTs was used in two scenarios, and generated four tools, two tools for each scenario. The first scenario focused on performance test, and the second scenario focused on structural tests. Using the number of lines of code as parameter to effort necessary to generate these tools, the author consider this a significant improvement as it was easier to develop the specific components to finish the tools generated. Future works will be developed to include functional test to the possible features supported by the PLeTs.

5.4.5 Project Comparison

Some papers presented in this chapter are not directly related, but since the Self-adaptive model-based online testing framework develop in this project is a complex system that combines many different techniques and methods, some features of the solutions present in these papers are related with the project.

The previous works will be addressed from now with different cognomens, following the related works presentation sequence: Enabling proactive adaptation through just-in-time testing of conversational services, will be addressed as JITC; Online testing framework for web services, will be addressed as OTFWS; A testing service for lifelong validation of dynamic SOA, will be addressed as TLIVD; A Software Product Line for Model-Based Testing Tools, as the article suggest, will be addressed as PLeTS. The Self-adaptive model-based online testing framework present in this work will be adressed as SAMBA. The table 5.3 exhibit how the presented works relate to the project.

Related to the results of these works, unfortunately the OTFWS paper, which is among the present works the one with more research fields in common with this project, only presented as result test cases generated from a model. The JITC also present only a comparison of the differences between test cases generated before and after an adaptation is required without test case execution results. The TLIVD present results on robustness tests in a test scenario, which gave evidences that the designed testing service works. The PLeTS, the work with the least in common research fields, only used the models as input for the test case generation, unfortunately the test cases were all scripted before their execution, which classifies the tests as offline.

Relationship with the project					
Research Field	**JIT**	**OTF**	**TLIV**	**PLeTS**	**SAMBA**
Online Test	✓	✓	✓		✓
SOA Composition Testing	✓	✓	✓		✓
Model-Based Test	✓	✓		✓	✓
Model-Generation		✓			✓
Self-adaptive System					✓

Table 5.3: Project's research fields compared with related works

Chapter 6

SAMBA Online Test Framework

As described in the preview chapter there are many challenges related to SOA testing: "limited access to information about the consumed services, lack of control over the components of the SOA application, the dynamicity of the environment, and the costs related to tests are major issues".

Other issues are raised depending of the target SOA structural component to be tested. Between the three structural component, infrastructure, services, and environment, the challenges related to test services, more precisely, composite web services, stand out when compared to the others structural components test challenges.

Among the challenges related to composite web service testing is the *independent evolution of participating services*, this problems is caused by unexpected changes on individual services. Service integrators usually have no control over the services they consume or the evolution mechanisms of these services, yet any change in one of these WS components may result in failures of the SOA application [28].

If a change passes unnoticed, the service integrator may be unable to identify the source of the possible failures. Even when the service integrator is aware of a change, it still needs to perform regression tests to ensure the proper behavior of the SOA application.

Another important fact is that service integrators may not be able to perform traditional (offline) Verification and Validation (V&V) of SOA applications while they are online, which include the test process, on the entire set of services. This raises a problem considering the interoperability requirements of the SOA paradigm [11].

Since information about the service components is crucial for the service integrator to trigger regression tests, service orchestrations, when compared with service choreographies, have better characteristics to make a possible solution for the presented challenges feasible. Briefly, the challenges selected for project are:

- Target orchestrated compositions
- Focus on regression test
- Handle the dynamicity and independent evolution of participating services
- Avoid costs and side effects related to tests
- Perform online testing

The project main goal is to to generate and perform regression tests on dynamic orchestrated WS compositions, with test cases designed with updated information about the target composition, and execution of tests with minimum interference over the system under test (SUT) during runtime.

The SAMBA Online Test Framework (SAMBA Framework), which performs composition regression tests, is explained in the following five sections: Requirements, which defines the functional and non-functional requirements of the project; Solution Context, which describe to whom the project should be target and how it interact with other systems; Framework Description, which present the SAMBA Framework; Framework Project, which describe all the components of the framework, how they interact and behave; and the Framework Implementation, which describe details about the deployment, tools and libs used in the project.

6.1 Requirements

6.1.1 Functional Requirements

The solution proposed should have the following features:

- Targets composite orchestrations.
- Monitor services, so that changes can be automatically detected.
- Regression test should be triggered when changes are detected.
- Information about test execution must be stored for future analysis.
- Test cases generated and test results must be stored.

6.1.2 Non-functional Requirements

The solution should be designed considering this technical requirement:

- Be modular for easy update.
- Be dynamic and adapt following the evolution of monitored services.
- Avoid costs and side effects related to test execution.
- Solution must use standard communication protocols (SOAP and REST) for easy integration.

6.2 Solution Context

The solution must be focused in the perspective of the service integrator, which uses existing services (individual or composite) either to create composite services or to create

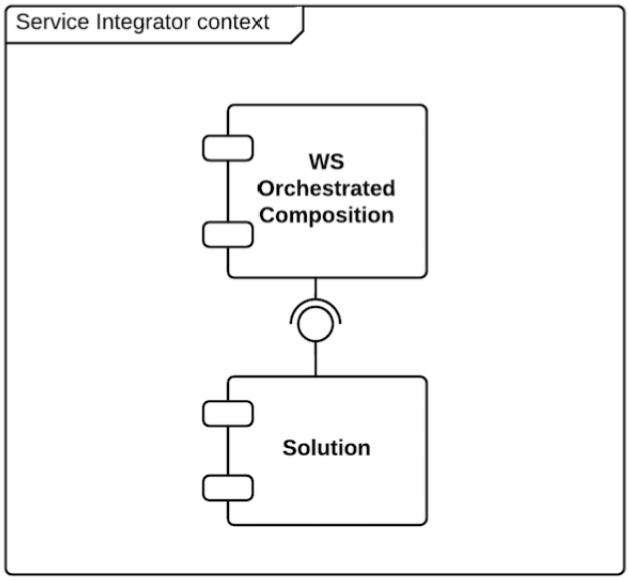


Figure 6.1: Context in which the solution will operate

an end user application [28]. The solution must offer a interface to be integrated to an already existing WS orchestrated composition, the figure 6.1 illustrates this situation.

The WS orchestrated composition must be able to share information about the services in which it relies to achieve its business goal (e.g., names, addresses, and interfaces), the solution must be able to perform regression tests using only the information shared by the WS orchestrated composition.

6.3 Framework Description

6.3.1 Proposed solution

The solution proposed is a self-adaptive model-based online test framework (SAMBA Online Test Framework), it focus on service composition regression tests, using the information available in BPEL files to generate models of the target WS compositions, which are used to generate tests cases. The technique applied to overcome some of the challenges of the SOA application dynamic environment is the Self-adaptation, it has been used as an approach to overcome such problems related to dynamic systems [20] [24]. Combining the ideas of models@run.time, an approach to manage the complexity in runtime environments by developing adaptation mechanisms that leverage software models [6] [35], this project tries to achieves its goals.

Self-adaptability

There are two main components in a Self-adaptive system: A managed system and the managing system (control loop). The managed system in this project are WS compositions, and BPEL orchestrations running on a host server. The managing system is responsible to handle the evolutions of the managed systems, and how to respond to such changes [21].

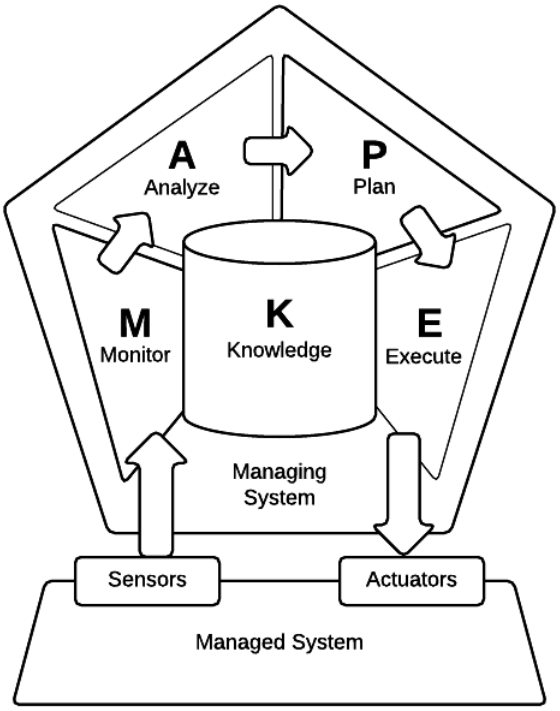


Figure 6.2: MAPE-K control loop

An engineering approach to realize self-adaptation is by a feedback control loop called MAPE-K, which many dynamic adaptive systems (DAS) are based [23], the figure 6.2 illustrates how the control loop is organized and how it interacts with the managed system. MAPE-K is a sequence of four computational phases: Monitor, Analyze, Plan, Execute. These computational phases are independent, however they interact by sharing and manipulating the same knowledge base.

6.3.2 SAMBA Framework description

The SAMBA Framework is presented in the figure 6.3. Four main components are identified: Service Assemble Monitor (SAM), Model Generator (MG), Model-based Online Test Case Generator (MBOTCG), and Test Service (TS).

The SAM is the component responsible of monitor the evolution of the WS compositions on the host server, the MG is responsible of the analysis of the evolution, the MBOTCG is responsible of plan the test cases, and the TS is responsible of the execution. However since the SAMBA Framework performs online regression test, depending of the results of the execution of a test instruction the test case might be changed during its generation.

In order to execute a complete test case a synchronized work between the MBOTCG and TS is necessary. The TS request to the MBOTCG to start the generation of the test case, and progressively requests for new test instructions as it finishes the previews test execution and report the result of the test to the MBOTCG.

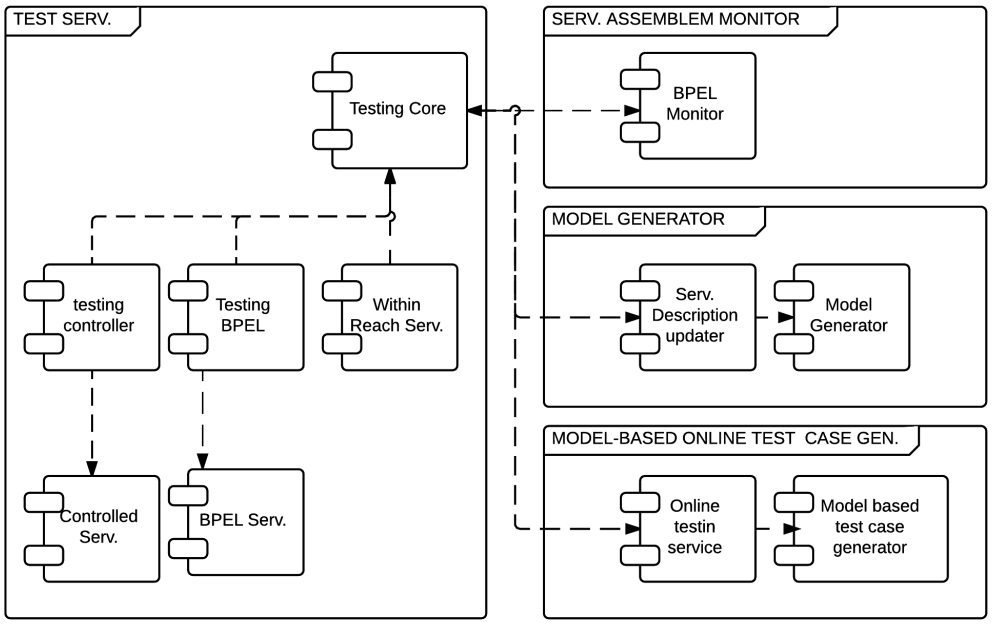


Figure 6.3: Self-adaptive Model-based (SAMBA) Online Test Framework

The SAMBA framework includes the following features:

- A monitoring system to trigger model updates when changes are detected in the WS compositions.
- Extract models from WS composition's BPEL files (changes in the BPEL files are automatically reflected in the models).
- Automatic model-based test case generation from WS composition models.
- Automatic Online model-based test case execution.
- Generation of Test reports and Test execution metadata.

SAMBA Framework components

The SAM is responsible to periodically verify if there is a new version of the monitored WS compositions on the host server and inform the TS which WS composition was updated.

The MG is responsible for the model generation and management of required information to achieve this objective. It also manages the evolution history information of the monitored WS compositions.

The MBOTCG is responsible to handle the model of the target WS composition, setup the parameters used for the test case generation process, send to the Test Service the instructions of which tests should be executed, receive the result of the test execution, and generate a report of the whole test case executed.

The MBOTCG will rely on a tool for the model-based test case generation called Graphwalker [31]. There are many solutions for test case generation from models and some example tools are listed in the section "Tools for online testing" at chapter 4. These tools have their own characteristics and aim at specific situations and problems. Graphwalker was chosen because it has well documented requirements for the model generation, and it also has robust features, which allows for the configuration of algorithms and stop conditions for the test case generation, and many forms to be integrated to any project.

The TS is in charge of executing the tests, following the tests plan communicated by the MBOTCG, collecting test results, and forwarding these results to the MBOTCG. As the test cases are being generated, test instructions are progressively sent by the MBOTCG to the Test Service. These instructions are received, decoded and used to trigger a functional test of the required operation on a specific WS. The output of the test is then sent back to the MBOTCG which uses this information to decide the next step of the test case and to fill the test report.

The TS, originally developed by Ceccarelli et al.[12], was allowed to be used and modified for this project, it was originally designed to detect changes on services that belongs to the compositions, update the description of these services if possible, and perform online robustness tests. The SAMBA Framework will rely on the TS capabilities to perform online tests.

The framework components share access to folders on the host server, in which they can store, read and write files, these folders are the knowledge base, the figure 6.4 illustrates the entity relationship of the elements in the knowledge base. Each WS composition monitored will have its own production directory, none or many archive directories, and none or one report file.

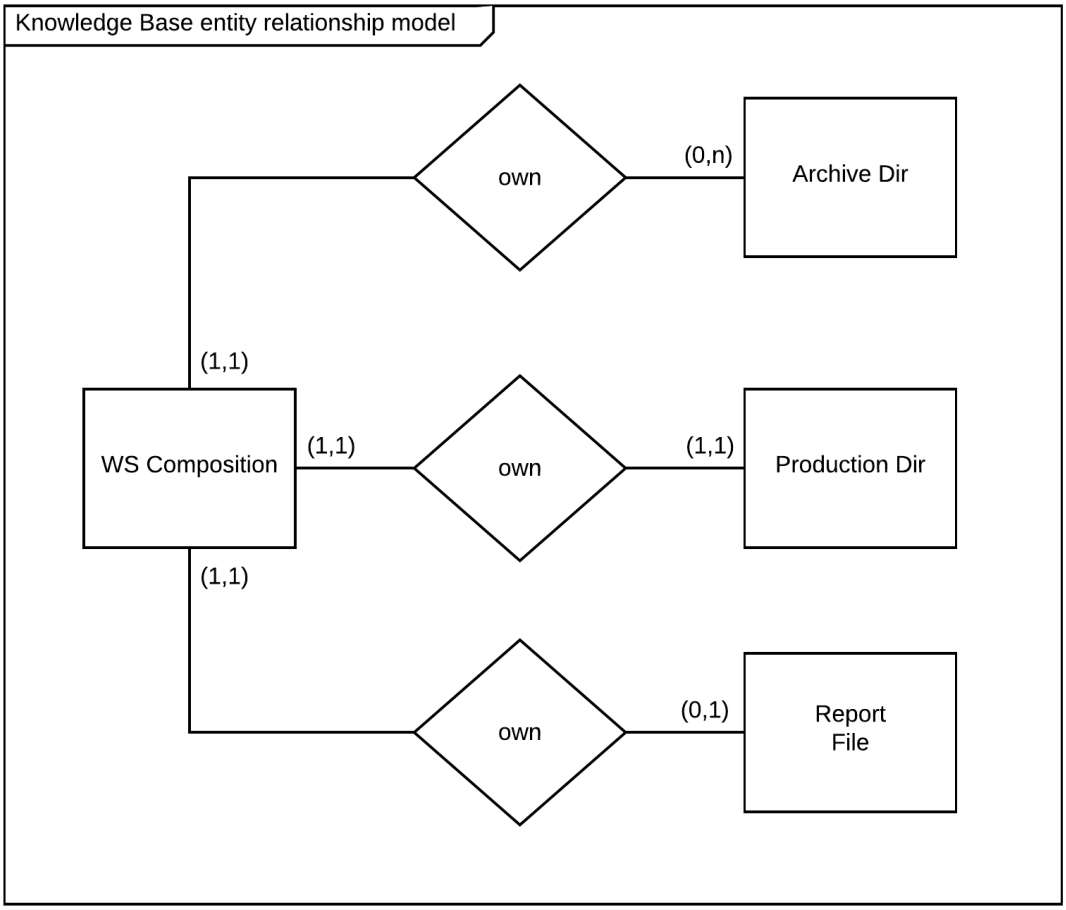


Figure 6.4: SAMBA Framework's Knowledge base entity relationship

6.4 Framework Project

6.4.1 Service Assemble Monitor

Service assemble monitor (SAM) is the component that periodically verifies if there is an update on one of the WS compositions monitored. Its behavior is described in the figure 6.5. This component runs the verification accordingly to a predefined frequency, in this work the time defined between executions was 30 seconds.

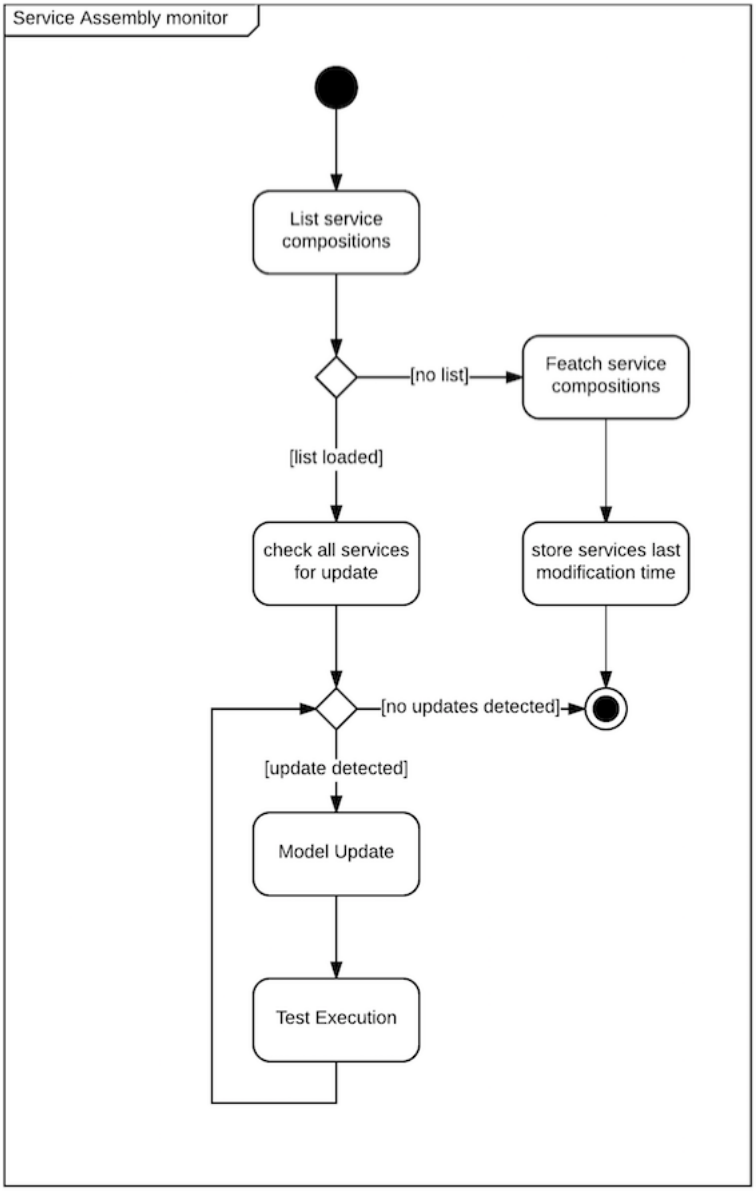


Figure 6.5: Service assembly monitor activity diagram

The first thing that this component does when it starts its execution is to list the WS compositions that are being hosted on the host server. Depending on the host server the path to the folder where the information about the deployed WS compositions is located changes, therefore, it needs to be defined by the user. If there is no listed compositions the component does a search on the host server and stores names and paths for the

WS compositions available, the last modification date information of each service is also stored. If the list of WS compositions is already available the component does a quick verification of the modification dates of the WS compositions available in the server and compares it to the date it has stored for each correspondent WS composition. In case of an update detection the component sends a message to the MG that triggers the model update for the correspondent WS composition, as the model update is finished the Test Execution message sent to the TS, when the test execution is finished the component continues to verify if there is any other WS composition that had an update.

6.4.2 Model Generator

The model Generator (MG) is responsible to handle all the process necessary to update and generate new models of the monitored WS compositions. The component manages copies of the WS description files required to generate a state model representation of the target WS composition described in the orchestration BPEL file.

The BPEL file has information about the services that belong to the WS composition, these files are downloaded and stored locally in order to enable the generation of preview WS composition designs if necessary, and also to keep the history of the WS composition evolution. In case of failures during the test case execution this information could be valuable to unveil the source of possible problems.

The MG has access to read and write in two folders in the host server knowledge base. The folder *Archive* is the folder responsible to store the information of the previews versions of the WS composition, the folder structure generated by the component is exhibited in the figure 6.6. The folder *Production* stores the current WS descriptions files used by the WS composition, described in the BPEL file.

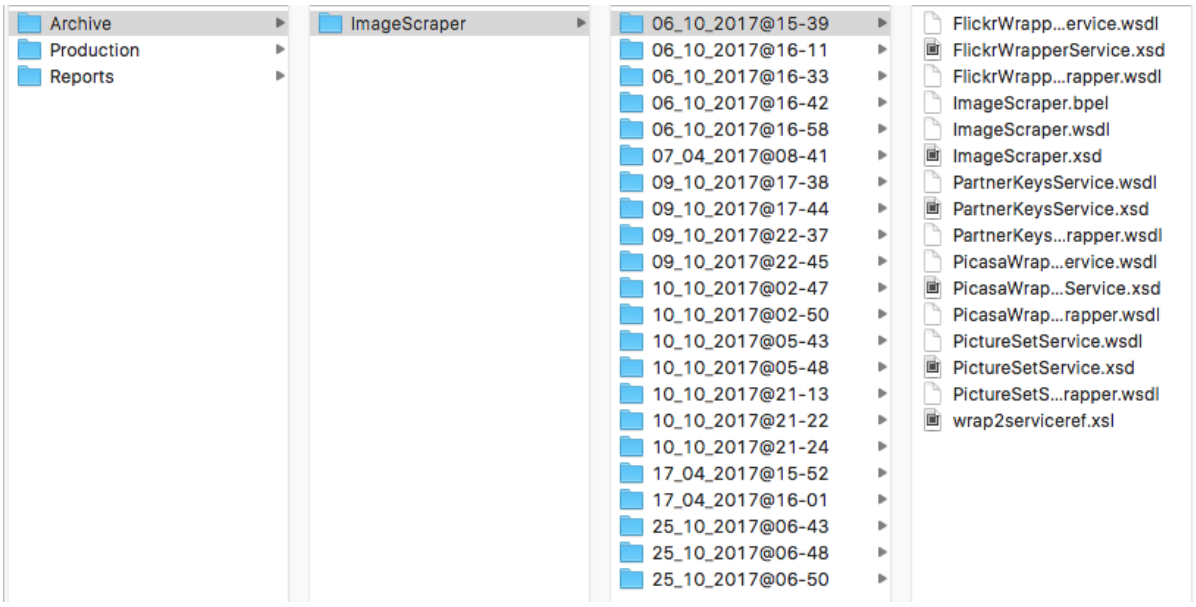


Figure 6.6: Folder structure managed by the Model Generator component

With the updated WS composition information locally available the MG can start the model generation process. This process is presented in the figure 6.7, which describes

all the steps required in generate a model version of a target WS composition. The *BPELUpdated* process starts checking if there is already a folder with the same name as the Target BPEL process. If there is already a folder with the same name, all of its content is archived, if there is no folder it is created. The BPEL file of the target WS composition is downloaded, which contains all the information about the services used by the WS composition, all the other service description files listed are also downloaded. The BPEL file is parsed to a XML format, than the information about business process described is extracted. the basis for the model are created, fitting the model format required for the MBOTCG component. The business process is converted to a collection of edges and vertices, that are later connected as the WS composition described in the BPEL file.

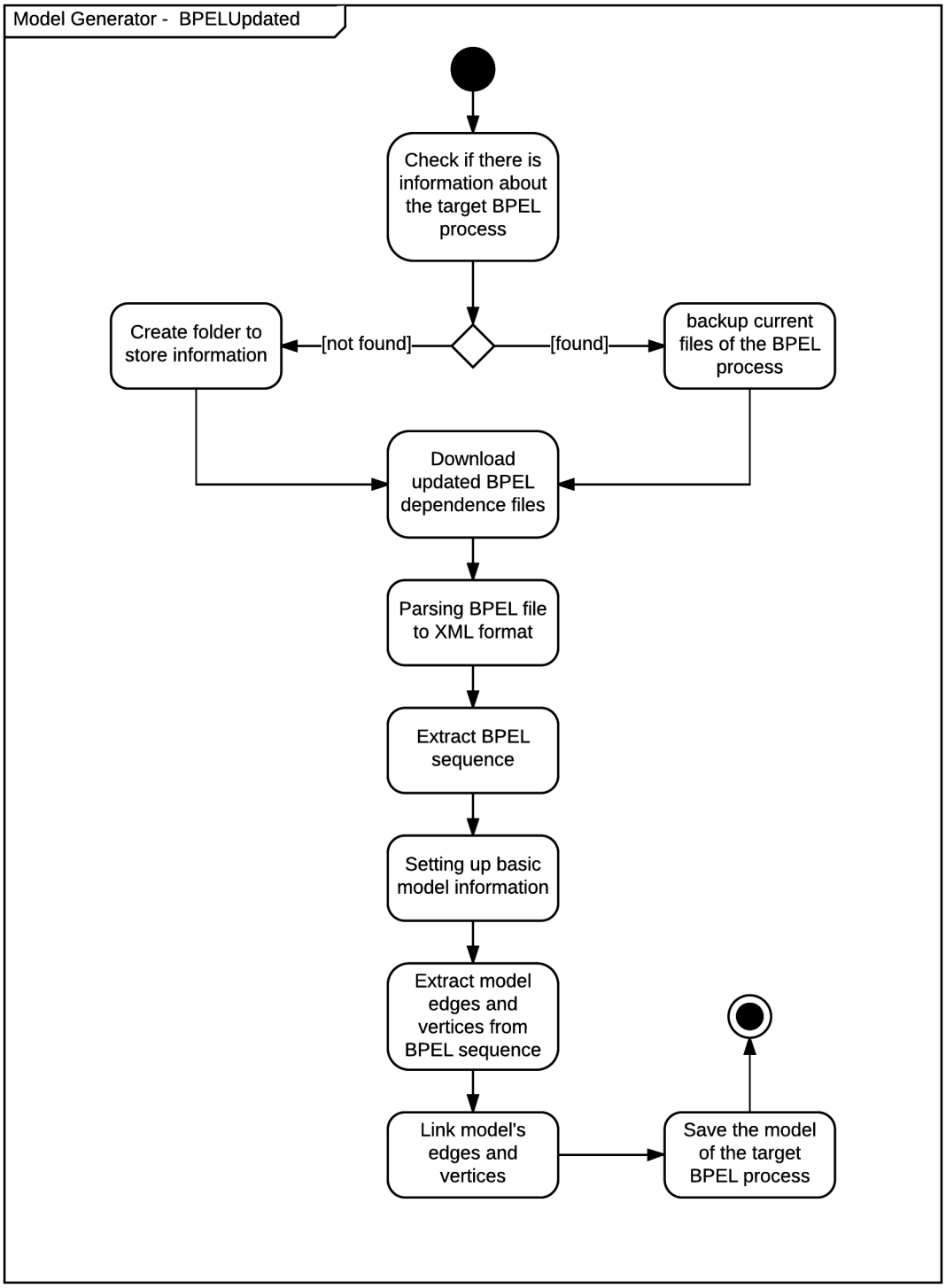


Figure 6.7: Model generation process

The model of the target WS composition is described using a json file, a model example is presented in the figure 6.8. A graphical representation of the model created using the information of the WS composition named ImageScraper, selected as case study in this work, is presented in the figure 6.9. This graphical representation was created using the model editor yEd [47].


```
{"name":"ImageScraper",
 "models":[{"name":"ImageScraper.bpel",
                      "id":"",
                      "generator":"random(edge_coverage(100))",
                      "startElementId":"e0",
                      "actions":[],
                      "vertices":[{"id":"n0","name":"v_0","properties":null},
                                  {"id":"n1","name":"v_1","properties":null},
                                  {"id":"n2","name":"v_2","properties":null},
                                  {"id":"n3","name":"v_3","properties":null},
                                  {"id":"n4","name":"v_4","properties":null},
                                  {"id":"n5","name":"v_5","properties":null},
                                  {"id":"n6","name":"v_6","properties":null},
                                  {"id":"n7","name":"v_7","properties":null},
                                  {"id":"n8","name":"v_8","properties":null}],
                      "edges":[{"id":"e0","name":"e_init","targetVertexId":"n0","sourceVertexId":""},
                               {"id":"e1","name":"e_ExploreFolksonomy@ext","targetVertexId":"n1","sourceVertexId":"n0"},
                               {"id":"e2","name":"getFolksonomyContent@picasa","targetVertexId":"n2","sourceVertexId":"n1",}
                               {"id":"e3","name":"shuffle@helper","targetVertexId":"n3","sourceVertexId":"n2"},
                               {"id":"e4","name":"get@key_registry","targetVertexId":"n4","sourceVertexId":"n1"},
                               {"id":"e5","name":"getFolksonomyContent@flickr","targetVertexId":"n5","sourceVertexId":"n4"},
                               {"id":"e6","name":"shuffle@helper","targetVertexId":"n3","sourceVertexId":"n5"},
                               {"id":"e7","name":"merge@helper","targetVertexId":"n6","sourceVertexId":"n3"},
                               {"id":"e8","name":"truncate@helper","targetVertexId":"n7","sourceVertexId":"n6"},
                               {"id":"e9","name":"e_ExploreFolksonomy@ext","targetVertexId":"n8","sourceVertexId":"n7"},
                               {"id":"e10","name":"e_loop","targetVertexId":"n0","sourceVertexId":"n8"}]}]}
```


Figure 6.8: json model generated by the model generator component

In the figure 6.9 the vertices hold the information required to perform test operations, the edges are used just to link the vertices in the same order as the operations would be executed in the business process. The edges of the model have their names as references to the services and operations used by the WS composition. this information is separated by the "@" character. The vertices of the model, represented by boxes, are used originally to perform assertions by Graphwalker during the test case generation, but in our approach this feature is not required, since the assertions are made by the Test Service's Testing Core.

Reserved strings are used in the model for two reasons: Some edges and all vertices in the model are required by GraphWalker in order to generate test cases, and there are edges representing operations that are not directly linked to the business process, like operations to store input and output parameters of the orchestration. Strings with prefixes "e_" and "v_", like "e_init","e_loop" and "v_INIT" are ignored in the current MBOTCG implementation, edges and vertices with that prefix does not generate test operations requests from the OTServ during the test case generation.

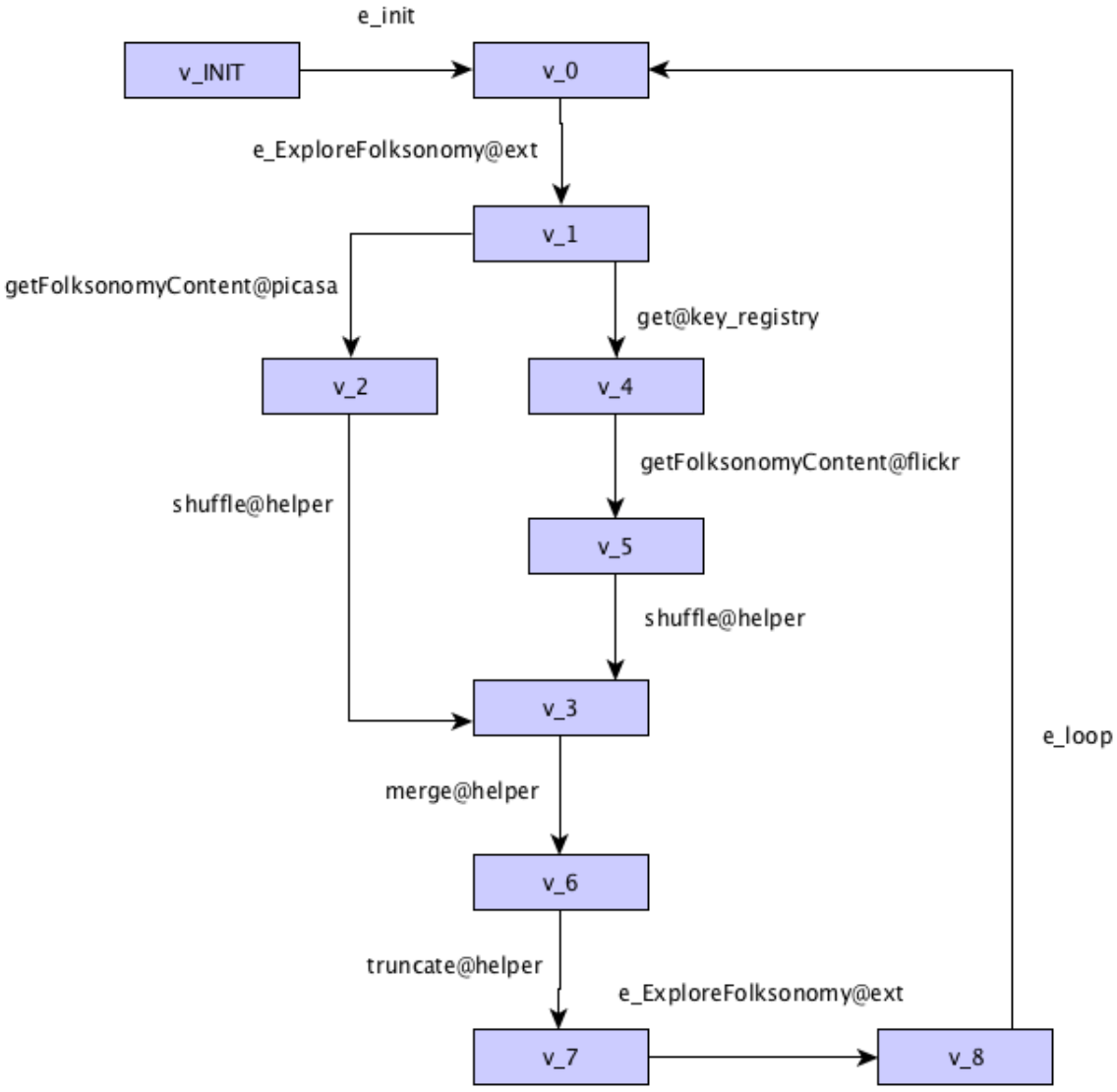


Figure 6.9: Visual representation of the model extracted from the ImageScraper BPEL file

6.4.3 Model-based Online Test Case Generator

The Model-based Online Test Case Generator (MBOTCG) Service has two components: Model-Based Test Case Generator (MBTCGen) and the Online Testing Service (OTServ). Both components are web services that interact using the REST protocol.

MBTCGen is a REST service running the version 3.4.2 of Graphwalker [31]. Graphwalker has built-in REST API with methods to load models, fetch data from the test case generated, restart or abort the test case generation, get and set data to a model if needed. MBTCGen is responsible to handle the model of the target WS composition and the test case generation process. The model, described in a json file, generated by the MG service, which used the information available in the BPEL file of the target WS composition.

The OTServ is responsible for sending the instructions, web service name and operation name, of test operation to the Test Service using SOAP messages. It receives a boolean as result for each test operation required during the test case generation, then updates the test report with the result, web service name and operation of the test operation performed. The figure 6.10 present an example of a successful test case report generated after a conclusion of a test case execution.

```
___TEST EXECUTION INFO___ Test execution date and time: Tue Oct 10 22:28:17 BRT 2017
Model used for test generation: ImageScraper.json
Model creation date: 09/01/2017
Path generator: random(edge_coverage(100))
___TEST CASE GENERATED___
test:result
get@key_registry : PASSED
getFolksonomyContent@flickr : PASSED
shuffle@helper : PASSED
merge@helper : PASSED
truncate@helper : PASSED
get@key_registry : PASSED
getFolksonomyContent@flickr : PASSED
shuffle@helper : PASSED
merge@helper : PASSED
truncate@helper : PASSED
getFolksonomyContent@picasa : PASSED
shuffle@helper : PASSED
```

Figure 6.10: test report generated by the Model-based Online Test Case Generator

OTServ is also responsible for the setup of the MBTCGen, allowing the user to set parameters needed for the test case generation like test case generator , stop condition and model. It also triggers the start and stop of the test case generation. The OTServ converts the output of the MBOTCGen, since its output are REST messages, it is necessary to convert it to SOAP format in order to make the interaction easier with the Test Service. During the execution of a test case generation the OTServ stores information like execution date, execution time, name and creation date of the model used to generate the test case, the test case generator (the algorithm used to traverse the model and generate the test cases) and stop condition (the condition to stop the test case generation). There is a list of generators and stop conditions available on Graphwalker website [31]. When the test case generation and execution stops, a report with the information can be generated if requested, instructions and results of the test operations performed during the test case are stored in a report file with all the other data collected by the OTServ, as presented in the figure 6.10.

6.4.4 Test Service

The architecture of the Test Service component implemented is described in the figure 6.3. There are several internal components, most of these components were inherited from the original Testing Service architecture proposed by [12], which was originally designed to execute robustness tests of services at runtime. Such architecture comprises the following set of components: Testing controller, Testing BPEL, Within Reach Service, Controlled Service, BPEL Service and Testing Core.

Testing controller is responsible for the automatic deployment and undeployment of copies of required WSes and databases if necessary, redirecting the test operations to the copies, allowing the online test execution without interfering the runtime environment. Testing BPEL is responsible for extraction of the list of services required for the business process to be executed, it also manages the deployment and undeployment of copies of BPEL services if they needed to be tested. Services that are part of the WS composition, known, but not controlled by the provider, are tested as external services by applying black-box tests from the testing core. Controlled services and BPEL services can be tested as white-box, through the Testing BPEL and the Testing Controller. The Testing core is responsible for guiding the test activities: selection policies, merging results, tracks evolution of services and manages the testing BPEL and testing controlled [12].

In order to achieve the test framework goal, changes were introduced on the original Testing Core components, these changes enable the component to i) perform functional test and of robustness tests on the WS that it has access ii) interact with the other three new components of the SAMBA framework by new WS operations.

Original Testing Core component

In the original version of the Testing Core component any time a test was performed the component would check the status of the target system and its relationship with the service owner. There were three levels of relationship defined: *controlled service*, a service own by the provider and all its internal information; *within reach service*, a service that is part of the SOA and available to be used by the provider, however the provider has no control over it; *unknown service*, are services that are part of the SOA, but the provider is not aware of its participation [11].

Depending on these information a different test approach would be used, if the service was online and was a controlled service (BPEL or not) a copy of the respective target application would be deployed, since the service is controlled by the provider there is enough information available and white-box tests are feasible. If the service was offline the test would be executed directly to it without jeopardizing any users of the service. If the service is not under control of the provider and is listed as a within reach service it is tested directly by the testing core by applying black-box tests through the WS interface. The test results are integrated and stored by the testing core on a database.

Upgraded Testing Core component

The component received five upgrades: a testing Oracle and four new WS operations. The testing oracle is an add-on that enable the testing core to perform functional black-box tests to services that are part of any WS composition of the provider.

The test oracle receives the test case instruction from the MBOTCG, decodes the instruction, by separating the the name of the target service and the target function to be tested. It verifies if the service is known and available to be tested, than it requests a test execution of the required operation. It checks the result of the test and returns to the MBOTCG if the test was a success or a failure.

The four new WS operations are designed to enable the execution of tests and the share of information needed in order to complete the whole self-adaptive loop of the SAMBAframework, the operations are: *getDependenciesPathForBpelName*, *fetchDataFromFilePath*, *exeServiceTestOperation*, *startOnlineModelbasedTestForBPEL*.

The operation *getDependenciesPathForBpelName* is used by the MG in order to get the list of dependence files required by the target BPEL composition. The operation *fetchDataFromFilePath* is used by the MG to download a specific file in order to store it to be used as a local reference for the model generation process. The operation *exeServiceTestOperation* is used to perform a single test over a target WS operation.

The operation *startOnlineModelbasedTestForBPEL* manages the full test case generation and execution on a target WS composition. The process is presented in the figure 6.12. The operation checks if the MBOTCG is available, loads the model and resets the test oracle and the variables needed to reproduce the business process of the target WS composition. It starts the test case generation loop and keeps track of the test case size generated. The test case loop is finished if the test case size is bigger than the max size defined for a test case (this was added in order to prevent deadlocks during the test case generation), or if the test MBOTCG reaches its determined stop condition, or if the result of the performed test is negative. The process is finished in two possibilities, a request for abortion or stop for the test case generation, the difference between them is that during an abortion the test report will present a message informing that the test case was aborted, on a regular test generation stop request the report does not present this message, the figure 6.11 exhibit a aborted test case execution performed by the operation.

```
___TEST EXECUTION INFO___ Test execution date and time: Tue Sep 05 05:50:41 BRT 2017
Model used for test generation: ImageScraper.json
Model creation date: 09/01/2017
Path generator: random(edge_coverage(100))
___TEST CASE GENERATED___
test:result
getFolksonomyContent@picasa : PASSED
shuffle@helper : PASSED
merge@helper : PASSED
truncate@helper : PASSED
getFolksonomyContent@picasa : PASSED
shuffle@helper : PASSED
merge@helper : PASSED
truncate@helper : PASSED
[Test Case Aborted] Stop condition not reached : FAIL
```

Figure 6.11: Test report generated after startOnlineModelbasedTestForBPEL execution finished with an abortion request

The figure 6.12 exhibit the test case generation loop, there are three situations on the activity diagram that are related to the conditions to finish the test case generation process. The first one is related to the size of the test case being generated, if the test case reaches a predefined max size the test case generation is aborted. The second is related to the stop condition defined during the load model process, this stop condition is always related to the traverse information of a model, which could be about the number of vertices or edges reached during the test case generation, execution total time or a specific edge or vertex name. When the stop condition is reached the Model-based test case generation tool informs that that are no more test to be performed, this triggers the test case generation stop process. The third is related to a negative result of a test operation during the test case generation process, this event triggers the test case generation stop process. All these situations result in the generation of a test report, this report contains all the test operations and results in the execution order, so the test case could be replicated if necessary.

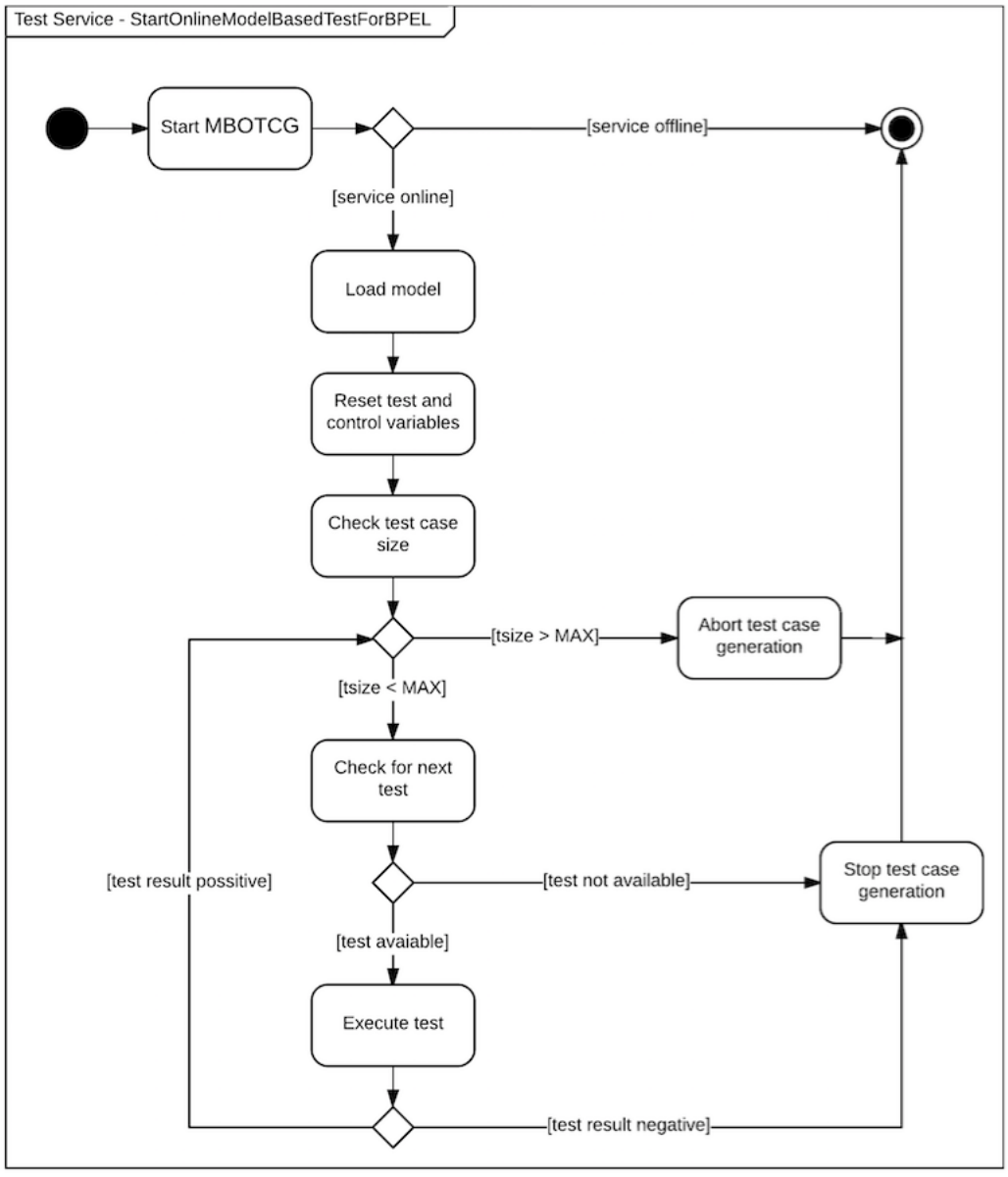


Figure 6.12: startOnlineMoldelbasedTestForBPEL operation activity diagram

6.5 Framework Implementation

The MBOTCG, MG and Knowledge base were hosted in a mac OS X EL Captain, which is the main OS of the host machine (MacBok Pro 13-inches late 2011). The SAM and TS with the orchestration used in the case study, were hosted in a virtual machine (VM VirtualBox 5.1) running an Ubuntu OS 10.11.6. The communication between services was enable by the use of a virtual network bridging the two operational systems.

The Service Assemble Monitor (SAM), Test Service (TS) source and all its components codes were written in NetBeans 6.7.1, compiled with java 1.6, and hosted in a GlassFish 2.1.1 Server. These versions of Netbeans and GlassFish server come with a BPEL editor and BPEL engine that can run the case study. The Model Generator (MG) and Model-Based Online Test Case Generator (MBOTCG) were developed in NetBeans 8.2, compiled with java 1.8, and hosted in a GlassFish Pyrana 4.2 Server. The figure 6.13 present the UML deployment diagram for the framework implementation.

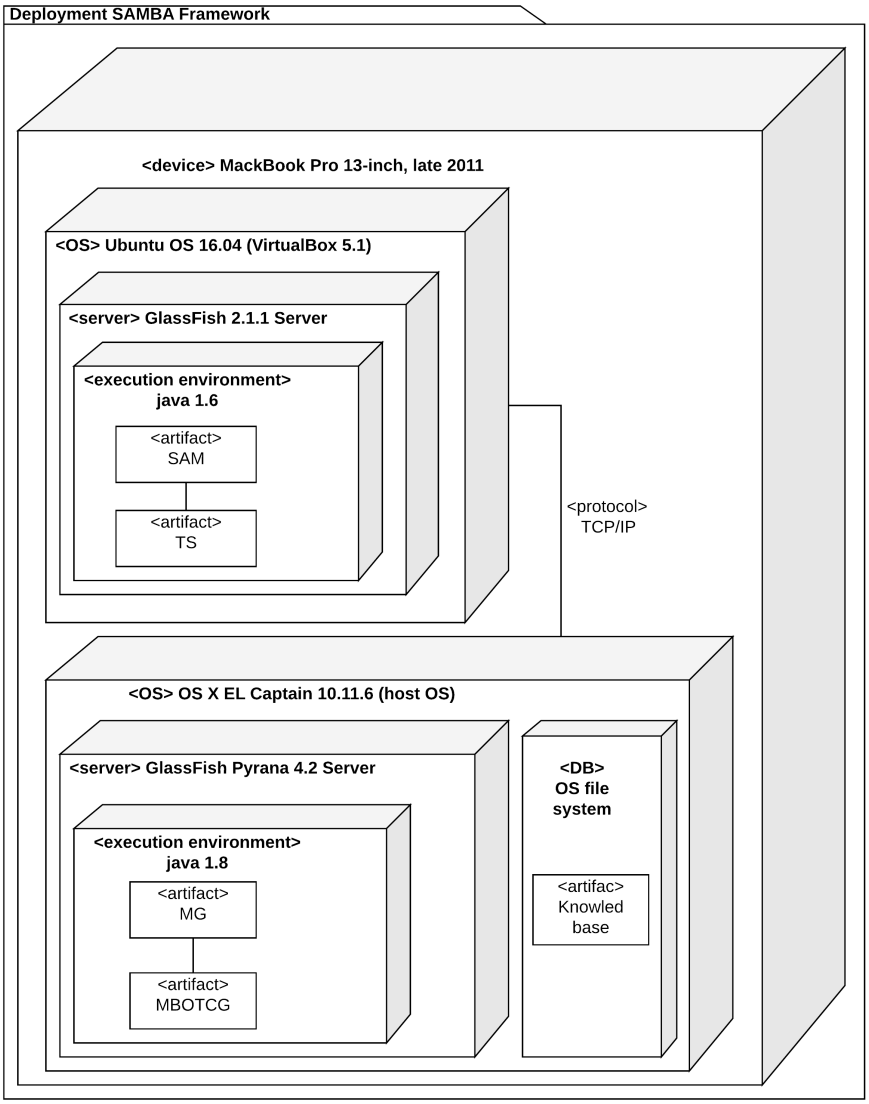


Figure 6.13: SAMBA Framework UML deployment diagram

Tools and Libs

The SAMBA Framework relies on several preexisting solutions, open-source and proprietary ones. The table 6.1 presents the tools used for the project deployment, and the table 6.2 presents the public libraries and versions used in the project.

Tool	Version
NetBeans	8.1
NetBeans	6.7.1
Payara Server	4.1.1
GlassFish Server	2.1
Graphwalker	3.4.2
Yed Graph Editor	3.17.1

Table 6.1: Tools used to implement the project

Library	Version
commons-io	2.5
jackson-annotations	2.6.7
jackson-mapper	1.9.2
jackson-databind	2.6.3
jackson-core	2.6.3
javaee-web-api	7.0
webservices-rt	2.3
javax-annotation-api	1.2-b03
javax-ws-rs-api	2.0.1
jersey-client	1.19.2
json	20160212
json-simple	1.1.1
hamcrest-core	1.1
jsr311-api	1.1.1
junit	4.10

Table 6.2: Libraries used to implement the project

Chapter 7

Experimental evaluation

The SAMBA Framework was designed to perform automatic WS composition model-based online regression tests, using the monitored WS composition's BPEL files to generate models which will be used to generate tests cases. Each WS component of the framework plays an important role in the self-adaptive control loop, and contribute directly to the achievement of the framework's designed goals.

It is important to validate the framework's behavior and check if its implementation meets to all the requirements set for the project, as described in the preview chapter. It is also important to define research questions for the project experimental evaluation. These questions will be the cornerstone of the test plan that will be designed to evaluate project implementation.

During the development of the framework's components unit and integration tests where exhaustively performed, however these tests were designed to attend the service developer perspective. Test cases in the experimental evaluation must be designed to also answer quality requirement of the framework in the perspective of service integrators.

The SAMBA framework has been applied to the jSeduite [15], a free SOA application that deals with information broadcast inside academic institutions. In total it is composed of 31 JAX-WS (Java API for XML Web Services) Web Services representing information sources and 8 BPEL orchestrations expressing business processes. These services access data stored in a MySQL database. Some of these services involve external WSs and applications, like Flickr, Twitter, Picasa, RSS Feed Services, News services, weather forecast services. Details on the role and scope of each service can be found in [15].

The JSeduite BPEL orchestration selected to evaluate the SAMBA framework is named ImageScraper. This orchestration invokes two services that login in two different well-known image hosting services (Flickr and Picasa), then it includes services to perform searches for images that match a desired collection of tags, and returns a shuffled determined size collection of pictures from both services.

Imagescraper is one of the biggest and more complex BPEL orchestrations available on JSeduite. It involves five different WSes, performs invocations of 7 different WS operations, and has a parallel invocation of operations in the business process.

This chapter has the following sections: test plan, which will present information about how the test plan was designed; test case settings and execution, which will explain how the test cases were setup and how the results were collected; and test results, presenting

the results of the test cases execution and information about the performance of the framework.

7.1 Test plan

To validate if the SAMBA framework is able to achieve its goals the test plan target its requirements and the core features. The two core features are: the self-adaptive control loop and the regression test execution. In order to validate the two core features a test environment was set and three test scenarios were designed. The test plan stress the framework and check if the requirements set for the project were achieved as defined in the chapter 6.

7.1.1 Research Questions

The implemented framework, if able to achieve its requirements, will raise some relevant questions related to its use, effects on other applications and about the regression tests generated. The test plan covers the follow research questions:

- **How tests would interfere in the application during runtime?** - The SAMBA Framework performs online regression tests, so there is the possibility that these tests might affect the behaviour or the target WS composition, causing problems on WS operation replies or forcing the Ws composition to an unpredicted and undesired state. It also might interfere on other WS applications on different levels (e.g., consume too much computational time or memory).

- **How to generate meaningful tests in case of service change?** - The SAMBA Framework model-based test case generation tool uses random algorithms to generate the test cases, therefore it is necessary to determine what is a "meaningful test case". The framework produces regression tests, in this context a "meaningful test case" means that a test case is able to stress the modifications in a target SUT.

7.1.2 Environment

The SAMBA Framework was exercised on a BPEL orchestrated WS composition named ImageScraper. Measurements of the execution time, resource consumption, the size of the test case and the failure detection capabilities were stored in order to later be analyzed. The execution duration time is measured from the the beginning of a test case generation until its completion. The test case size is measured by counting the number of test operations executed during the test case generation. The test case generation is finished if a stop condition is reached or a failure is detected.

Resource consumption is measured using the system management tools available on the two OSes (OS X and Ubuntu) used in the experiment. Using the OS X system monitor tool it is possible to track the processes that are related to the services involved in the test scenario. Information about memory and CPU consumption during the execution of the test cases. In the Ubuntu OS it is possible to run an operation using the terminal called

"top", which exhibit information equivalent to the OS X system monitor tool about the current system process. Finally, failure detection capability is simply measured checking the reports of the test cases executed for each scenarios described.

7.1.3 Scenarios

Two test scenarios were designed to validate the SAMBA Framework online regression test execution feature. Both test scenarios focus on the detection of orchestration problems. There are many possibilities that could lead to an orchestration failure. The test scenarios were designed to stress two types of information that the model holds about the composition: WS names and WS Operation names.

One test scenario aims to evaluate SAMBA Framework's self-adaptive control loop, that involves the capability to detect changes in the monitored WS compositions, trigger the model generation, test case generation and execution. The test scenario's metrics and their relation with the goals of the experimental evaluation are detailed next.

Unreachable WS

In this test scenario one WS of the ImageScraper orchestration have its name updated and then redeployed to the server. The average time to perform the regression tests, the average size of the test cases generated, and the test results (stored in the report file) were used to evaluate the SAMBA Framework regression test execution capability. This test scenario was repeated with the same settings and with the same WS updated, More precisely, the service connecting to Flickr was selected to be updated. The quantitative data generated from the repetitions was used in an statistical analysis of the SAMBA Framework functional behavior.

WS operation fail

In this test scenario one WS operation of the ImageScraper orchestration have its name updated and redeployed to the server. The average time to perform the regression tests, the average size of the test cases generated, and the test results (stored in the report file) were used to evaluate the SAMBA Framework regression test execution capability. This test scenario was repeated with the same settings and with the same WS operation updated, more precisely, the *merge* operation of the "Helper WS" of the ImageScraper orchestration was choosen to be updated, presented in the figure 6.9. The quantitative data generated from the repetitions was used in an statistical analysis of the SAMBA Framework functional behavior.

WS composition update

In this scenario the ImageScraper orchestration was updated in the server, BPEL file mutants were updated in the Host Server, simulating the WS orchestration update. The mutant files were generated using a tool called muBPEL, these mutants are slightly modified (mutated) versions of the original program in which a syntactical change has been made [18]. Three mutant BPEL files were selected randomly and used in the repetition

of the test scenario. The orchestration models and the test reports generated by the regression test execution were used as qualitative information to evaluate the the SAMBA Framework's self-adaptive capabilities.

7.2 Test scenarios settings and measurements

For all the case studies, the stop condition used in the model-based test case generation tool was 100% coverage of the edges of the composition model.

The scenarios "unreachable WS" and "WS operation fail" were repeated 500 times each, these scenarios targeted the SAMBA Framework online regression test execution feature. This was done in order to evaluate the framework performance using statistical analysis. The experiments were executed by a script, where the different parameters are configured, for example the number of tests to execute, or the name of the target WS orchestration, both scenarios generated quantitative data about the SAMBA framework.

The scenario "WS composition update" did not require execution repetitions, because its goal was to validate the Framework's self-adaptive control loop feature. The scenario performed the execution of three updates in the ImageScraper BPEL orchestration using mutations of the original BPEL file. The test cases generated during the three updates used the same configurations and parameters used in the other test scenarios. This scenario generated qualitative data about the SAMBA framework.

Two files are generated during the online test case generation and execution: a *test report* and a *test meta*. The *test report* is a text file that describes the test case generated and the result of its execution. The *test meta* is generated for statistical analysis, it is a Comma Separated Value (CSV) file that stores the test case size, test result and the computational time required to perform the test case.

The table 7.1 present information about the tests scenarios. The column "Generation" is related to the test case generation Algorithm (ALG), all of them are based on random (RAM) generation, and stop condition (StpCond). The column "Execution" is related to the number of repetitions (REP) and use of mutants (MUT) for the test scenario. The column "Measurement" is related to the type of result produced by the test scenario, which could be quantitative (Quant) or qualitative (Quali), and the target feature of the SAMBA Framework, which could be the regression test (RegTst) or self-adaptive (SfAdpt) capabilities.

Test scenarios settings and measurements						
	Generation		**Execution**		**Measurement**	
Test scenario	**ALG**	**StpCond**	**REP**	**MUT**	**Type**	**Target**
Unreachable WS	RAM	100% cover.	500		Quant	RegTst
WS operation fail	RAM	100% cover.	500		Quant	RegTst
WS composition update	RAM	100% cover.	1	✓	Quali	SfAdpt

Table 7.1: Test scenarios settings and measurements information

7.3 Test results

Following are the results for the three test scenarios. To analyze the dispersion of the values, we selected the Poisson distribution. This is popular for modelling events that occurs a discrete number of times in an interval of time or space. Since the results of test case size generated are discrete and there is no variables lower than zero, we can consider the lower bound for the data sets generated for these scenarios to be zero. The time duration of the test cases generated is classified as discrete, there is no test case execution with negative time duration, so to simplify the analysis the data sets were clustered in a number of intervals equals to the number of intervals used in the test case size chart. This way both data can be analyzed as discrete and with a lower bound of zero, making Poisson distribution the best choice to analyze the results.

For the two first scenarios, it is presented two histograms for each scenario, one presenting the distribution of test case size generated and one representing the distribution of test cases duration time. Statistical analyses of the results of each scenario are presented in a table. Information about computational load and overall result are presented as well. The results for these scenarios is presented following the order of section 7.1.3.

The third scenario present the results of the three mutations tested using the SAMBA framework self-adaptive capability, the results are cross-analyzed with the mutant BPEL files and their respective models generated.

7.3.1 Unreachable WS

All test cases generated in this scenario were able to detect the nonconformity of the orchestration, the test reports revealed failures while trying contact the renamed WS. The statistical analysis of the two resulted data sets are presented in the table 7.2.

Statistic	**Time (milliseconds)**	**Size (nº of operations)**
Mean	1142.74	6.18
Median	943	6
Mode	341	2
StdDeviation	33.80	5.54

Table 7.2: First scenario's information summarized

The mean values of both data sets are bigger than its respective medians. The low values for the mean in both data sets, and fact that they are lower than the medians is a sign of concentration of values close to the lower bound. The test case duration time histogram, presented in the figure 7.1, and test case size histogram, presented in the figure 7.2, show that both distributions are concentrated in the first two interval groups, the histograms also show that the use of a Poisson distribution to analyze the results is accurate.

The Pearson correlation coefficient [3] of both data sets resulted in an r value of 0.9782, which means that both data sets are very close to a total positive linear correlation. With the result it is possible to say that a test case in this scenario has 80% probability of having a size less than 15 and also 80% probability of duration time less than 2613 ms.

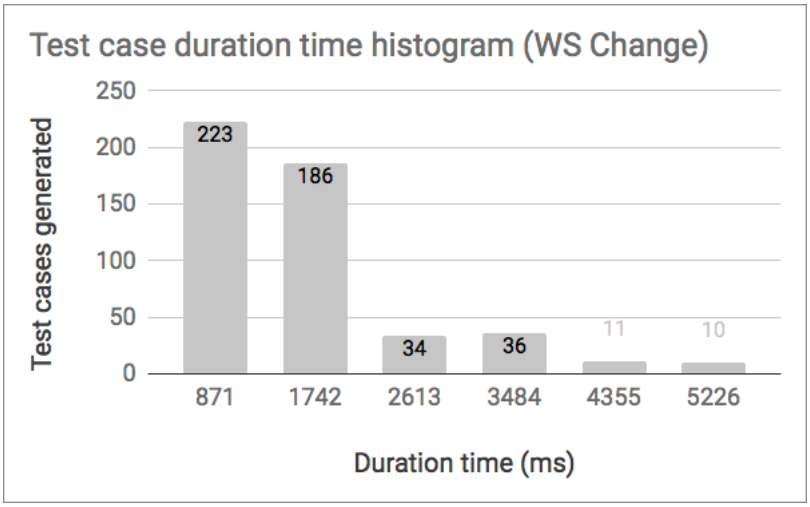


Figure 7.1: Test case duration time histogram of the first scenario

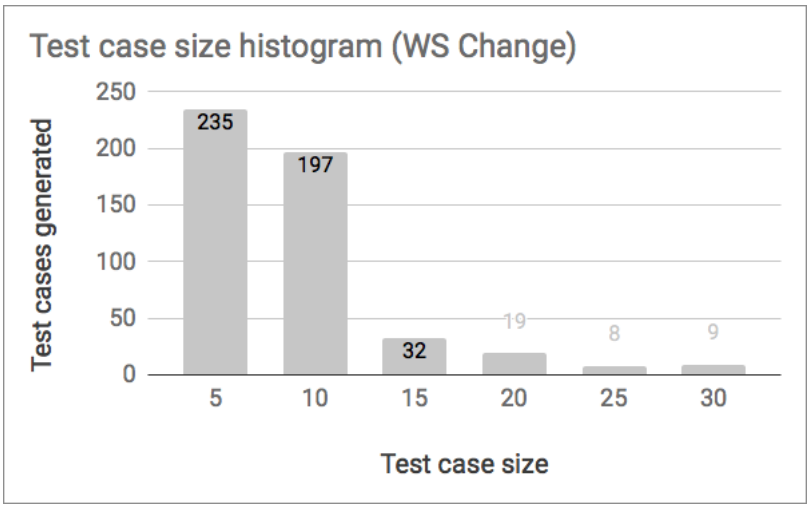


Figure 7.2: Test case size histogram of the first scenario

7.3.2 WS operation fail

All test cases generated in this scenario detected the nonconformity of the orchestration. A failure is reported when trying to execute the WS operation that was deactivated. Statistical information are presented in Figure 7.3.

Statistic	Time (milliseconds)	Size (nº of operations)
Mean	541.4	3.52
Median	489	4
Mode	401	4
StdDeviation	23.26	0.49

Table 7.3: Second scenario's information summarized

The mean values for test case duration time is bigger than the median and mode. The histogram of this data set, presented in the figure 7.3, shows a distribution concentrated in the first two interval groups. Figure 7.4 shows that there are only two sizes of test cases.

The Pearson correlation coefficient of both data sets resulted in a r value of 0.2793, which means that both data sets are very close to a total positive linear correlation. Observing the results of the scenario and both histograms present, it is possible to say that for this specific scenario there is 80% probability that the test case duration time will be less than 840 ms and 100% probability chance to have its size less than 4. Since the test case duration time was smaller than the other results it is possible that for this scenario the test environment interfered with the results.

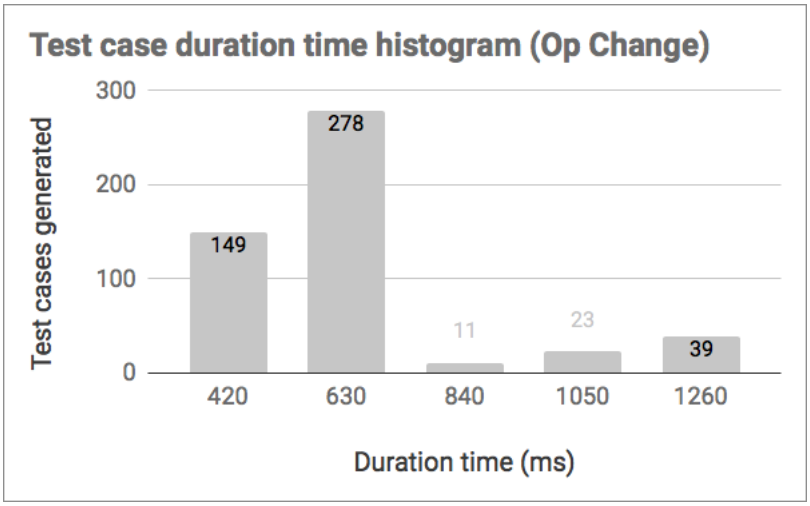


Figure 7.3: Test case duration time histogram of the second scenario

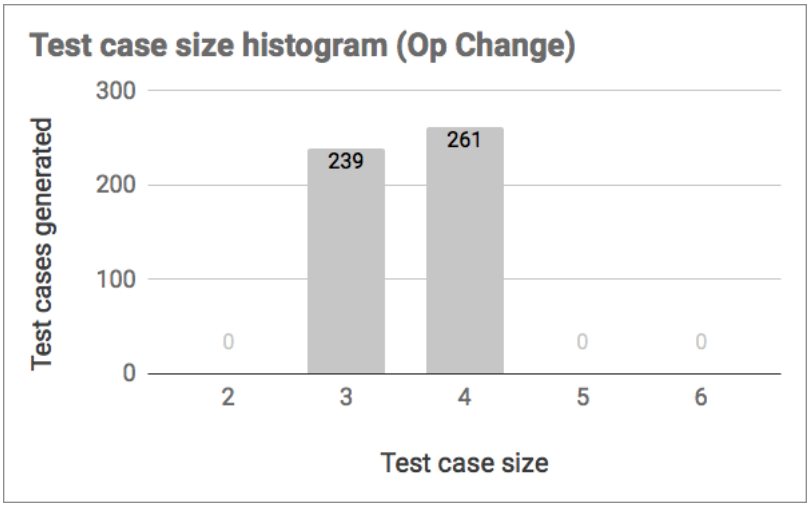


Figure 7.4: Test case size histogram of the second scenario

7.3.3 WS composition update

Using the muBPEL tool it was possible to generate more than two hundred different orchestrator mutants, three mutants were selected randomly. The SAMBA Framework was deployed fully and tested. The following figure 7.5 is a graphical representation of the original BPEL file, used to generate the models and deployed on the Ubuntu virtual machine. All three mutations were correctly detected, the files on the knowledge base were all archived properly, the model generation and the online test case generation were correctly executed.

The first and second mutants deployed on the server are presented in the figures 7.6 and 7.7. The composition regression test was performed and no problems were detected. The third mutant presented in the figure 7.8 got its test generation aborted since the test case size surpassed the determined test case max size allowed. The SAMBA Framework was able to perform and complete its task in all three mutants test.

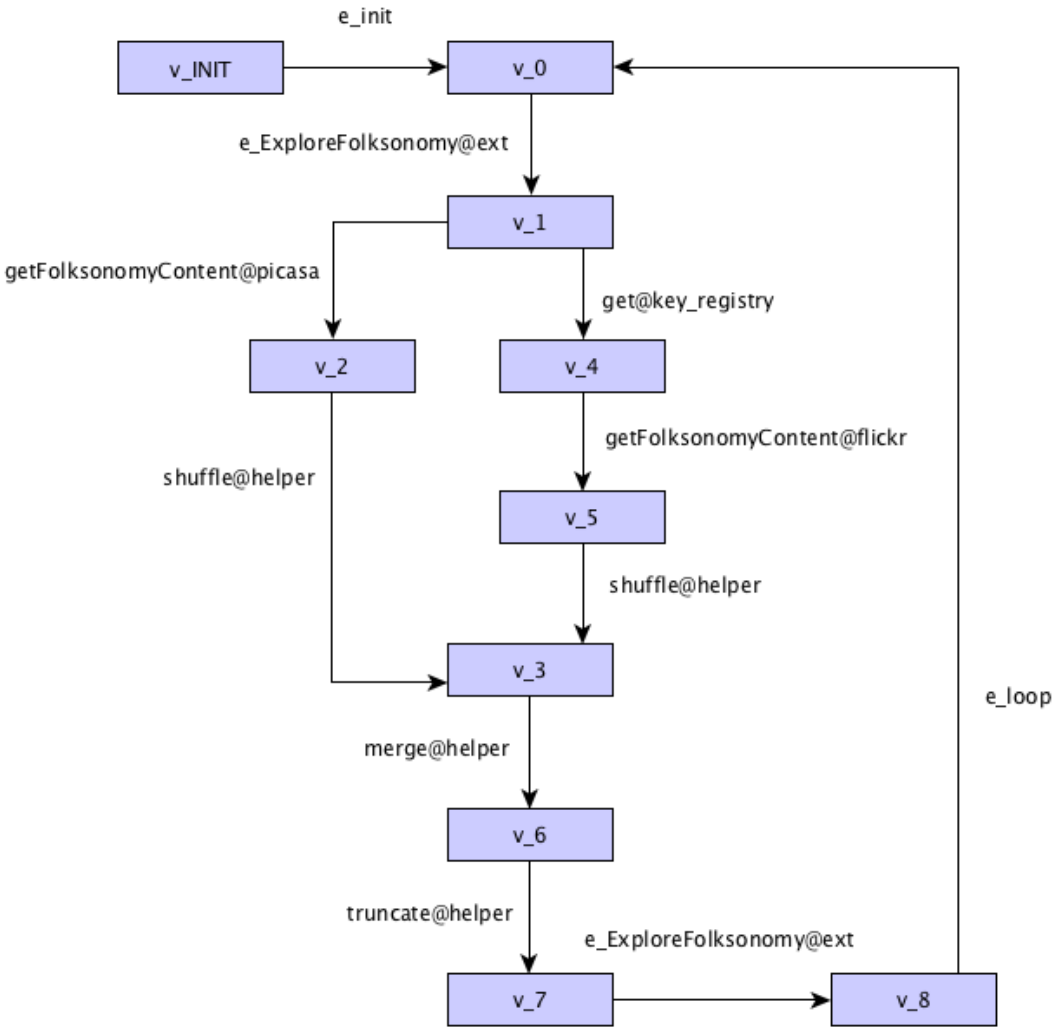


Figure 7.5: Model generated for the ImageScraper BPEL file

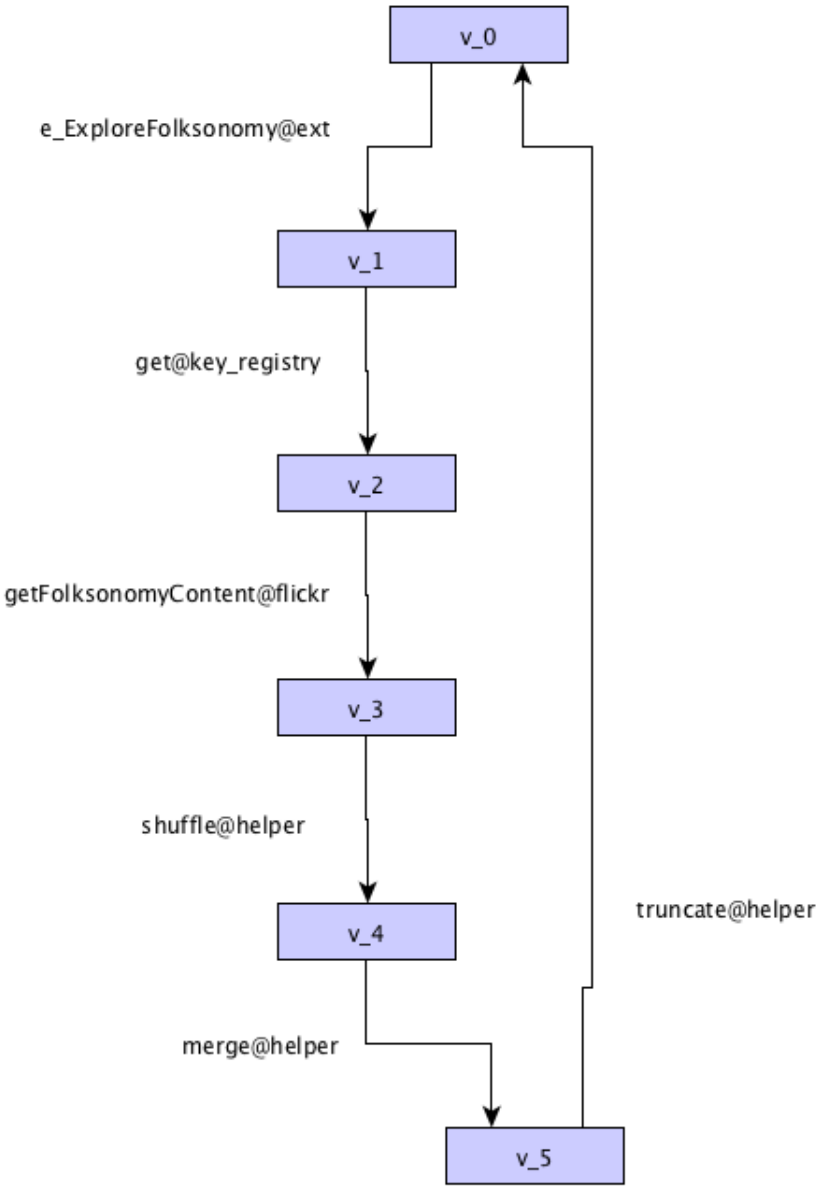


Figure 7.6: Model generated for the first ImageScraper mutant BPEL file

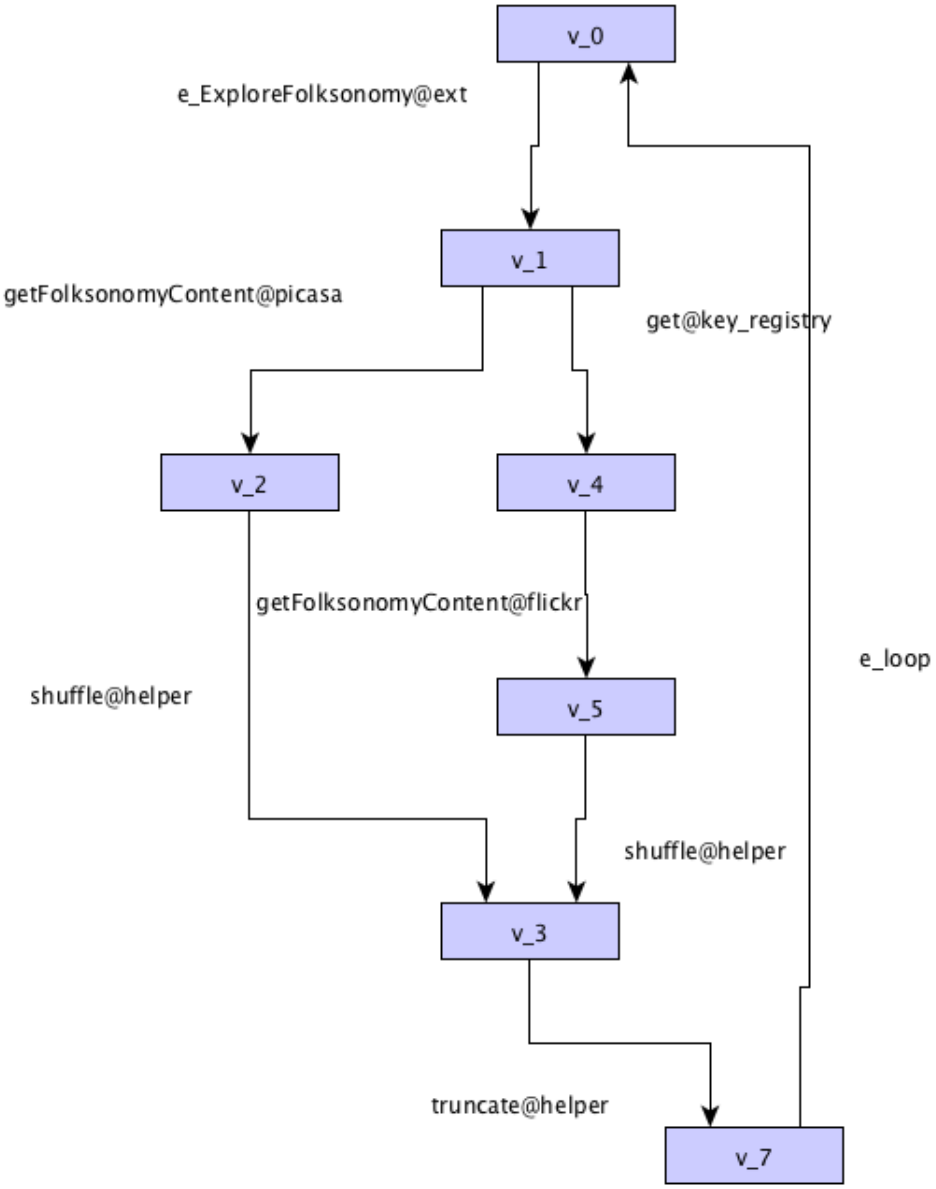


Figure 7.7: Model generated for the second ImageScraper mutant BPEL file

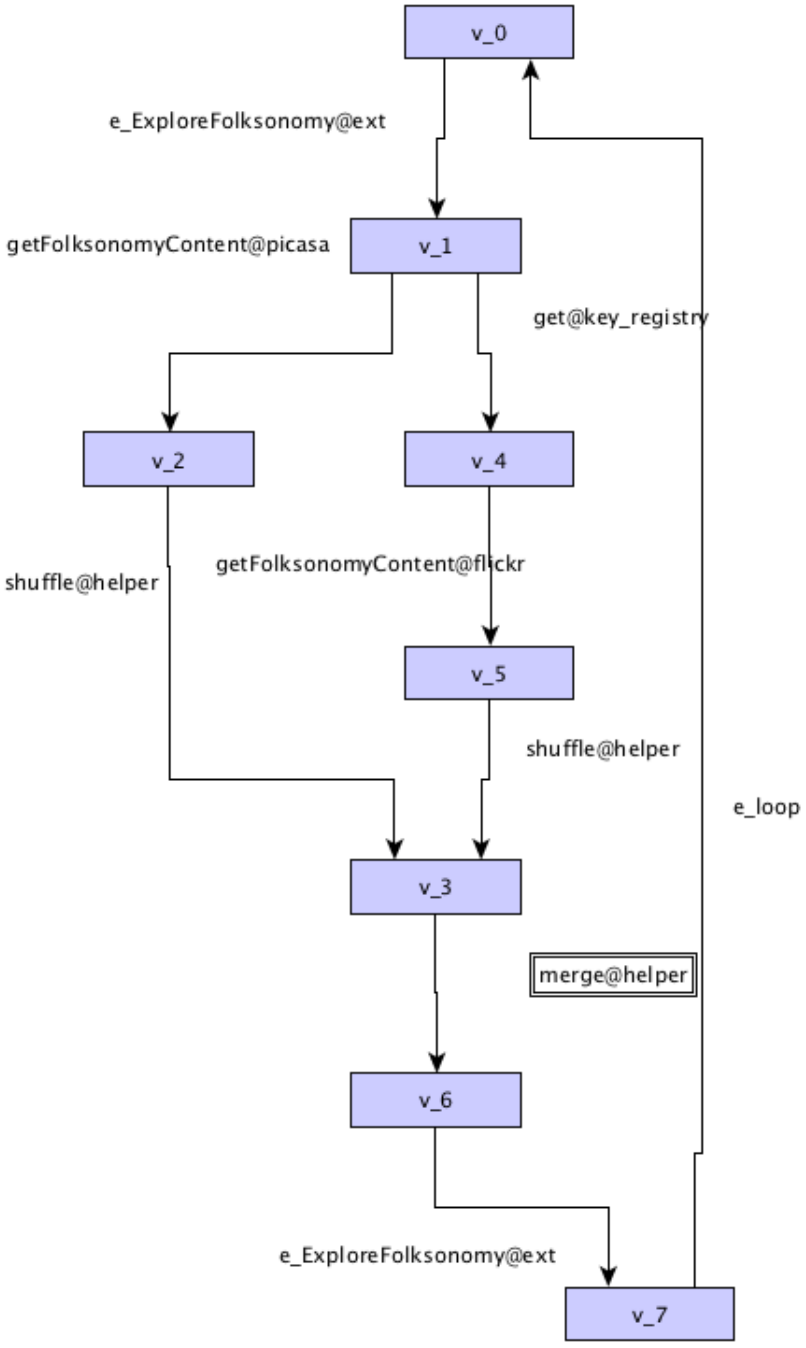


Figure 7.8: Model generated for the third ImageScraper mutant BPEL file

7.3.4 Resource consumption

In all scenarios the SAM consumed 4Mb of RAM an 5% of the CPU load, the MBOTCG consumed 181Mb maximum of RAM and was responsible for 4.2% of the CPU load. The MBOTCG consumed 850Mb maximum of RAM and was responsible for 15% of the CPU load. The Test service was responsible for the consumption of 350Mb of RAM and 40% of the CPU load.

7.3.5 Results and Research questions

- How tests would interfere in the application during runtime?

  Depending of the type of test being performed different effects in the application can appear. Since only composition regression test was contemplated in the test scenarios, it is not possible to answer this question completely. The tests performed by the SAMBA framework did not affect the performance of the WS that belong to the WS composition, no failures or problems were reported during the test case generation process, not even during the first two scenarios, that performed together one thousand test case generations.

- How to generate meaningful tests in case of service change?

  Since the path generation algorithm used by the MBOTCG was limited, all the scenarios used a random path generator, with the same stop condition (100% edge coverage). The test cases results and data analysis showed that because the random path generation property it was impossible to guarantee that the test cases generated would be optimal (property of a test case to execute the minimum number of test operations to reach its stop condition) or meaningful (property of a test case to stress only the modifications in a target SUT). However the test cases generated covered all the functions of the models, which means that the modifications were tested. The analysis of the test cases generated showed that, for the target WS composition, there was a high probability that the size of the test cases generated would be minimal to cover all the model, which could be understood as the test cases generated performed only the necessary operations to reach the stop condition with minimal repetition.

7.3.6 Threats to validity

The SAMBA Framework does not require much computational resources, and performed expected as designed. It was able to perform online regression composition tests, automatically update models from BPEL orchestrator files and generate test cases from these models.

The model generation process was able to extract enough information from the orchestration description files to enable the SAMBA Framework to perform composition regression tests. The information used to generate the models were the orchestration operation sequences, WS names and WS operations available in the BPEL files. Because of that the BPEL process was not replicated perfectly on the model, characteristics like

conditions, loops and external triggers are not handled. This might affect the quality of the test cases generated. The actual model can be used to validate control flow of WS compositions, but it can not be used to certify if the data flow is correct.

The use of the self-adaptive control loop mape-k allowed the SAMBA Framework to accomplish the automatic test case generation and execution. The use of an open source model based test case generation tool enable the framework to generate test cases from an updated model of the target WS composition. However the tool has its limitations, algorithms used to generate the test cases work fine for random path generation, but they don't avoid already covered model edges and vertices. Because of that it is not possible to affirm that all the test operations on a test case will be relevant, since some could be repeated.

The Testing service had to be updated to work with the MBOTCG, and since the model used did not have enough information about the parameters needed to perform the WS operations, it was necessary to create a test oracle exclusive to the target BPEL orchestration. The need for a test oracle to translate the test instructions and to validate outputs of WS operations affects the self-adaptive capability of the SAMBA Framework. If the update in the the target WS composition is related to the addiction/substitution of a WS component or WS operation it might be necessary to update the test oracle, however if this situation occurs the test report will warn the system integrator that an update on the oracle is necessary, never the less it undermines the self-adaptive capability of the framework.

The test scenarios were performed in a well defined test environment, however the designed environment suffered from a classic problem of SOA testing, it does not mimic the same situations possible in the deployment environment. In the test scenarios the WS were deployed and functional, but they were not under heavy use, therefore the test scenarios are not sufficient to affirm that the online test case execution affected or not the applications in some way.

Chapter 8

Conclusion

In this document model-based testing techniques, challenges of SOA testing, and a self-adaptive model-based online test case generation (SAMBA) framework is presented. The framework focuses on the problem of *independent evolution of participants* [28], it uses a self-adaptive loop to update models of target WS compositions, and later uses the models to perform online regression test cases generation.

This project delivered a process for model generation using information available in BPEL files, and a technique to perform online regression testing by combining model-based test case generation and an online test oracle, that works as a middleware between a test case generation and a test services. The SAMBA Framework was tested in three test scenarios, these scenarios were designed to validate requirements, features and to evaluate the performance of the implementation. The quantitative and qualitative data collected during the test scenarios were sufficient to verify if the requirements and features were achieved.

However, the use of only one WS orchestration on the test scenarios raises questions about the quality of the implementation. The model generation process might not be able to convert more complex business processes, therefore the quality of the regression tests generated in different scenarios is questionable, since it depends directly on the models generated and their fidelity to the target WS composition.

The need for a test oracle is also a problem, since the components on the SAMBA Framework were developed in Java, they need to be compiled in order to run in the server. Therefore whenever a BPEL composition is updated, the test oracle also might be updated in order to correctly represent the current interfaces, inputs and outputs of the WSes that belongs to the BPEL orchestration. This can be solved by converting the test oracle to a WS written in an interpreted language, like python or ruby, allowing for a flexible and adaptive test oracle.

The SAMBA Framework produces reports and has a well documented WS composition evolution history, and in the current version it performs composition regression test. It is possible to combine the model-based test case generation process with robustness tests, in order to simulate the consequences of a fault propagation inside a target WS composition and the possible effects that it would have in its host server.

It is possible to adapt the framework so it could be helpful during agile software development, reducing the effort to design and execute tests. It could also be modified in

order to detect other kinds of deviant WS composition behaviors.

To improve the quality of the test cases generated a more robust method to handle the model update could be designed. It is possible to compare the preview and the new model of an application, find the difference between them, generate a reduced model (only with the modifications of the target WS composition), and use it to perform the generation of online regression tests.

The results obtained during the SAMBA Framework online test case generation and execution were submitted on a paper, and a future work with a more complete evaluation of the SAMBA Framework, with different and more complex WS orchestrations, and with more details of the model generation process are already planned.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[3] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. *Pearson Correlation Coefficient*, pages 1–4. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[4] Eddy Bernard, Fabrice Bouquet, Amandine Charbonnier, Bruno Legeard, Fabien Peureux, Mark Utting, and Eric Torreborre. Model-based testing from uml models. In *GI Jahrestagung (2)*, pages 223–230, 2006.

[5] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[6] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[7] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, 23(4):261–313, 2013.

[8] Antonio Bucchiarone, Hernán Melgratti, and Francesco Severoni. Testing service composition. In *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07)*, 2007.

[9] Gerardo Canfora and Massimiliano Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering*, pages 78–105. Springer, 2009.

[10] Tien-Dung Cao, Patrick Felix, Richard Castanet, and Ismail Berrada. Online testing framework for web services. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 363–372. IEEE, 2010.

[11] Andrea Ceccarelli, Marco Vieira, and Andrea Bondavalli. A service discovery approach for testing dynamic soas. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 133–142. IEEE, 2011.

[12] Andrea Ceccarelli, Marco Vieira, and Andrea Bondavalli. A testing service for lifelong validation of dynamic soa. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 1–8. IEEE, 2011.

[13] Radovan Cervenka. Unified modeling language state machines @ONLINE, November 2015.

[14] Elder de Macedo Rodrigues and Avelino Francisco Zorzo. Plets-uma linha de produto de ferramentas de teste baseado em modelos.

[15] C Delerce-Mauris, L Palacin, S Martarello, S Mosser, and M Blay-Fornarino. Plateforme seduite: une approche soa de la diffusion d'informations. *University of Nice, I3S CNRS, Sophia Antipolis, France*, 2009.

[16] Darcy DiNucci. Fragmented future, design & new media. *Print [online]*, 1, 1999.

[17] Ringo Doe. What is web 2.0: Design patterns and business models for the next generation of software @ONLINE, 2005.

[18] Antonio García Domínguez, Antonia Estero Botaro, Juan José Domínguez Jiménez, Inmaculada Medina Bulo, and Francisco Palomo Lozano. Mubpel: una herramienta de mutación firme para wsbpel 2.0. *Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos*, pages 415–418.

[19] Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. Enabling proactive adaptation through just-in-time testing of conversational services. In *Towards a Service-Based Internet*, pages 63–75. Springer, 2010.

[20] Sebastian Götz, Ilias Gerostathopoulos, Filip Krikava, Adnan Shahzada, and Romina Spalazzese. Adaptive exchange of distributed partial models@ run. time for highly dynamic systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 64–70. IEEE Press, 2015.

[21] Didac Gil De La Iglesia and Danny Weyns. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):15, 2015.

[22] Poonkavithai Kalamegam and Zayaraz Godandapani. A survey on testing soa built using web services. *International Journal of Software Engineering and Its Applications*, 6(4), 2012.

[23] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[24] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015.

[25] Lu Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[26] Marius Mikucionis, Kim G Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 396–397. IEEE, 2004.

[27] MIT. Model-based testing (mbt) @ONLINE, July 2014.

[28] Edwin J Morris, William B Anderson, Sriram Balasubramanian, David J Carney, John Morley, Patrick R Place, and Soumya Simanta. Testing in service-oriented environments. 2010.

[29] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[30] SOA Oasis. Reference model tc. *OASIS Reference Model for Service Oriented Architecture*, 1, 2005.

[31] Nils Olsson. Graphwalker - model-based testing @ONLINE, October 2014.

[32] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[33] Chris Peltz. Web services orchestration and choreography. *Computer*, (10):46–52, 2003.

[34] Polarsys. Eclipse titan@ONLINE, July 2015.

[35] Georg Püschel, Sebastian Götz, Claas Wilke, and Uwe Aßmann. Towards systematic model-based testing of self-adaptive software. In *Proc. 5th Int. Conf. Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, pages 65–70. Citeseer, 2013.

[36] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[37] Keng Siau. *Principle Advancements in Database Management Technologies: New Applications and Frameworks: New Applications and Frameworks*. IGI Global, 2009.

[38] OASIS Standard. Web services business process execution language version 2.0. *URL: http://docs. oasis-open. org/wsbpel/2.0/OS/wsbpel-v2. 0-OS. html*, 2007.

[39] Georgios Tselentis, John Domingue, and Alex Galis. *Towards the Future Internet: A European Research Perspective*. IOS press, 2009.

[40] uml diagrams org. Behavioral state machine @ONLINE, February 2013.

[41] uml diagrams org. Protocol state machine @ONLINE, February 2013.

[42] Mark Utting. Modeljunit test generation tool@ONLINE, November 2015.

[43] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. 2006.

[44] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[45] w3School. Xml wsdl @ONLINE, November 2015.

[46] Yong Wang. A survey on formal methods for web service composition. *arXiv preprint arXiv:1306.5535*, 2013.

[47] yWorks GmbH. yed - graph editor @ONLINE, September 2017.