TÉCNICO LISBOA

Test Accelerator for Service Oriented Architectures

(SOA-Test Accelerator)

João Filipe Garrett Paixão Florêncio

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Doctor José Carlos Campos Costa

Examination Committee

Chairperson: Doctor X

Supervisor: Doctor José Carlos Campos Costa

Member of Committee: Doctor António Manuel Ferreira Rito da Silva

October 2015

Abstract

This document describes the design and development of a tool, "SOA – Test Accelerator", created to automate intelligent test creation in service oriented architectures. The need for such a tool comes from the increasing difference between testing some independent services and testing their overall interaction. As the system architecture grows in number of services, manually creating test case scenarios becomes a heavy burden. SOA-TA's ultimate goal is to reduce the time spent on combining and orchestrating service calls to simulate a business process. The work presented here is based on five stages and their outcome depends on the exhaustiveness level chosen by the user. First, the automatic generation of test cases through process descriptions analysis, having business requirements in consideration. Second, the generation of the input set required to execute these test cases. Third, the production of specific service calls, by means of test scripts to be run on Apache JMeter. Fourth, the execution of these scripts and fifth, showing the results. SOA-TA will be useful for operations that rely on consecutive service calls, and need to ensure the overall system compliance with previously set requirements.

Keywords:SOA testing,test case generation,test case execution,web services,service testing,coverage.

Resumo

Este documento descreve o desenho e desenvolvimento da ferramenta "SOA – Test Accelerator", criada para automatizar a criação inteligente de testes para arquitecturas orientadas a serviços. A necessidade desta ferramenta provém da crescente diferença entre testar um conjunto de serviços independentes e testar as suas interacções. À medida que o sistema aumenta em número de serviços o tempo gasto na criação de cenários de teste aumenta proporcionalmente, factor este que aqui tentamos mitigar. O objectivo maior do SOA-TA é, portanto, reduzir o tempo despendido em combinar e orquestrar as chamadas aos serviços que simulam um processo de negócio. O funcionamento é baseado em cinco etapas e o seu resultado depende da exaustividade escolhida pelo utilizador. A saber, inicialmente, a geração automática de casos de teste através de uma descrição dos processos, tendo em consideração os requisitos do negócio. Seguidamente, a criação da lista de dados de entrada que alimentam esses casos de teste. Em terceiro, a geração das chamadas aos serviços, por scripts de Apache JMeter. Em quarto a execução desses mesmos scripts e por último a apresentação dos resultados. O SOA-TA será útil em operações que dependem de chamadas consecutivas a serviços e que precisam de garantir que os requisitos previamente estabelecidos são respeitados.

Keywords: testes, geração de testes, execução de testes, SOA, cobertura

Acknowledgements

I would like to thank INESC-I&D for giving me the opportunity of learning with the best, Wintrust for the vision and for believing in our work, my thesis supervisor Professor José Costa for the continuous support of my study and related research, for his patience, motivation, and immense knowledge.

Last but not the least, I would like to thank my family and friends, not only for their willingness to help but specially for their support. They made it all worth it.

Contents

List of Tables

List of Figures

Acronyms List

BPD Business Process Diagram

BPEL Business process execution language

BSF Bean Scripting Framework

CA Computer Associates International Inc.

DG Directed Graph

EA Enterprise Architect (software)

GUI Graphical User Interface

HTTP HyperText Transfer Protocol

ISTQB International Software Testing Qualifications Board

OASIS Organization for the Advancement of Structured Information Standards

REST Representational State Transfer

SCR Software Cost Reduction

SMT Satisfiability Modulo Theories

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

TAP Transportes Aéreos Portugueses

UDDI Universal Description Discovery and Integration

V&V Verification and Validation

WADL Web Application Description language

WSDL Web Services Description language

WSIL Web Services Inspection language

XML Extensible Markup Language

1

Introduction

We cannot solve our problems with the same
thinking we used when we created them.
-Albert Einstein

Nobody is perfect. Humans make mistakes all the time. While some of them are negligible, some are expensive or dangerous. We are used to assume that our work may have mistakes, even minor ones, and it needs testing. Software testing in particular is now recognized as a key activity of large systems development. Companies have come to understand that, in order to achieve an higher quality product, efforts on Verification and Validation (V&V) [1], save time and money. Service-Oriented Architecture (SOA) [2], as implemented by web services, present features that add much complexity to the testing burden. But first, let us clarify what SOA actually is.

According to Organization for the Advancement of Structured Information Standards (OASIS), SOA is "A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations" [3]. In simpler and concrete terms, it is a pattern in computer software design in which application components provide services to other components, with both parts typically connected over a network. It stands out from other architectural approaches for being independent of vendor, product or technology.

There are numerous studies proving the benefits of implementing or migrating to service architectures [4]. According to Florida State University professor Tom Yoon and his paper presented at the "Americas Conference on Information Systems" [5], all five of the organizations in their sample, experienced improved business agility and lower costs as a result of SOA implementation.

Following these and other conclusions, spending on SOA initiatives was expected to top $40 billion by 2011, which reflects a compound annual growth rate of 45.2% from 2007-2011 [6].

In spite of the popularity service architectures are gaining among technology companies, professionals recognize that the migration is not so straightforward as it might look. Deploying a software application can be a one-time activity, with the process going trough all usual stages before release. On the other hand, with SOA, the process is more evolutionary, it is a journey for an organization over a long period of time [7].

To keep up with this paradigm change, software functionality is now required to be available in a smaller frame of time than it was a few years ago. This fact, due to human nature, inevitably leads to mistakes and to an increase in the probability of finding bugs. SOA, by its dynamic and always-on nature makes most testing techniques not directly applicable to it [8]. For example, in a service-oriented system it is almost impossible to precisely identify the actual piece of code that is being invoked at a given call-site [9]. As Bartolini [10] states, "even if best practices are followed by the developer to test a service to ensure its quality, nothing guarantees that it will operate smoothly as part of a dynamic distributed system made of multiple orchestrated but autonomous services."

As service oriented systems are the pinnacle of this loosely coupled kind of systems, we need to ensure more than just the functional integrity of its units, we need to be sure all the components are well integrated. This lead software testing to be separated in three stages: unit, integration and validation testing.

We are going to focus on the integration stage, in which individual software modules, in our case web services, are combined and tested as a group. In order for us to group all the components of a service based system we need to create test cases while focusing on the tasks those services will preform. Coming up with relevant and useful test is as of now almost always a manual task. Thus, if the process of deriving test cases could be automated and provide requirements-based and coverage-based test suites that satisfy the most stringent standard, such as the DO-178B [11] used in civil aviation, dramatic time and cost savings would be realized [12]. This is what we are aiming to do.

1.1 Motivation

In order to fill the gap between regular testing and modern service oriented architecture testing, arises the SOA-Test Accelerator. The main objective of our work was to create something that takes the responsibility off the testers of coming up with intelligent and smart test paths or scenarios.A wide variety of applications dedicated to independent service testing, usually called system micro-state testing, are already available. Our intention was to focus on the macro-state, the so called orchestration level, which guarantees that the whole system accomplishes its task.

Commercial tools are mainly software applications where tests can be executed, and results can be seen and extracted. In all of them, the tester has to specifically create the test cases, give the inputs, and come up with a testing strategy suitable for the process.

Our goal was to create an on-line platform where the user could enter the BPMN [13] description of the tasks to test and then our Test-Accelerator would do the rest. Which means creating the test cases based on coverage standards, discover intelligently the proper input data, run the tests, and ultimately provide the user an overview of the results.

In simpler terms, we are going to test, not one service, but multiple services interacting to achieve an expected result.

The current document will detail the goals, challenges and solutions found along the way.

1.2 Objectives

As previously stated,to our knowledge there is not much work done on testing services oriented architectures as a global system hence our first goal was how could we address this issue and somehow create a useful tool. In collaboration with WinTrust, a Portuguese company specialized in software testing, we realized testers spent too much time creating test scenarios. Besides this, obviously important, extent factor, the only way of certifying paths was to create a test case for each. This led to having parts of the system untested and others tested more than once. Exhaustive testing was required to assure the tasks would perform well.

Having this necessity in mind, we decided to come up with a system that could create test paths/scenarios based on a business process description. These paths would have to follow the coverage rules used by International Software Testing Qualifications Board

(ISTQB) further described on section 3.1.

The problem of automatically generating input data that will exercise the given paths was the next big issue to solve and the only way it seemed feasible was having concrete knowledge on every operation and knowing input boundaries. This task will also be detailed on section 4.5.

Therefore, the main improvement or innovation on SOA-TA is the adaptation of concepts and ideas already studied on graph analysis and satisfiability modulo theories, and bringing them to improve automatic testing. Therefore the main goal was to create a system that would serve testers and their work, creating value for the company using it.

1.3 Thesis Outline

Here on chapter 1, we introduced the thesis theme, stated our goals and the steps needed to achieve them.

On chapter 2 we are going to look at the state of the art. Much research has been done in SOA testing. We chose a small set of the most relatable work done by credited people to enlighten our path and to give us some insight on the matter. We are also going to see what commercial tools have to offer today and analyse why SOA-TA will be different. In the end, section 2.3 will summarise our conclusions on the related work.

On chapter 3 we will go through the theoretical model supporting our line of thought. We are going to detail all the coverage metrics used here and all the underlying models and techniques to apply them.

On chapter 4 is the full description of our solution. Here we are going go through all the phases of the process, from using a BPMN model to generate test paths, discovering input data, generating the scripts to actually executing them. It is divided in seven sections, each addressing a specific stage. They are organized sequential and chronologically, meaning section 4.2 will talk about the first stage of the system, model creation, and section 4.8 of the last one, reading test results.

Chapter 5 focuses on the implementation phase. Here we are going go describe actually how we have created SOA-TA, its methodologies, technologies and technical decisions. Different sections address a specific component, section 5.1 describes the user interface, 5.2 traces the business logic layer and the last one, 5.3, addresses the database and data access.

On chapter 6, we are going to establish the methods on which we relied to evaluate the work done while chapter 7 shows our measured results. Chapter 8 contains this documents's conclusions and a future work analysis.

1.4 Summary

On this chapter we have introduced the thesis theme. First, established the importance of software testing nowadays and in the future, whether manual or automatic. Also introduced the concept of service oriented architectures and how its distinct features make usual approaches not directly applicable to them. From this fact came the motivation of our work. We intended to create a platform for automatic testing at the architecture level, including test case and input data generation, and execution, with an all-in-one solution. As of today, there is no product doing all these tasks, and even combining several of them would not solve the problem we are facing, since test case generation follows strict coverage rules which will be discussed further.

2

Related Work

Tying the work we have done with previous research is not an easy job. This is due to the fact that we are joining two different study areas, automatic test case generation and service architectures testing platforms. While there is some really interesting work done in the first area, there is virtually no application of its results on the later. Hence, the following sections will be divided in this two major categories, the academic (mostly research work) and the commercial testing frameworks.

2.1 Academic

In this section we are going to review some strategies presented by the academic community to both test-case generation and general SOA testing problems. Each subsection focuses specifically in a single piece of work, presented in proceedings, papers or books.

2.1.1 Test-Case Generation

Coverage based test-case generation using model checkers

Rayadurgam [12] worked on a method of automatically generating test-cases with model checkers. Model checking is a technique to exhaustively and automatically check if a certain model meets a given specification [14]. The authors suggest a formal framework to describe both the system's model and its specifications, which is then used alongside the test criteria to produce test-cases. The main algorithm relies on generating a set of properties and then asking the model checker to verify them one by one. This method does not address the obstacle of state space explosion. Also, creating a formal description

of simpler systems and models is a task that not everyone person could do, besides its necessity might be obliterated when using a different approach like ours, in which a simple process description in BPMN is used.

WSDL-based automatic test case generation for Web services testing

The approach chosen by these investigators [15] relies on a Web Services Description language (WSDL) document analysis to figure out dependencies between operations. The service contract is parsed and test data is generated by examining the message data types according to standard XML schema syntax. Input and output dependencies reveal which operations can be grouped in the same test case. Although they are indeed automatically generated and cover all the operations, this method tests each service independently and not all the services used in a given process. Unless the whole system at hand is a unique service, analysing its operation set does not suit integration testing.

2.1.2 Testing SOA

Testing in a Service Oriented World

Ribarov et al [8] address testing in three major functions, unit, integration and functional system testing. Unit testing is a critical activity to ensure the architecture's functional integrity. Since there is a great similarity in unit service testing and unit general software testing, the authors propose the re-utilization of work already done in components testing and so generate black box tests from the WSDL document, and take advantage of commercial tools potentials (like the ones mentioned on section 2.2) to generate test-cases.

Integration testing is also challenging mainly due to third party services dependencies and the possibility of services being missing in the moment of testing. The authors suggest an integration testing strategy executed as soon as the development begins. Simulating missing or unavailable services may be delicate in some cases but usually, necessary.

Functional System testing is ensuring the whole structure is tested in its integrity. The authors state that it has to be accepted that for all but the simplest of services, it is very difficult to test exhaustively every input or output. According to them, "Up to date there is no end-to-end automated system testing solution for SOA on the market." [8] Properly knowing the system functions is vital to guarantee its overall performance.

Whitening SOA Testing

This paper [10] presents a solution to go around a common trait existing test approaches share, which is they treat the web services as black boxes and focus on the external behaviour but ignore internal structure, as our SOA-TA does. The authors claim to have found a solution on how to make services more transparent to an external tester and still maintain the flexibility, dynamism and loose coupling of SOA [16].

Their approach called Service Oriented Coverage Testing (SOCT) relies on creating a governance framework testing at the orchestration level during validation. Using this method, it can monitor what parts of actual source code are executed but it requires services developers to instrument the code so as to enable target program entities execution monitoring. Then, SOCT's results present what blocks and operations were covered in the tests. This gives an idea of how exhaustive the test was.

This method of addressing SOA testing shows good results. Nevertheless the need to modify services code to allow its instrumentation, cannot be disregarded. In a large and diverse system, it is common to make use of services held by third parties which cannot be modified or even accessed [17].

The concept of service oriented coverage testing is however similar to the idea guiding our own SOA-Test Accelerator. In a more general, not source code specific, way, we are also focusing on what parts of the system are tested with each test-case.

A Survey on Testing SOA Built using Web Services

In this survey Kalamegam et al [18], stress the end-to-end role importance in testing an orchestration or service sequence. What is happening inside the environment and the insight required is also pointed out as a crucial factor to successful testing.

At the orchestration level, white or black box testing may be considered. Black-box testing is a method of software testing that examines the functionality of the tested system without peering into its internal structures or workings [19]. On the opposite, white-box testing refers to test methods that rely on the internal structure of the software. White-box methods are based on executing or "covering" specific elements of the code [20].

Usually in SOA, black-box testing is the most common approach since the very nature of web services is related to covering implementation details and focusing of interactions in a most general and standardized way.

Kalamegam [18] states the importance of devising an efficient strategy for testing the SOA infrastructure, web services (individual and composite services) and end-to-end business sequences.

Coyote: An XML-Based Framework for Web Services Testing

In this work, Tsai [21] proposes an XML-based object-oriented (OO) testing frame to test web services rapidly. It uses test scenarios crated using a standard XML notation to generate test-cases. Created test scripts are read in the test engine and then executed, in an all-in-one, framework.

The idea behind coyote framework seems promising, however Tsai [21] does not show any results to substantiate their framework's success and there are nearly no references to its usage.

2.2 Commercial Testing Frameworks

In this section we are going review some tools commercially available today claiming to address the service based architectures testing. We are not aiming to provide an exhaustive list of every single software platform that has some testing capabilities, we have just selected a few of the biggest, most widely used and most representative of the state-of-the-art.

2.2.1 HP Unified Functional Testing (UFT)

Hewlett-Packard has launched in 2012 a framework which provides functional and regression test automation for software applications and environments [22] It has incorporated a previous web services specific tool, HP Service test. It enables developers to test, from a single console, all three layers: the interface, the service and the database layer.

As the other testing frameworks/tools presented in this chapter, it claims to automate testing. This is, at some extent true since it uses a scripting language to make repetitive tasks automatic. However, as stated in "Testing in a Service Oriented World" [8], most of the automated testing tools are oriented to unit testing of the services, which means it only automates independent testing of services, not the whole architecture testing process.

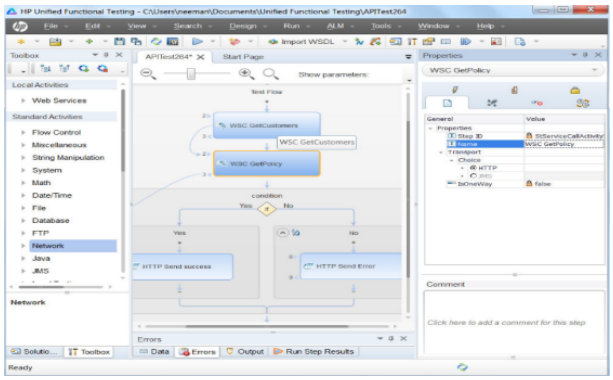

Figure 2.1: HP UFT Screenshot

2.2.2 Oracle Testing Accelerators for Web Services

Oracle has launched in 2011 the 12c version of its framework, Enterprise Manager, which includes Testing Accelerators for Web Services. It allows testing the quality and performance of service-oriented architectures based applications directly at the Web Service interface level. It claims to automate functional and regression testing of services and uses an OpenScript platform to allow users to generate scripts.

These scripts can combine multiple Simple Object Access Protocol (SOAP) requests in a single test script. Web Services test scripts can be created by selecting which methods to call from this store of parsed Web Service method requests or by specifying the requests manually. It can also specify the data inputs for those Web Service requests. It can reuse response values from one request, parse them and then use it as input for subsequent requests using script variables.

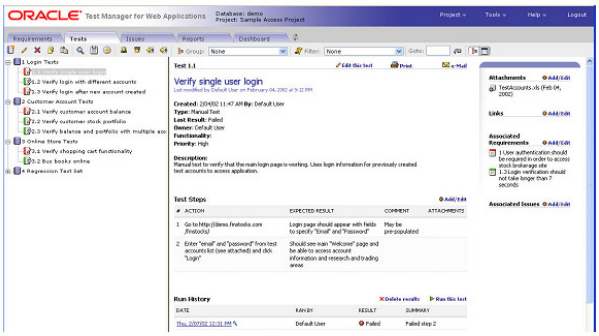

Figure 2.2: Oracle Testing Accelerator Screenshot

Oracle's testing accelerator cannot however, make any coverage analysis and suggest or create any test cases. Its focused on the simplification of the test process. It materializes an idea very similar to the one shown by Rayadurgam [12], testing services on the interface level.

2.2.3 Parasoft SOAtest

Parasoft, a software testing company claims to have robust software and to be the industry leader since 2002. It uses automatically-generated tests as building blocks, allows complex test scenarios that exercise and validate business transactions across multiple endpoints. "Tests can be automatically generated from artefacts such as WSDL , Web Application Description language (WADL), Universal Description Discovery and Integration (UDDI), Web Services Inspection language (WSIL), Extensible Markup Language (XML), Schema, Business process execution language (BPEL), HyperText Transfer Protocol (HTTP), traffic, and key industry platforms. For example, just provide a WSDL or WADL to generate a test suite that exercises each defined resource element/operation's well as checks for schema validity, semantic validity, interoperability, and definition file changes." [23]

This strategy is also really similar to what we have seen on Rayadurgam [12]. It relies on the service description analysis to come up with test cases in which related operations are grouped together.

Other features are:

- Provides capabilities such as test parametrization.
- Alerts to changes in definition files.
- Generated tests can be:

  Extended with additional values and assertions.

  Leveraged for load/performance testing and security testing.

  Used to build end-to-end test scenarios that cover multiple layers and/or service invocations.

  Parametrized to increase their scope and comprehensiveness.

This strategy might be adequate when the tested subject is a unique service, when a set interrelated services are used to perform a task, platforms like Parasoft's fail to grasp the relation between them, and therefore cannot automatically generate tests.

2.2.4 LISA for SOA Testing

ITKO a company part of Computer Associates International Inc. (CA), produces testing software and claims to have a product suite that reduces software delivery time lines by 30% or more. Their Lisa for SOA Testing solution relies mainly on service virtualization. Other features it supports are:

- Automated regression, functional and performance testing at build or in a continuous validation mode
- Capture, model and simulate the behaviour of services and underlying systems.
- Test case reuse using point-and-click graphical interface for test creation and maintenance
- Allows developers and non-developers to quickly and easily build and elaborate.

Lisa's platform shares a popular trait among testing framework, it serves specifically the purpose of executing tests without automating the test case creation process. It does not use task or process descriptions to find relevant test scenarios.

2.2.5 Apache JMeter

JMeter is a powerful, easy-to-use, and free load-testing tool. Since it is a pure Java application it is completely portable between OS's. Although JMeter is known more as a performance testing tool, functional testing elements can be integrated within the Test Plan, which was originally designed to support load testing. Many other load-testing tools provide little or none of this feature, restricting themselves to performance-testing purposes. Other features it supports are:

- Functions can be used to provide dynamic input and extensibility.
- Scriptable Samplers (BeanShell, Bean Scripting Framework (BSF)-compatible languages and JSR223-compatible languages)
- Easy-to-use Graphical User Interface (GUI)

Jmeter was created to be first of all a load testing tool. It has evolved to support functional testing. Nonetheless, we can see its performance testing features are what has

made its success. It does not support WSDL parsing, as all web services testing is done manually, in a sense of knowing what are the types and operations a web service supports or provides.

2.3 Related Work Critical Analysis

The work shown on 2.1 presents some capable strategies to automatically generate test cases, however they also present some issues like relying on a system's formal description or not having in consideration what coverage methods the user wants to use. In the practical world there are few operations requiring formal validation and verification, while almost every company basing its processes on web services needs a way to know how reliable the testing results are.

Turning the heavy and time consuming process as testing is, to a lighter and faster one could not rely on creating a mathematical, formal description of the system and its requirements. However, we could not disregard the important factor of coverage compliance and exhaustiveness levels. That is what as motivated us to use a simple process description in the test generation process.

Regarding commercial solutions, they all present some kind of automation, whether on test creation or test execution. No company presents a way of dynamically perceive meaningful execution paths from business process descriptions.

Our first focus was to allow external process modelling, with "Bizagi", a tool described in section 4.2 or other tool supporting BPMN format, and knowing what specific inputs/outputs are needed/generated. In the future, knowing the inputs may become an issue when specific service behaviour is unknown, not completely defined yet, or changed somewhere along the way. This issue must be addressed and discussed with the users, as soon as testing starts. It is virtually impossible to assure the correct behaviour of the tool if the tested system changes in between. Allowing for the user to upload files previously created in Bizagi or other .BPMN file format compliant application, removed the burden of having to provide a graphical interface where they could describe the system to test.

Script creation accelerators are also a key feature in some solutions to allow efficient manual creation procedure. Our approach is based on automatic scrip creation based on the execution graph.

One of the distinguishing features of WinTrust - SOA Test Accelerator is finding and

certifying the minimum number of test scenarios and automatic providing with proper input. Although we might look at commercial tools as competition, in fact they are not, since none of them offers the functions we propose. As stated in the beginning, the SOA-TA final stage is the script generation. This script would have to be executed in one of the testing tools mentioned in this chapter. To do this, we have taken into account the financial cost of software licenses/selling price, the available documentation and market share they have and chose Apache JMeter [24] to be the script execution tool.

In conclusion, as we can see that most commercial tools have some common features like facilitating test creation by simple users, while giving advanced users/developers a way to extend them; all rely on a visual interface to simplify test creation. Most of them use some scripting language (OpenScript, Visual Basic, BeanShell, etc.). However the most decisive factor is that all solutions assume users know how to test their system to get an acceptable coverage level, while SOA-TA knows exactly what and how to test in order to comply with formal coverage rules. It is not focused on test programming but on test conceptual creation.

Other SOA-TA's differentiating features are:

- Using Bizagi as interface to model the execution flow allows an efficient way to share testing scenarios
- Automatically generation of test-case scenarios
- Allow user to choose coverage metric - 0-switch coverage or 1-switch coverage
- Discovery of input information required to achieve coverage
- Validation of test-case scenarios according to the metric desired
- Script generation for later execution on JMeter

In the next chapter, we are going to clarify some testing concepts and standards.

2.4 Summary

Related work was presented in two different groups, the academic or research based, and the commercially available testing tools. Within scholar articles, we have also separated test cases generation from service architectures testing in general. In the first ones, the

shown coverage-based test case generation methodologies are useful when using code, formal specifications or requirements in Software Cost Reduction (SCR) notation [25] as a source, which is not our case. Besides, there is no mention of automatic input generation alongside the test cases. However, their ideas on transition coverage supported our implementation.

On section 2.2 we have explored and depicted a set of the most used testing frameworks today. These show a huge potential in becoming the testing function easier and faster with scripting capabilities, yet, they lack the main functions of our SOA-TA, which are auto generation of test cases and its input data, while using coverage analysis.

3

Background

In this chapter, we are going to provide a description of the guiding metrics used in the system development. As said before, since SOA-TA is a testing tool and since WinTrust, the company funding the project, has a close relationship with ISTQB (International Software Testing Qualifications Board), we made an effort to follow its guidelines. The rules described in the next sections, particularly section 3.1.3, are explained in detail in several software testing manuals [26, 27] therefore we do not share their authorship. Concerning these rules, what we need to know in the first place is that we are going to use them to ensure two different test exhaustiveness levels. In large-scale systems, it is virtually impossible to test all the combinations of output decisive factors and so this standard was designed to measure the tests thoroughness.

3.1 Coverage Metrics

From our perspective, testing an architecture orchestration of interconnected services, implies making sure information produced by one service or operation is suitable to serve as input to the next services in the process. Our first task was to find a solution to accurately model services, its calls, the information produced and workflow of a singular or multiple tasks. Then, we would apply coverage rules to check how the task tests should be created.

3.1.1 The Model

Service calls specific flow, lead us to choose a state-based testing approach. State-based testing is ideal when we have sequences of occurring events and conditions that apply to

those events [26], meaning the proper handling of a situation depends on the events and conditions that have occurred in the past.

There are several types of state machines like finite automatons (no guards or actions), Mealy machines (outputs depend on current state and inputs), Moore machines (outputs depend on current state only), state charts (hierarchical states), state transition diagram which are graphic representations of a state machine and state transition tables being a tabular representation of a state machine. In our case it is easy to see that from the above that a Mealy machine is what describes better our system since the input for each state will determine output.

Having said this, each service operation in our model will be represented as an individual state. In the case of subsequent service calling (the usual in Service Oriented architectures) all the other services, which can be called "from" the first, will be represented as another state. If the orchestrator, i.e. the program responsible for calling service operations, applies any transformation to its output we assume it is part of that operation actions so a state is a black box which receives data and outputs data, only.

3.1.2 Diagram

A classic form of state diagram for a finite state machine is the directed graph [28]. The notion of graph proved itself very useful since there is a great amount of work already done on extracting meaningful information from them (efficient searches, path-cost analysis, augmentation paths, etc.) [29, 30]. The type of graph we will use is a Directed Graph (DG). In graphs notation there are two main concepts, vertices (or nodes) and edges. A vertex of a DG can have incoming vertices and outgoing vertices [31]. We can describe a graph in mathematical terminology as $G = (V, E, V_0, V_f)$, where

- V is a set of nodes
- V_0 is a set of initial nodes, where $V_0 \subseteq V$
- V_f is a set of final nodes, where $V_f \subseteq V$
- E is a set of edges, where E is a subset of $V \times V$

In our case, the nodes will represent service operations and the edges will be its calls with all the information required (operation parameters). "Example graph - 1", shown on Fig. 3.1 demonstrates a possible flow with its formal description being:

- $V = \{A, B, C, D, E, F, G\}$
- $V_0 = \{A\}$

- $V_f = \{G\}$
- $E = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

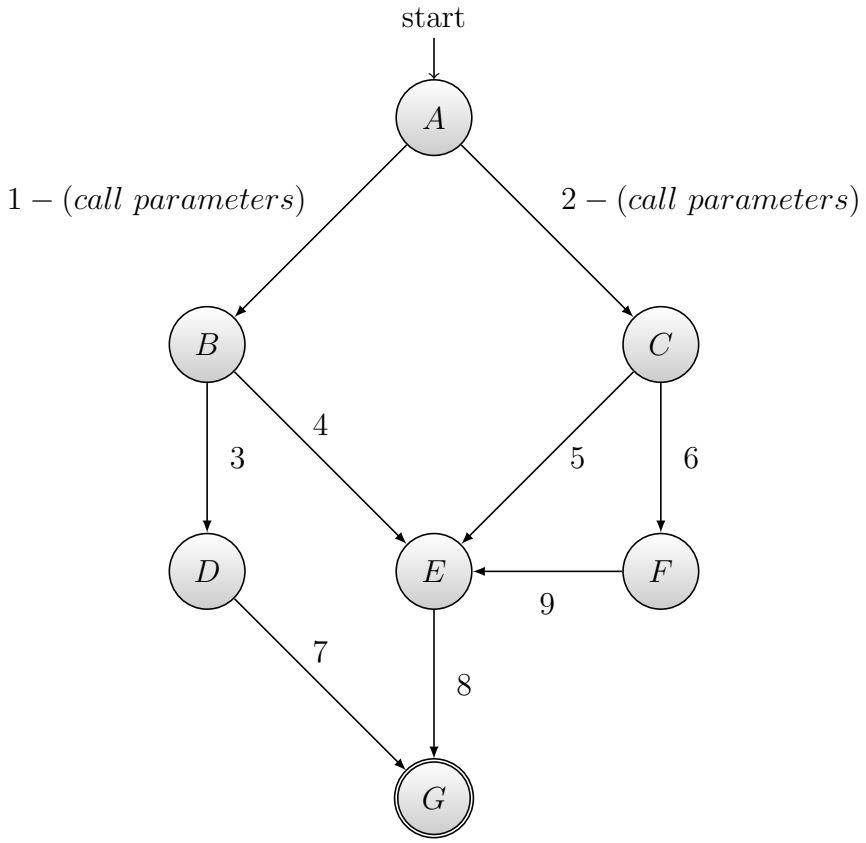

Figure 3.1: Example graph – 1 – Directed Graph

Correctly creating the execution graph (which services call each) is crucial to ensure the good performance of SOA-TA's following steps. Modelling should be done carefully for each situation, having in mind possible pitfalls like graph loops that should not exist.

3.1.3 Switch Coverage

As stated in the beginning of 3.1, here we are going to explain the "$(N-1)$ Switch Coverage" testing strategy [26]. It requires a state-based diagram, which in our case is the DG, mentioned in the previous section. $(N-1)$ Switch Coverage, also named "*Chow's switch coverage*", after Professor Chow, who developed it, states that at least one test must cover each transition sequence of length N or less. Therefore, N represents the length of the transition sequence tested. By this description is easy to understand that $1 - switch$ coverage extends $0 - switch$ coverage. A higher level always assumes the lower levels are also covered. For example, if we would like to test all transitions once,

individually, the sequence length (N) would be 1, "$(N-1)$ switch coverage" becomes $0-switch$ coverage in this case. Since the requirements phase it has been decided that SOA-TA would have to support $0-switch$ coverage (or $Chow-0$) and $1-switch$ coverage (or $Chow-1$) test generation. To clarify possible misunderstandings let us look at the following example.

"0-switch coverage" or "Chow-0" (N=1)

This is the lowest level of coverage used in our system. It covers only transitions of length 1. It simply tests one service calling another and that is it, for all transitions in the graph. In the example on Fig. 3.2, all transitions (edges) of the graph would be tested once.

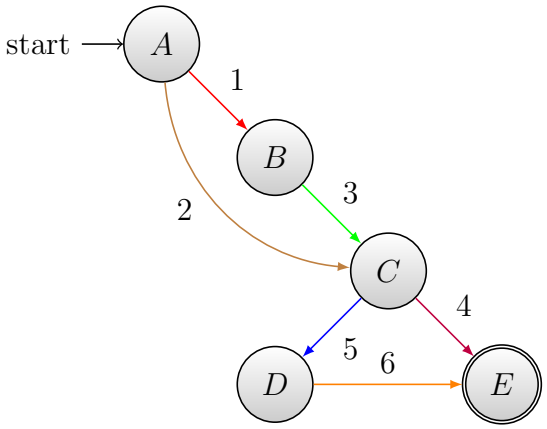

Figure 3.2: Example graph - 2 – Directed Graph

To achieve $Chow-0$ we would have to test transitions $1, 2, 3, 4, 5$ and 6 ate least once, each.

This testing strategy or rule only implies this transitions to be tested, whether in the same test case or in individual test cases. From our perspective however, since vertices/nodes represent steps of an whole joint process, it is only acceptable to test them in an end-to-end mode, from the first step to the last. So, to achieve a coverage of $Chow-0$ as mentioned above, we would have to generate two different test cases. One exercising the path $A \rightarrow_1 B \rightarrow_3 C \rightarrow_4 E$, and the other $A \rightarrow_2 C \rightarrow_5 D \rightarrow_6 E$.

"1-switch coverage" or "Chow-1" (N=2).

This is the higher level of coverage used in our system. It covers transitions of length 2. It tests all two consecutive service calls. On Fig. 3.2, there are 6 transitions of length 2. To achieve $Chow-1$ we would have to tests all pairs of consecutive edges, edges 1 and

3, 3 and 4, 3 and 5, 5 and 6, 2 and 5, 2 and 4. So, to achieve a coverage of $Chow - 1$, we would have to generate at least 4 different test cases, like:

- $A \rightarrow_1 B \rightarrow_3 C \rightarrow_4 E$
- $A \rightarrow_2 C \rightarrow_4 E$
- $A \rightarrow_1 B \rightarrow_3 C \rightarrow_5 D \rightarrow_6 E$
- $A \rightarrow_2 C \rightarrow_5 D \rightarrow_6 E$

A more complex and exhaustive example on 0 *and* 1 *switch* coverage levels is presented now.

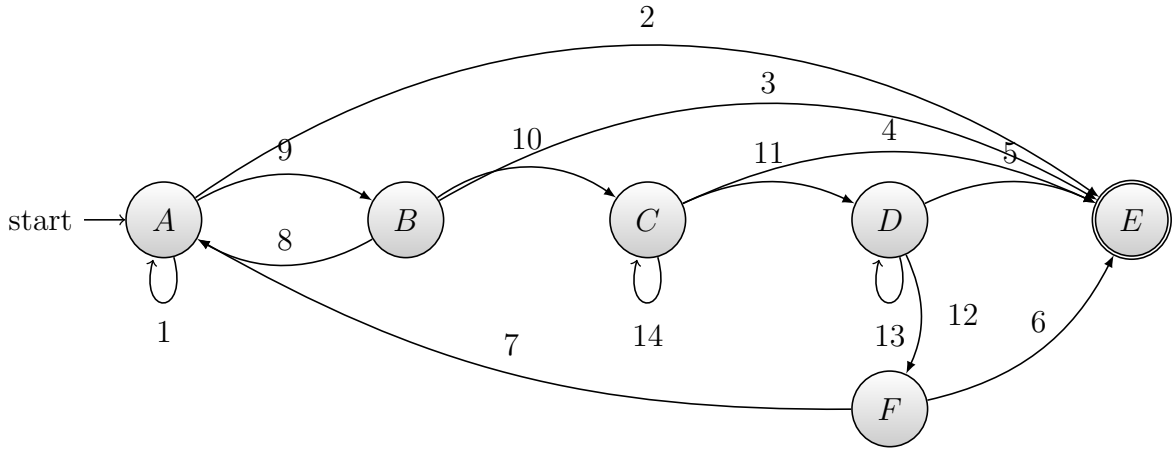

Figure 3.3: Example graph - 3

Steps	1	2	3	4	5	6	7	8	9	10	11	12	13
Path 1	A	B	C	D	F	A	B	C	C	D	D	F	E
Path 2	A	B	A	A	E								
Path 3	A	B	C	D	E								
Path 4	A	B	C	E									
Path 5	A	B	E										

Table 3.1: Chow-0 Test Paths

Tables 3.1 and 3.2 show respectively a possible path set to achieve both coverages levels used. These are to be read horizontally, from the process *Start* node, A, to the *End* node, E. Please note that these are not the only possible paths that verify the coverage levels, there are many, in this case even infinite possibilities. As these were automatically generated by SOA-TA they are minimum, with the first test cases exploring the maximum

Steps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Path 6	A	B	A	B	C	D	D	D	F	A	B	A	A	B	C	C	D	F	E
Path 7	A	B	C	D	F	A	A	A	E										
Path 8	A	B	C	D	F	A	E												
Path 9	A	B	C	C	C	E													
Path 10	A	B	C	D	D	E													
Path 3	A	B	C	D	E														
Path 4	A	B	C	E															
Path 11	A	A	B	E															
Path 12	A	B	A	E															
Path 13	A	E																	

Table 3.2: Chow-1 Test Paths

transitions in themselves before creating another path. Highlighted in bold, *Path* 3 and 4 are common to both levels meaning that in a full set they would appear only once.

3.2 Summary

In section 3.1 we have stated and detailed the coverage metrics used to create test paths. First introduced the concept of state-based testing, then the diagram in the form of a directed graph, both decisive at this stage. In path generation, the criterion know as $(N-1)$ *switch* coverage [32] was used to ensure compliance with the most up-to-date ISTQBs testing guidelines. SOA-TA relies on two different exhaustiveness levels. The first, $0-$*switch* coverage, in which all sequences of length 1 are tested, and the second in which sequences of length 2 are also tested.

4

Solution Description

In this chapter, we are going to provide a full description of our system. As mentioned before, sections following 4.2, in which we will provide a general system description, are sequentially lined up, which means they describe consecutive running stages. The first one is about creating a model to describe business processes.

4.1 General Description

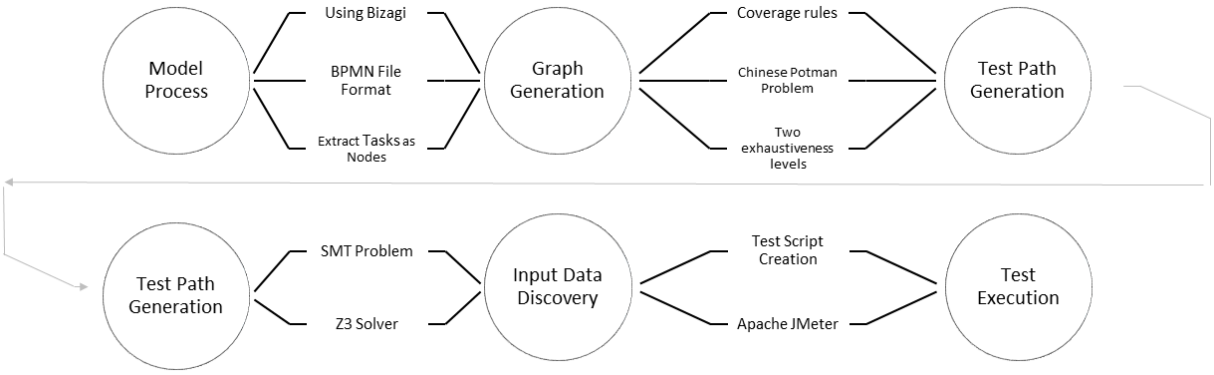

Figure 4.1: General System Description

Figure 4.1 shows an overview of our SOA-TA. Let us follow the steps (represented by circles in the picture) one by one. The first stage portrayed is *Model Process*, this is an external task, not executed inside the platform, in which a general task process, like booking a flight, making a bank operation or filing a report is modelled using business process modelling notation (*BPMN*) [13]. Then, the tester using SOA-TA, will upload this file and behind the scenes, a parser will read it and transform its relevant data into a directed graph, this stage is called, as the name implies, *Graph Generation*. Once

the graph is generated, the coverage rules detailed in section 3.1.3 are used to produce test paths, i.e. different courses of action to execute the task. This can be done in two exhaustiveness levels and depend directly on the solution of a graph traversing problem, known as Chinese Postman problem. The following *Input Data Discovery* stage is where we generate data to feed the test cases. Knowing what web services are used in the process, knowing what restrictions some output values are bound to and with the ability to preform some attempts, getting suitable input becomes a decision problem, addressed in section 4.5. With all the decisive stages concluded, in *Test Execution*, we take the test script containing paths to test and accurate input data and execute it. This is merely a brief description of the process. Next sections will detail how this is achieved.

4.2 Modelling the process

As stated in 3.1.1, system flow's nature led us to choose a directed graph to represent the business processes. This also has to do with the increased usage of business process modelling languages such as BPMN [13] and BPEL [16]. Most companies already have their business processes modelled in one of these two languages, which eases the model producing and often exemption. BPMN defines a Business Process Diagram (BPD), based on a flowcharting technique tailored for creating graphical models of business process operations [13]. BPEL (Business Process Execution Language), is an OASIS standard [16] executable language for specifying actions within business processes with web services.

Since there is a strict correlation between these and since both support convenient graph-based modelling [33], we decided to use BPMN (.bpmn file extension) with the help of Bizagi Modeler, a freeware tool.

Bizagi Modeler is a modelling tool much like Microsoft Visio [34] or Enterprise Architect [35], but more targeted to create BPMN models. We are not going to describe all the artefacts and methods used in the notation since it is not the focus of this work. For further reading on the matter, please address to the standard documentation [13].

Figure 4.2 shows an example of a possible business process involving five different services, each with one or more operations executed.

As Fig. 4.2 depicts, "Process 1" starts when "Service A Operation A_1" is called and ends when "Service B Operation B_2", "Service D Operation D_1" or "Service E Operation E_1" return. The information seen on the edges might represent one of two

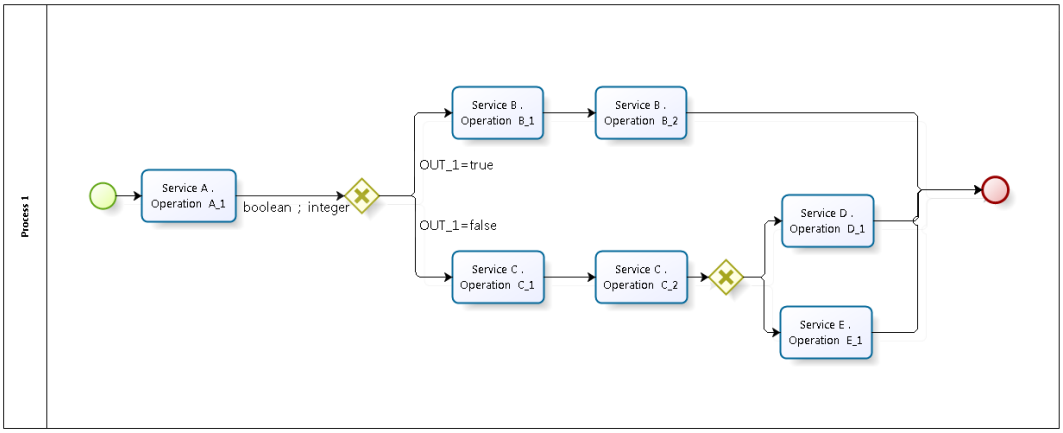

Figure 4.2: Process Example 1 - Bizagi Modeler

different things depending on its source. If it is a task, it represents its output data, on the other hand if it is a gateway, it contains the condition that makes the execution follow that direction. We will see on the input data generation section, Section 4.5, how decisive this edge notes will be. Once the process is created, the next step is exporting the model file as a BPMN file as shown on fig. 4.3.

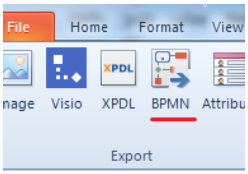

Figure 4.3: Bizagi Modeler BPMN Export

This will create a XML-like file with all the data required to rebuild the model outside the Bizagi Modeller application. As a regular XML file, all information is structured within nodes.

Listing 4.2 shows an example of the code produced for a task and a sequence flow, which in our approach refers respectively, to an operation and an edge. Here, we can see it stores a lot of layout related information, the *BizagiProperties*, but it also contains the elements that allow us to build the graph.

The modelling process is completely external to SOA-TA. As far as it is concerned, there is no difference from modelling with Bizagi or other software, as Enterprise Architect (software) (EA) or Microsoft Visio, as long as the BPMN file format constraints are respected.

Listing 4.1: XML Task Code

```
<task id="Id_90905bfe-613d-4e35-a25f-2c2325100f5d" name="Task 1">
<documentation />
<extensionElements>
<bizagi:BizagiExtensions xmlns:bizagi="http://www.bizagi.com/
    bpmn20">
<bizagi:BizagiProperties>
<bizagi:BizagiProperty name="bgColor" value="#ECEFFF" />
<bizagi:BizagiProperty name="borderColor" value="#03689A" />
</bizagi:BizagiProperties>
</bizagi:BizagiExtensions>
</extensionElements>
<incoming>Id_def559f5-6523-499c-9cc5-b70df19dcc9c</incoming>
<incoming>Id_1b74592c-db3d-4e32-a2a0-175edd1bbc63</incoming>
<outgoing>Id_1c0da27b-7553-40a7-8f60-5f5854f159d0</outgoing>
</task>

<sequenceFlow id="Id_1b74592c-db3d-4e32-a2a0-175edd1bbc63"
      sourceRef="Id_fe92eae0-5086-46da-b97e-a7457e9358ba"
      targetRef="Id_90905bfe-613d-4e35-a25f-2c2325100f5d">
<documentation />
</sequenceFlow>
```

4.2.1 Tibco Logs Exception Case

Besides reading BPMN files, WinTrust requested that we could also build the process description graph from a log file produced by the Tibco platform [36]. Tibco platform is a middleware responsible for managing SOA's execution and performance. It saves execution logs making it possible for us to trace steps and extract data to build a process graph directly from production. Including this option would be an exception case and that is why we are not focusing our attention on it, nonetheless all the shown procedures are common to the both ways of modelling processes and constructing graphs.

4.3 Graph Generation

The graph generation is the following stage of SOA-TA. As we have seen on the previous section, Bizagi Modeler exports a file with all the information we need to create a graph. Using a XML parser, we will store the nodes and edges information in a way suited to preform graph searches. For example, from Fig.4.2 we would generate a graph like the one portrayed on the left side of Fig.4.4.

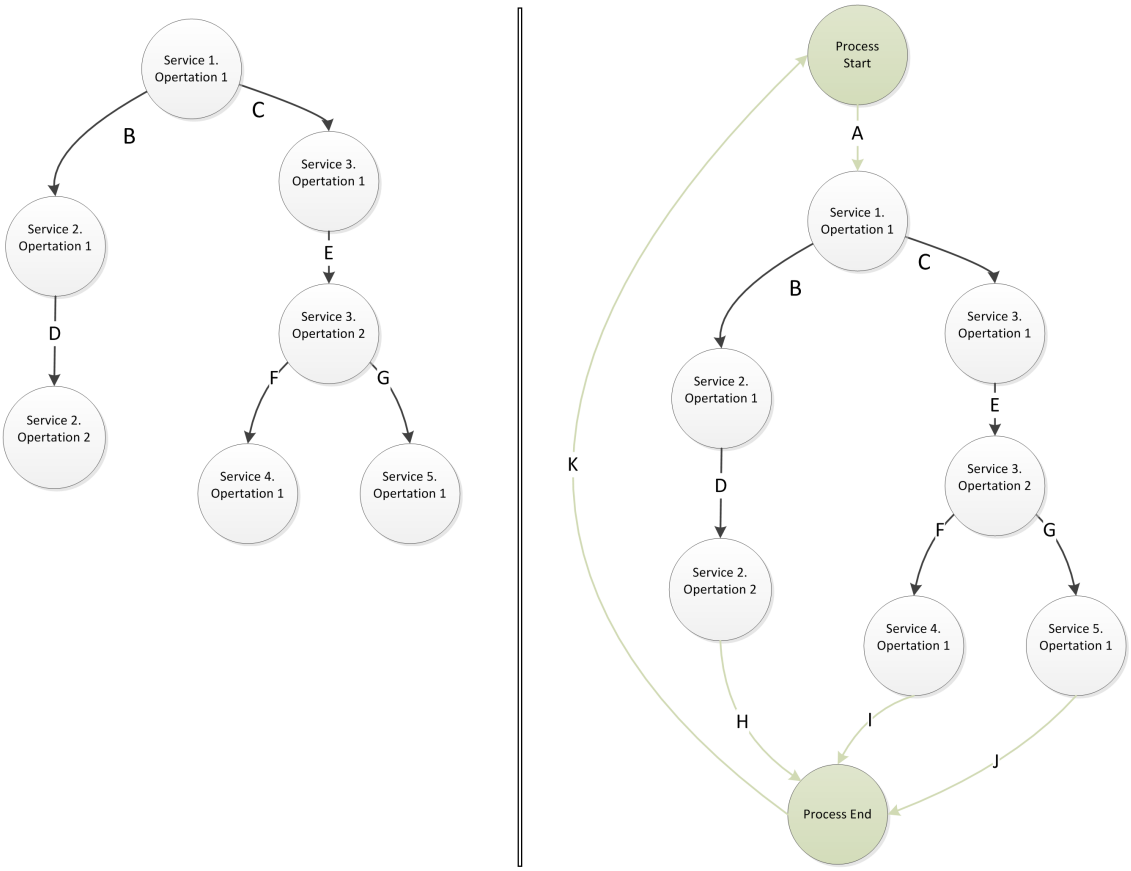

Figure 4.4: Bizagi Process Example 1's Graph; Left – Before modification; Right – After correction

As we can see, upon graph generation some alterations are made. On the right side of Fig.4.4, stands the graph on its final form where two nodes were added and their respective connections to the other nodes. These represent the start and the end of the process, which will be necessary to the path-finding algorithm to run. Another addition to the graphs is an edge from the "process end" node to the "process start" node. Although it might seem strange to make this change and create a loop, this is also a requirement for the algorithm described in Section 4.4. The graph will be stored in a way allowing its traverse and with edges and nodes easily retrieved by an id attribute. Once we have the

graph in the right conditions like the one on the right side of the picture, it is time to generate test paths.

4.4 Test Path Generation

The goal of the stage of the process is to produce a test case for each business process. It is assumed that the SOA – Test Accelerator user already chose what level of coverage he wants, the $0 - switch$ or the $1 - switch$.

In the $0 - switch$ coverage case, the system must generate a test case in which all the transitions should be tested at least one time. In the graph terms, this means coming up with a path covering all edges at least once. This is analogue to probably the best-known combinatorial optimization problem [37], called the "Chinese Postman Problem" or the "Route Inspection Problem". In this classical problem, we assume there is a postman who needs to deliver mail to a certain neighbourhood. The postman is unwilling to walk unnecessary miles, so he wants to find the shortest route through all the neighbourhood.

If we solve this problem within our graph we would have a path covering all transitions, and thus, covering all service operation calls. To solve this problem, we will use the strategy documented on Costa [38]. For this strategy to work, based on finding Eulerian circuits, it requires that all nodes of the graph be accessible from any one of them, and that is why we added the extra edge, in Section 4.3, connecting the last and the first nodes.

In the $1 - switch$ coverage case, this approach will not work only for itself. Here, the system must generate a test case in which all pairs of consecutive transitions (*length* 2 sequences) should be tested at least one time.

To solve this, we came up with a solution based on what is known in graph theory as path contraction [39]. Path contraction is the process of taking all the edges of a path and contracting them to form a single edge between the two endpoints of the path. If we take all paths of length two of a graph, in other terms, all pairs of consecutive transitions, and form a new graph with its contracted paths, we would be able to use the Chinese postman problem's solution again. The idea is to turn each pair of two consecutive edges of the graph into one edge and still maintain graph consistency. The pseudo-code description of the algorithm we propose to do this is described in the listing Algorithm. 1.

```
Input: original_graph
Result: new_graph
foreach edge e1 in original_graph do
    if new_graph not containsNode (id equals e1.id) then
        create_node(id="e1") in new_graph;
    end
    foreach edge e2 consecutive_to e1 do
        if new_graph not containsNode (id equals e2.id) then
            create_node(id="e2") in new_graph;
        end
        create_edge(from e1 to e2)in new_graph;
    end
    return new_graph
end
```

Algorithm 1: Path Contraction Algorithm

This algorithm essentially relies on two simple steps. First, we take every edge of the original graph and transform it to a node in the new graph. Then, connect only the nodes which, in the original graph, were consecutive edges. We will end up having in the new graph, edges which represent two consecutive edges of the original one. Solving the Chinese postman problem in this new graph, will provide the paths in which all pairs of consecutive edges are tested. Figures 4.5 and 4.6 depict an example of what this algorithm accomplishes. Simplest as it might be, it is trivial to see the strict relation between the original and second. Each edge of the second gathers data from two edges of the first. To conclude, if only $Chow - 0$ level is required, we apply Chinese postman problem to the original graph, if $Chow - 1$ is required, we first create a new graph with the given algorithm and only them use the Chinese postman problem's results.

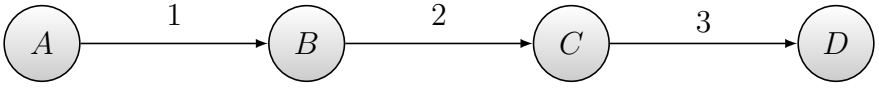

Figure 4.5: Example graph - 4 Before

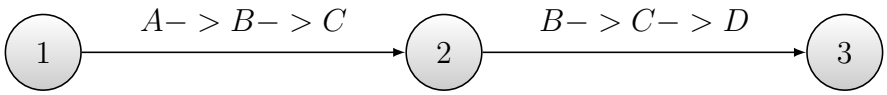

Figure 4.6: Example graph - 4 After

4.5 Input Data

Once the execution graph paths are generated by the methods described in the last sections we need to execute them. To test a specific path, a number of service operations need to be executed. In regular cases, most operations require some type of input data.

Imagine, for example, a purchase payment task in which you would have to provide your credit card number, then the card holder's name, and then the security code. If there were three separate web service operations to validate the format of the input fields, they would require three sets of data.

Our goal in this stage is to provide semi-automatically input values which will exercise the paths previously created.

The only way this job is feasible is by whitening up a bit our black box testing. In other terms, the service owners will have to give a description of what are the possible inputs and outputs of a service. If for example, we had three service operations in a row and if we already knew that the only way of the task execution to go through all of them was if the first had the input "false" and the second and third, the input "true", we would automatically feed them with this combination. That is the reason why we need the edge conditions mentioned on Section 4.2. When there is a fork in the process model, we need to know what is the condition the last operation output needs to meet to go one way or another, as shown on Fig. 4.7.

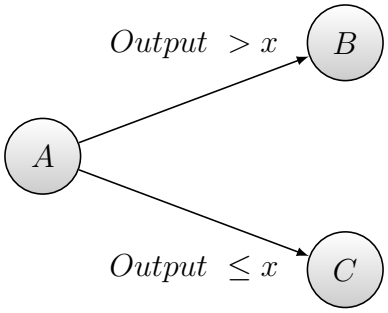

Figure 4.7: Example graph - 5

What we are going to do is prompting the service owners to build the BPMN with

the conditions of each decisive edge and then solve the whole path with the restrictions gathered. When we have all the path operations's output restrictions it is time to discover what inputs will fit, i.e, what input data will produce results matching those restrictions. To do this we do an auxiliary task of calling each step operation exhaustively (the number of executions can be restricted) from whithin our code to get a collection of input and output mappings. Then, we need to solve the problem of assigning a value to each input parameter in the whole path without braking the restrictions on the edges and without making decisions that would invalidate the path in steps to come.

This is what in computer science is known as a Satisfiability Modulo Theories problem [40]. It is a decision problem for logical formulas with respect to combinations expressed in classical first-order logic [41].

There are a number of works done in solving this kind of problems but we used a tool named Z3, a relatively new Satisfiability Modulo Theories (SMT) solver from Microsoft Research [41].

When given the correct data, the solver checks to see if the formulas are satisfiable of not, and if so, it provides with a model which satisfies it [40].

Once we have all the test cases, all the input to exercise those test cases we are ready for script generation phase.

4.6 Test Script Generation

The following stage of SOA – Test Accelerator is the generation of a script to run on Apache JMeter tool. As mentioned in Section 2.2, JMeter is an open source and pure Java application designed to load functional test behavior. Their well-documented API [42] allows easy script creation and execution within any java program. Our goal is to combine data gathered in previous stages and create test plans to exercise the test cases. JMeter has a sampler specifically designed to work with web services as shown on Fig. 4.8.

4.7 Test Script Execution

To execute tests on a service oriented system, or even to make use of them for that matter, there are two main approaches, using SOAP or Representational State Transfer (REST) (Representational State Transfer). Since the available webservices we had to test were

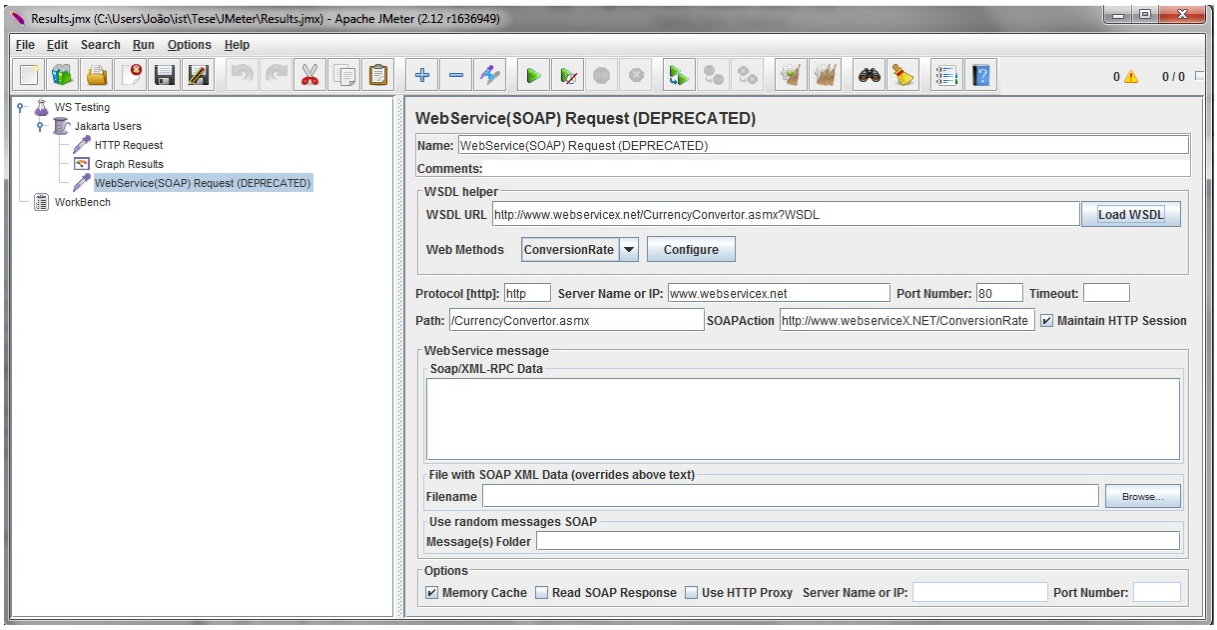

Figure 4.8: JMeter Web Service Sampler

SOAP based and since JMeter only supports this kind of requests, the choice was easy. However, if we chose REST, we believe our results would be as good and few things would change. We are not going to focus on the REST vs SOAP debate, as this is a big matter [43, 44] and not that much relevant in our work.

The .NET framework supports web service calls using standard HTTP requests with a manually built soap envelope inside, however, since the JMeter script was already created, it was simpler to just start a Jmeter process and pass the script as an argument. This is what we end up doing, start a thread to execute the scripts and wait for it to be done.

4.8 Reading Results

There are several paramenters one can measure when executing tests on webservices, such as

- Response headers
- Response data
- Success/Failure boolean
- Timestamp
- Hostname

- Response data type
- Latency
- Number of threads executing the test
- Elapsed Time
- Byte count
- Encoding
- Etc

SOA-TA is able to detail on the script which of this items to save. As of now, we are saving response data, success, timestamp and the elapsed time. If, in the future another information is relevant, minimum change is required in order to do so. Once the test execution finishes, SOA-TA picks up the file JMeter writes the results to, goes through it all and saves it in the database, parameter for parameter. In the end the database will store not only the data from which the tests were built (steps,inputs, etc) but also all its executions, with its pertaining data.

4.9 Summary

Essentially, the way we addressed the issue of developing the product Wintrust wanted, was to divide it in five stages. The model creation, responsible for parsing a process description file, specifically in BPMN format, and extracting all the required applicable data (i.e. services,operations and execution flows), while ignoring and removing all the notation's irrelevant artefacts.

The graph generation is accountable for the conception of a directed graph, expressing the process execution flow. Assuming a classic notion of graph, with a set of nodes and edges connecting them, each node would conceptually represent a service operation, while the edges would stand for the calls between them. This means that, if the graph contains an edge from A to B, B is an operation coming after A.

The test case generation is the component where we take the previously created graph and come up with complete execution paths, starting in the initial task/operation to the finish. Depending on the coverage level chosen by the user, we use an algorithm to solve

the commonly known as "Chinese Postman Problem" described in 4.4, to ensure all the transitions are tested at least once. If the user chooses a more comprehensive test method, this algorithm is used alongside a path-contraction algorithm, also described on section 4.4.

The following, Input Data Generation stage, is where we use some AI techniques, namely the ones related to SMT problem, to discover input data in order to exercise the test paths earlier generated. Given a set of restrictions, like, for example a person's age parameter must between 0 and 100 and so on, we will discover a value for each parameter in order not to become the path execution infeasible. To solve these decision problem we use a tool from Microsoft called Z3 solver, which aids us in this process of deciding which values to assign to input parameters.

In the script generation phase we gather all the data from the process's graph to the input data, to the services WSDL's, to the relation between inputs/outputs, and generate a script to execute the tests. We chose the format used by Apache JMeter, and in the background this is the tool that actually runs the tests which is the following stage. In the end, the results file, containing a set of useful information about the execution, is read, parsed and saved in the database.

5

Implementation

In this chapter, we are going to go through the implementation of SOA-TA. As previously stated, it is based on a on-line platform to be accessed via web browser and here we are going to detail development decisions and all the components needed to its functioning. Since the early stages of development, the stakeholders wanted the product to be always available, using cloud computing. This was going to allow sales in international market, as SaaS. It was also required that the solution found ought to be as universal and general as possible. Every future client who wanted to use SOA-TA should not be restricted to any operating system, platform, brand of processor/machine or programming language. As a way to ensure meeting all these requirements, we have chosen to build SOA-TA as a web application.

With a web-browser based client-side, there is no need to distribute and install software on potentially thousands of client computers and also obliterates the cross-platform problem. As any web application, functionality is supported through a set of dynamic web pages, which in the end run a couple of scripts and retrieve information from the server.

Diagram on Fig.5.1 shows the flow of communications in SOA-TA. Here, as in any other web application, different functionalities are separated in modules. Assigning responsibilities individually will allow elements to be loosely coupled and enhance security, since only the server's "Data Access" element will approach the database directly.

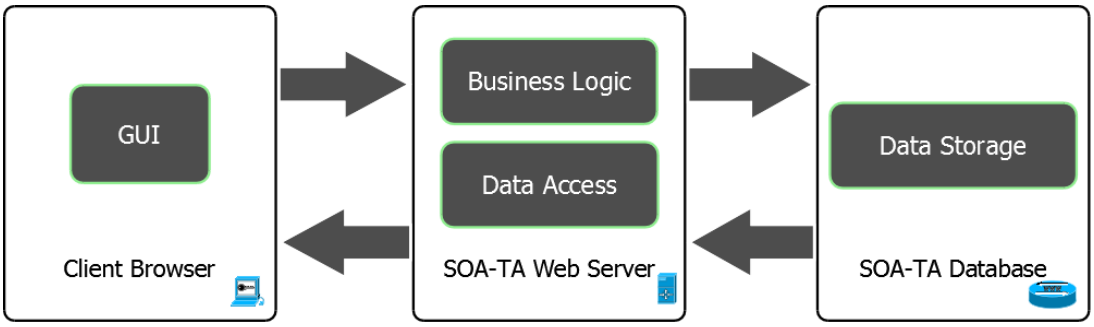

Figure 5.1: SOA-TA Communications Diagram

5.1 User Interface

Interaction with the user is a critical part of SOA-TA's functioning. Although it was not our major concern when developing this thesis, we had to make sure all our user related procedures were relatively simple and easy to use. Leonor Bento, an INESC-ID (now Wintrust) colleague was in charge of the interface design and implementation stages. A few simple step is what it takes to get the application going.

First, there is the login screen, each client has is own dedicated database so for them to access previous uploaded data, a correct login is required. A menu will appear on the left as shown of Fig. 5.2. To start using the test accelerator itself, the user navigates to the paths discovery tab and uploads a BPMN file containing a description of the processes to test. As the file loads, SOA-TA will automatically generate the test paths, store them in the the database and show them on the page, separated in three folders: $Chow-0$, $Chow-1$ and $Chow-N$. The first two will have the unique paths necessary to achieve that specific coverage level and the later will have paths that both have in common.

Once this is completed, the user must navigate to the "Data Discovery" tab enter each service WSDL link. Then he can link each task in the process with a service operation. This can be done in two different ways, automatically, where names of operations and services will be matched, or manually, where two lists will appear for matching one from each.

Following this process, the user can explicitly say what input values he wants to provide in the test. These can be previously set variable values, one of the possible values predefined in the WSDL(if it has any), or even some value from a previous operation result. This screen shown on the lower right side of 5.2 is where the user can make this options. All the values not set are going to be auto-generated as we have discussed on Section 4.5. The main menu, on the left side of the screen also displays two more options,

the test suites and the test sets.

In the test suites tab the user can edit the auto generated tests. This was created to provide the user a greater control to process. The test sets tab is where the actual testing takes place. Here the user can execute tests, see and compare results.

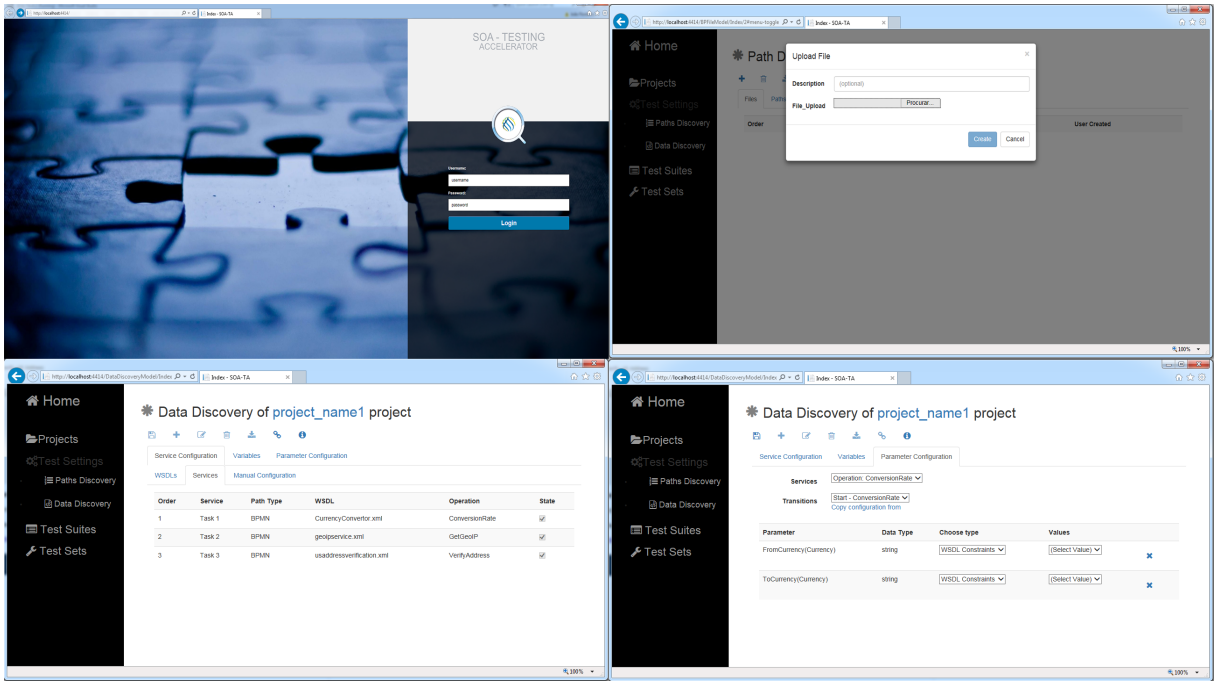

Figure 5.2: SOA-TA Usage Screenshots

5.2 Business Logic

In software engineering, business logic or domain logic refers to the part of the program that encodes the real-world business rules that determine how data can be created, displayed, stored, and changed, essentially the business core. If, for example, we were building a bank application, this could be where the instructions to preform money transfers, deposits, withdrawals, would be written. To interact with it, users would have to use the operations it provided. In our case, this is where the following functionalities are implemented.

- Coverage - The classes needed to accomplish path generation, which as been extensively described in section 4.4. The algorithm to solve the route inspection problem and the implementation of the Algorithm 1 shown on Section 4.4 are dealt with here.

- Data Discovery - Within this folder stands the reference for Microsoft's Z3 SMT solver and all the wrappers and data objects needed to interact with it. All the process of finding suitable input for the test is preformed by these objects.

- Parsers - There are two different parsers. One accountable for reading business process files in BPMN format and generating a directed graph from it. The other is responsible for going through WSDL documents and extract all the useful information from it, such as operations, data types, endpoints, namespaces etc. Both rely on reading the info from a XML source.

- Script Generation - In here we have gathered all the classes used in the generation of the JMeter XML-like scripts with all the test procedures. It encapsulates essentially a XML writer with a specific format.

- Script Execution - Here we grouped the code responsible for calling a JMeter process to actually execute de test. The process runs a command line tool which saves the output to a predefined folder. This activities are transparent for the user.

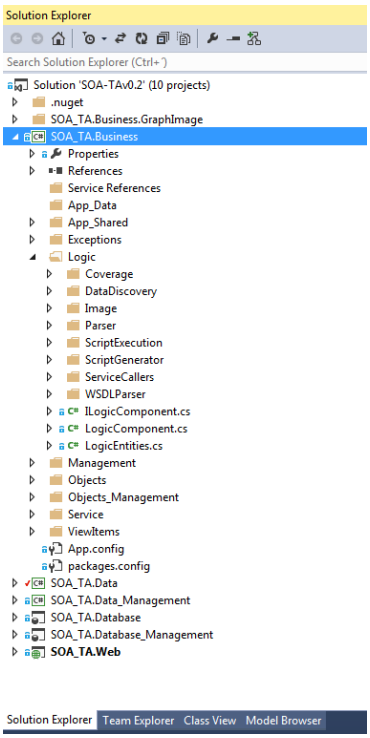

Figure 5.3: Business Logic structure

Fig. 5.3 displays the SOA-TA's folder structure. A clear separation of responsibilities can be seen. The folder named Logic collects all the modules mentioned above.

5.3 Database and Data Access

All the SOA-TA produced data is persisted in a relational database in Microsoft SQL Server. Its schema, shown on fig.5.3 represents all the tables and fields SOA-TA stores when it is properly functioning. All the user related data is not represented here since it is stored in a different database, which deals will all the security related issues as password hashing, access control and authentication in general. Most of these options come already taken care of in Microsoft ASP.NET framework, we just opted in this stage to use the standard verifications. If, in the future a stricter security policy needs to be enforced, minor changes are required. Data access is done via an object-relational mapper, Microsoft's Entity Framework, that allows programmers to work with relational data using the domain-specific objects. It also encapsulates all the transactional methods and deals with client concurrency. A strict relation between the database entities and the objects, managed by the mapper, makes the process of saving changes and updating values much easier.

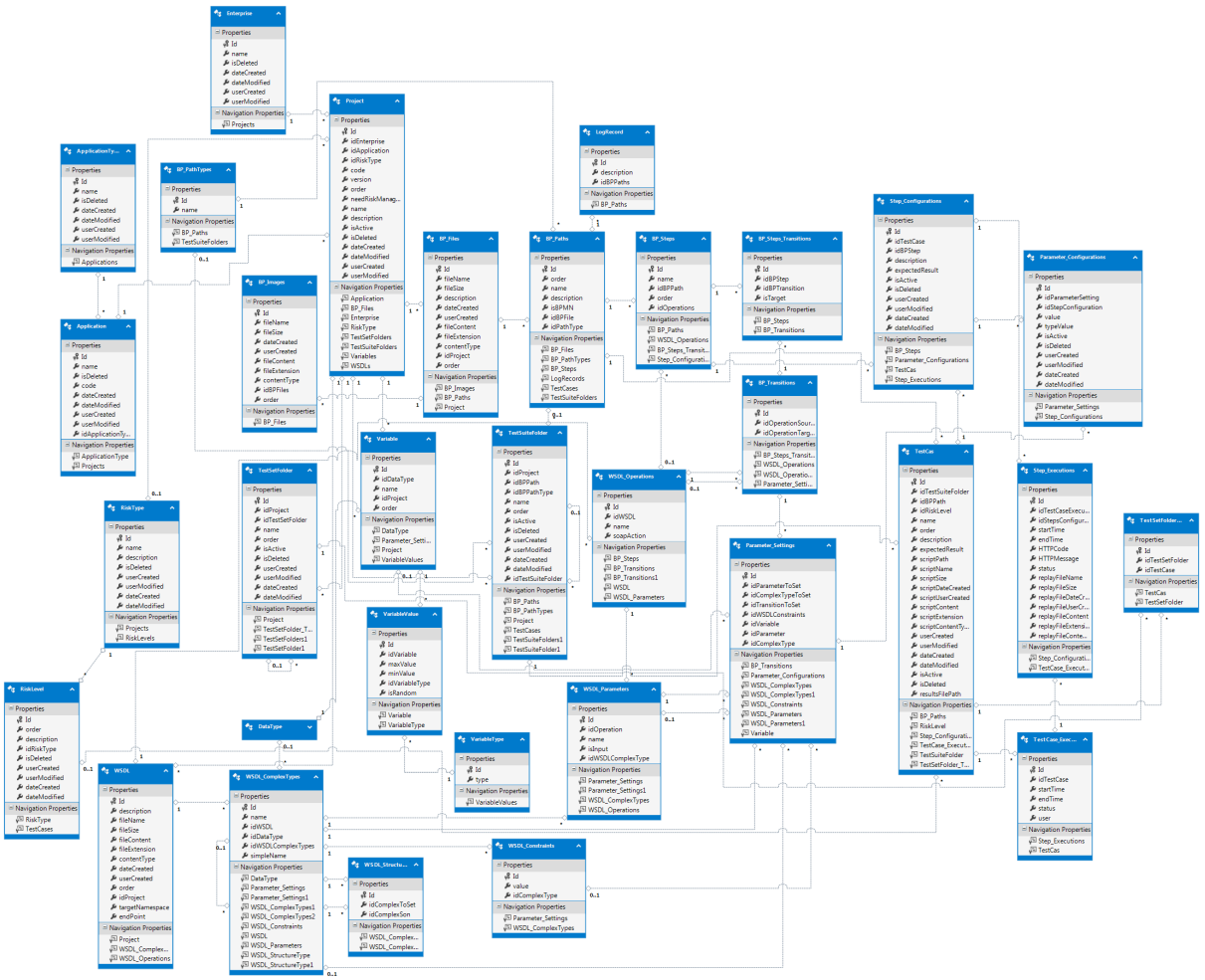

Figure 5.4: Database schema

5.4 Summary

SOA-TA was build up to be an on-line platform, using some usual standard components and procedures, such as the model view controller (MVC) architectural pattern to ensure responsibilities separation, having the database access done trough a persistence framework and using a browser as client. We have used Microsoft technologies, namely ASP.NET and Entity Framework, due to Wintrust's will to integrate our platform with another set of projects also build using the same technology. Related to the business logic core, encompassing elements as the coverage analysis and data discovery algorithms, it as also been done on regular C# classes, loosely coupled from rest of the platform's functions. As it was already mentioned the data access is made trough the Microsoft's Entity Framework, which eliminates de most of the data-access code we would normally need to write by means of an object-relational mapper. The database was build using MS SQL, and its schema is shown on section 5.3.

6

Evaluation Methods

In terms of evaluation methods, our initial strategy was to measure the amount of time an experienced tester would usually take to create test plans that exercise the same transitions SOA-TA tests. However we soon realized this is an impractical idea. Not only the testers shared different methods but, unless they are testing a small process, their goal usually is not to guarantee a formal and specific coverage level. So we end up evaluating our solution in two different perspectives, quantitative and qualitative.

Regarding quantitative evaluation, we tried to evaluate the correctness, minimization and amount of time spent by path generation phase, and the quality, i.e. utility and feasibility of the input data produce in the stage after that. We have also created and executed the test scripts to show the final end-to-end framework results.

Regarding qualitative evaluation the methods are more subjective. Whilst one can formally measure times and numeric outcomes it is harder to tell if certain tool facilitates and eases an human-preformed process. SOA-TA was supposed to accelerate the test process, hence its name; however in some cases it becomes more than an accelerator since it preforms tasks humans that could not, such as guarantee coverage levels for huge processes as we are going to witness in the chapter 7. This means that when we are considering adding new possibilities, as we have done, the tool clearly eases the process, even so it is supposed to be used by humans who are used to do things a certain way so usability was a major concern. Chapter 7 contains some fictional, yet realistic, and real-life example processes and the outcomes we got with them.

7

Results

In this chapter we are going to describe the evaluation preformed on SOA-TA. As it was already discussed in previous chapters, its operation depends on several components worthy of a independent assessment. For example, it is trivial to realize that if the test path generation does not provide feasible or correct test paths, all other components, such as input generation and test script creation will be invalidated. Having this matter in consideration, we have chosen to present the following results grouped in two sections, one with the coverage-based test path generation results and the other with system's end-to-end testing (focusing specially on the sections other that path generation).

7.1 Path Generation Results

Regarding the path generation stage, we are going to show some results on process examples. There are, as it was mentioned on section 4.2.1 two types of sources SOA-TA can use to build the directed graph from where paths are extracted, BPMN files and Tibco logs [36]. In the following examples we have grouped a small set of each to demonstrate our outcomes.

Fig. 7.1 shows a first example of a small, yet real-life process description. This graph was extracted from a Tibco log from the Transportes Aéreos Portugueses (TAP) - Transportes Aéreos Portugueses, a Wintrust client. It portraits a simple flow with just one fork after the $addMultiElementsPNR$ task. All graph figures presented here originating from Tibco logs were automatically generated by a Quickgraph for C# [45]. For this example, table 7.1 demonstrates the two distinct generated paths. In a small and simple example like this it is trivial to see that are two paths exactly are required for us to test

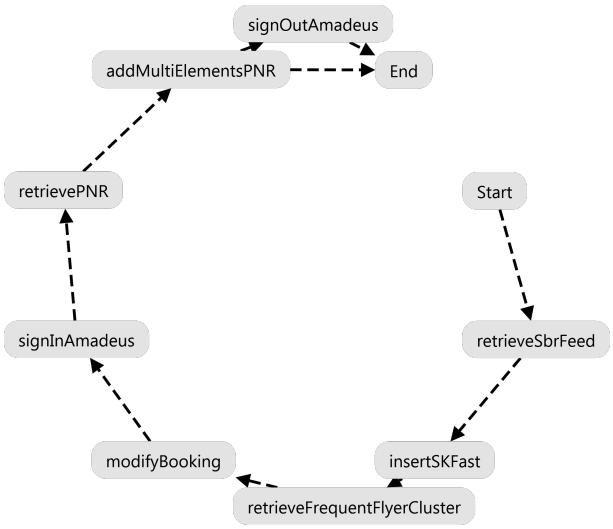

Figure 7.1: Example graph 6

all transitions of size 1, $0 - Switch$ coverage.

Step	Operation Name-Path 1	Operation Name-Path 2
1	retrieveSbrFeed	retrieveSbrFeed
2	insertSKFast	insertSKFast
3	retrieveFrequentFlyerCluster	retrieveFrequentFlyerCluster
4	modifyBooking	modifyBooking
5	signInAmadeus	signInAmadeus
6	retrievePNR	retrievePNR
7	addMultiElementsPNR	addMultiElementsPNR
8	signOutAmadeus	

Table 7.1: Generated Test Paths - Example graph 6

Fig. 7.2 shows a much more complex and also real-life process description. This graph represents the operations preformed by TAP's web services to make a ticket reservation for a staff member. It presents a structure flow with so many transitions between operations that makes human coverage analysis almost impossible. For all these transitions, $0 - switch$ coverage generated the paths contained on table 7.2. We have decided not to present the results for $1 - switch$ level here, as it is a big and repetitive table so we have included it in the attachments.

It is difficult to manually check, in this case, but if we take a closer look we will see that no edge was left untested. All transitions were tested at least once.

Regarding the time consumption, we have registered that for SOA-TA to generate all 7 paths, the one shown on 7.2 and six more to achieve the $1 - switch$ level, it took about 14 milliseconds, 4 on the first path and 10 on the remaining. For all other tested examples

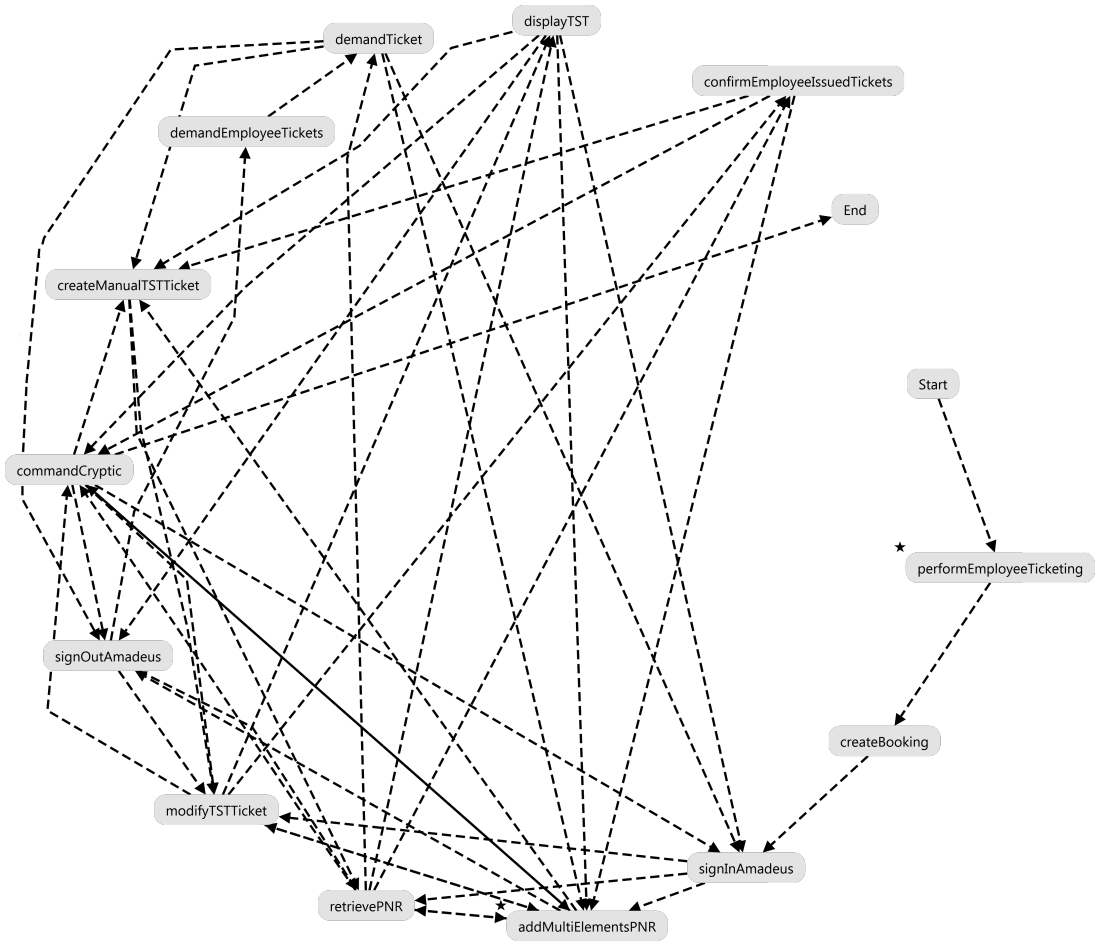

Figure 7.2: Example graph 7 - TAP Staff Booking

we have had similar results in terms of scale magnitude. Values in the order of tens of milliseconds make the time factor almost negligible, specially when we are talking about an on-line platform where network delays are much larger than that. Also, the database access and duplicate path analysis ended up taking a lot more time that the coverage algorithms.

We have also tested SOA-TA with process descriptions from Bizagi. Figure 7.3 shows an example of a simple process where tasks 2 4 and 5 are on purpose connected in a loop. We have created this situation to ensure the system knew how to handle this kind of process artefacts. In this case, the transition(s) involved in the loop also need to be tested which can cause operations in paths to be executed twice or more times in a row. Table 7.3 contains the generated paths for the example on Fig.7.3.

When taking a closer look at Fig. 7.3 it is possible to identify a single path to go from the process star to the end covering all transitions. This was the first automatically generated path, to achieve $Chow-0$ coverage, as we can see on table 7.3. When it comes to transitions of length 2, not all of them are tested with this single path, for example,

Step	Operation Name	Step	Operation Name
1	performEmployeeTicketing	31	modifyTSTTicket
2	performEmployeeTicketing	32	displayTST
3	createBooking	33	createManualTSTTicket
4	signInAmadeus	34	retrievePNR
5	retrievePNR	35	displayTST
6	addMultiElementsPNR	36	commandCryptic
7	commandCryptic	37	addMultiElementsPNR
8	signInAmadeus	38	signOutAmadeus
9	retrievePNR	39	demandEmployeeTickets
10	demandTicket	40	demandTicket
11	signInAmadeus	41	addMultiElementsPNR
12	modifyTSTTicket	42	createManualTSTTicket
13	displayTST	43	retrievePNR
14	signInAmadeus	44	confirmEmployeeIssuedTickets
15	addMultiElementsPNR	45	addMultiElementsPNR
16	retrievePNR	46	commandCryptic
17	demandTicket	47	modifyTSTTicket
18	signOutAmadeus	48	confirmEmployeeIssuedTickets
19	retrievePNR	49	createManualTSTTicket
20	demandTicket	50	modifyTSTTicket
21	createManualTSTTicket	51	commandCryptic
22	retrievePNR	52	signOutAmadeus
23	displayTST	53	retrievePNR
24	addMultiElementsPNR	54	confirmEmployeeIssuedTickets
25	addMultiElementsPNR	55	commandCryptic
26	retrievePNR	56	commandCryptic
27	displayTST	57	createManualTSTTicket
28	signOutAmadeus	58	retrievePNR
29	modifyTSTTicket	59	commandCryptic
30	addMultiElementsPNR		

Table 7.2: TAP Staff Booking Generated Paths - Chow-0

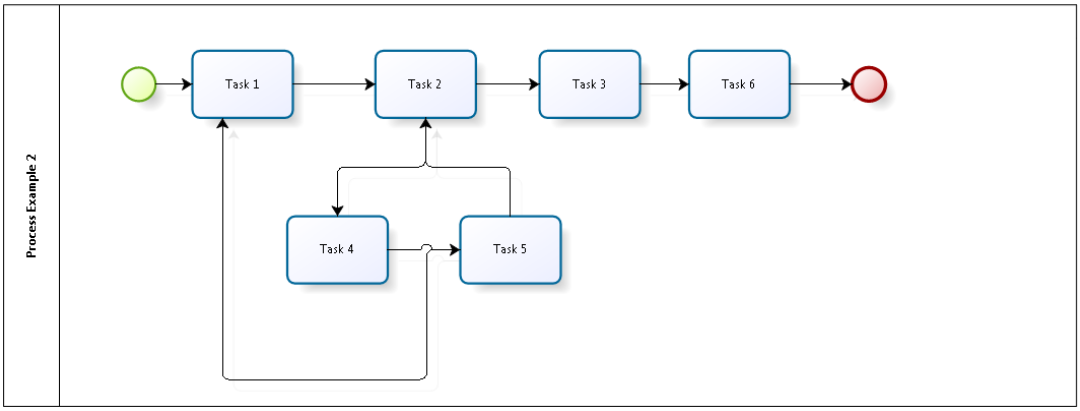

Figure 7.3: Process Example 2

	Chow-0	Chow -1		
Order	Path 1	Order	Path 2	Path 3
1	Task 1	1	Task 1	Task 1
2	Task 2	2	Task 2	Task 2
3	Task 4	3	Task 4	Task 4
4	Task 5	4	Task 5	Task 5
5	Task 1	5	Task 2	Task 1
6	Task 2	6	Task 4	Task 2
7	Task 4	7	Task 5	Task 3
8	Task 5	8	Task 2	Task 6
9	Task 2	9	Task 3	
10	Task 3	10	Task 6	
11	Task 6			

Table 7.3: Process Example 2 - Generated Paths

the transitions from $Task$ 1 to $Task$ 5 trough $Task$ 2, and from $Task$ 1 to $Task$ 3 trough $Task$ 2 were not tested. To get a scenario when these two transitions were tested, the two paths on the two rightmost columns of table 7.3 were added.

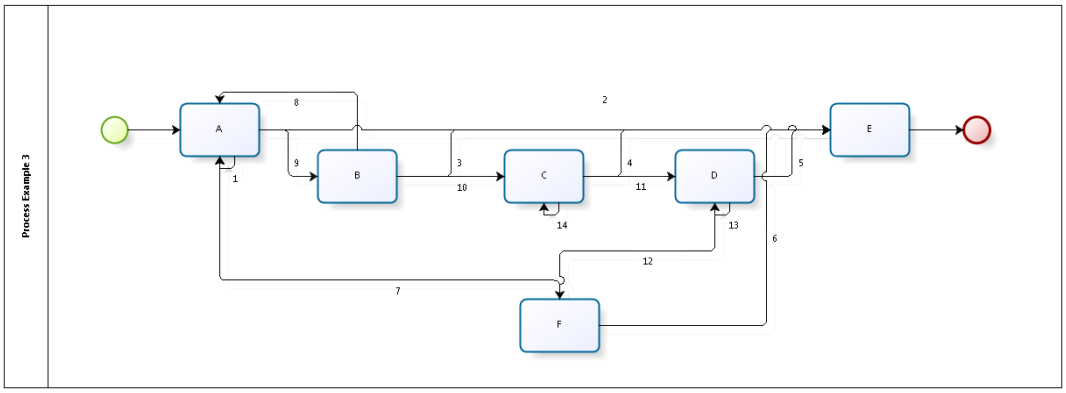

Figure 7.4: Process Example 3

The next example in Fig. 7.4, is a process that lead to a graph similar to what we have seen on Fig. 3.3 in section 3.1.3. As the transitions are the same we are not going to replay the path tables 3.1 and 3.2. Nonetheless we have manually check all the results on this tables and other examples so we can guarantee the correctness of SOA-TA's path generation algorithm.

The last result, on Fig. 7.5 we are going to show is from a process description with some extensive usage of BPMN artefacts that are not relevant for paths generation such as throw or catch events, diverging and converging gateways or tasks groupings. This was made specially to test SOA-TA's ability to properly build the graph and run the coverage

algorithms without letting the potentially large set of notation data interfering with the results.

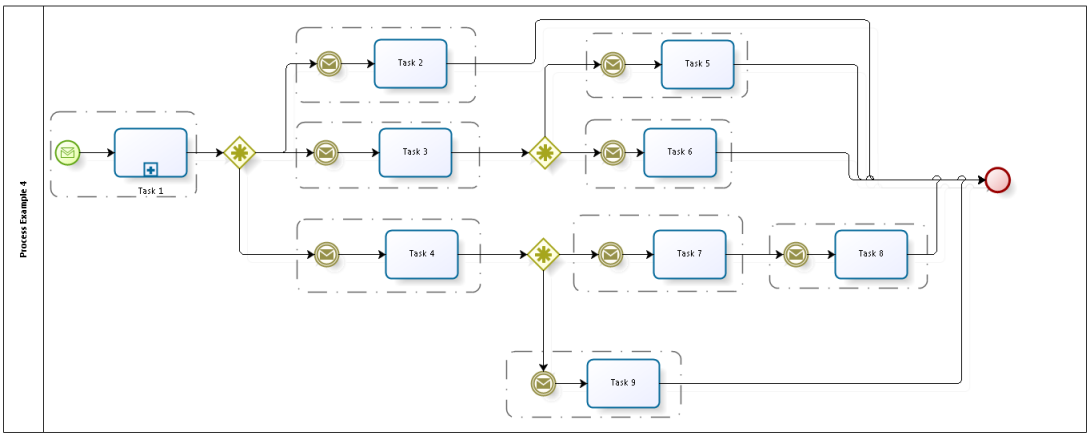

Figure 7.5: Process Example 4

Order	Path 1	Path 2	Path 3	Path 4	Path 5
1	Task 1	Task 1	Task 1	Task 1	Task 1
2	Task 4	Task 3	Task 3	Task 4	Task 2
3	Task 7	Task 5	Task 6	Task 9	
4	Task 8				

Table 7.4: Process Example 4 - Generated Paths

Table 7.4 contains the paths obtained when adding Process example 4 to SOA-TA. We have not separated the ones required to Chow-0 or Chow-1 because in this case, both levels would generate the same results, or in other words a higher exhaustiveness level would not get better results.

7.2 End-To-End

To test our system from end-to-end we need to start from a process or task description, then model in BPMN, upload the file to SOA-TA, ask it to generate test paths, provide static input, and then execute the tests and see if something went wrong. Let us now follow the process from the beginning with an example. First, consider the following user story which will be our starting point.

"A few years ago I bought a few stock shares of *Apple computer* and ever since its price boom I have considered selling them and buy a house in New York, a long time dream. However, first I will have to make a few checks to see if this is the right time to do it.

First I will check how *Appl* stocks are doing using $getStockValue$ service and how much stocks I would need to have to get to $500k$ dollars. If the values are too far away from my personally established threshold of $400k$ dollars a share, I will wait for some other time. Otherwise I want to use the currency converter web service, $ConvertUSDtoEUR$ to see how much euros I would have to put in to get to that budget. If I have to invest less that $100k$ euros, I will even check for a house at walking distance from central park, something like $500meter$ converted to *yards* (Americans don't fancy the metric system) with length converter service, *convertionRate* . Either way, I want to know how is the weather back there, I will use $GetCityWeatherByZIP$ operation of the weather web service, and... that's right, they use Fahrenheit, I'll have to get it back to Celsius with temperature converter service, $ConvertTemp$."

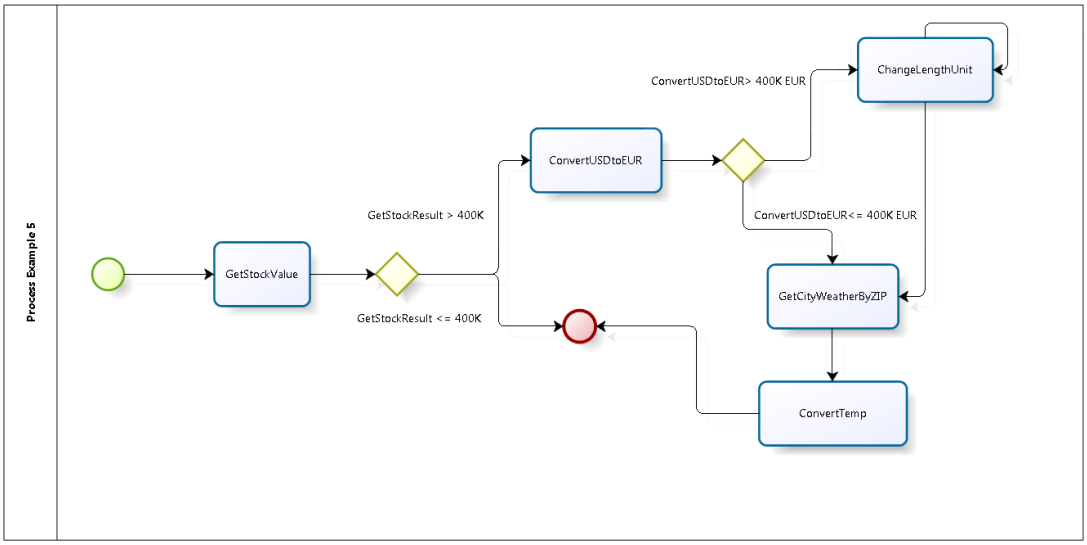

Figure 7.6: Process Example 5

This fictional user story would produce a process description like the one on Fig. 7.6. Note that we have included the correct operations's name for each task, this will simplify the path generation process. The services mentioned on the example are working services and can be found on the following endpoints.

- `http://www.webservicex.net/stockquote.asmx?WSDL`
- `http://www.webservicex.net/CurrencyConvertor.asmx?WSDL`
- `http://www.webservicex.net/length.asmx?WSDL`
- `http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL`
- `http://www.w3schools.com/webservices/tempconvert.asmx?WSDL`

Now, this part is concluded, we will upload this file to SOA-TA and make it generate the coverage paths. Generation phase comes up with 4 different paths which are shown on tables 7.5 7.6. Each one of these, has respectively paths generated from the lower and the higher exhaustiveness levels, $Chow-0$ and $Chow-1$. If we take a closer look we will that in the first, all transitions of size 1 are covered by the two paths, however without paths 3 and 4 from the table 7.6, not all transitions of size 2 would be tested.The path number 2, highlighted in bold is required for both coverage levels. Also note that, in cases of loops in the graph, as it is the case of the *ChangeLengthUnit* node, may produce test paths in which the same node gets tested three times in a row. This is conceptually accurate since the first execution comes from testing a previous transition and the first loop iteration and second and third executions come from executing the loop twice (transition with length 2).

Step	Path 1	**Path 2**
1	GetQuote	GetQuote
2	ConversionRate	ConversionRate
3	ChangeLengthUnit	GetCityWeatherByZIP
4	ChangeLengthUnit	ConvertTemp
5	GetCityWeatherByZIP	
6	ConvertTemp	

Table 7.5: Example 5 - Chow-0 Test Paths

Step	**Path 2**	Path 3	Path 4
1	GetQuote	GetQuote	GetQuote
2	ConversionRate	ConversionRate	ConversionRate
3	GetCityWeatherByZIP	ChangeLengthUnit	ChangeLengthUnit
4	ConvertTemp	ChangeLengthUnit	GetCityWeatherByZIP
5		ChangeLengthUnit	ConvertTemp
6		GetCityWeatherByZIP	
7		ConvertTemp	

Table 7.6: Example 5 - Chow-1 Test Paths

Now, with the paths already generated, we are going to automatically generate the input data necessary for the test to be run. As mentioned on section 4.5, we use a SMT solver to aid us in this matter. We are going to follow closely input data generation for $Path1$ from table 7.5 nonetheless we could have chose any of the other 3 since the process is similar.

So the path to generate input is as follows:

- $GetStockValue \rightarrow_1 ConvertUsdToEur \rightarrow_2 ChangeLengthUnit \rightarrow_3$
  $\rightarrow_3 ChangeLengthUnit \rightarrow_4 GetCityWeatherByZIP \rightarrow_5 ConvertTemp$

Adding the restrictions on the edges, we get something like:

- $GetStockValue \rightarrow (_{GetStockValueResult>400k}) \rightarrow ConvertUsdToEur \rightarrow$
  $(_{ConvertUsdToEurResult>400k}) \rightarrow ChangeLengthUnit \rightarrow ChangeLengthUnit$
  $\rightarrow GetCityWeatherByZIP \rightarrow ConvertTemp$

After assigning outputs of previous tasks to serve as inputs to other tasks and after assigning the static input value of 500 and 200 meters to $ChangeLengthUnit$ steps respectively and the the 10023 zip code to $GetCityWeatherByZIP$ we can start input discovery.

This will internally trigger the SMT solver which will take the two restrictions and build them in two first-order logic restrictions, like:

$(> \ GetStockValueResult \ 400000)$

$(> \ ConvertUsdToEurResult \ 400000)$

Then we would preform a set of unit tests to both $GetStockValue$ and $ConvertUsdToEur$ with random values adding the input output relation as a restriction to the solver. The following values were generated when executing these tests.

Input	Output	Input	Output
18 756	2098046,16	21 951	2455438,86
60002,66944	6711898,604	62 146	6951651,56
28 715	3212059,9	43 518	4867923,48
8 684	971392,24	85 889	9607543,54
42284,35611	4729928,074	61 124	6837330,64
1 242	138930,12	63561,58996	7109999,453
69 836	7811854,96	85331,07528	9545134,081
24 801	2774239,86	32 055	3585672,3

Table 7.7: GetStockValue - Unit test results (16 samples)

With the results from table 7.7 from the $GetStockValue$ operation, we would take all the outputs and feed it to $ConvertUsdToEur$. Table 7.8 shows the obtained outputs. Note, that our intention was to get an output from $ConvertUsdToEur$ of less than $400k$, we have clearly succeeded with most of these result, only one is bellow that threshold.

Input	Output	Input	Output
2 098 046,16	1 846 280,621	2 455 438,86	2 160 786,197
6 711 898,60	5 906 470,771	6 951 651,56	6 117 453,373
3 212 059,90	2 826 612,712	4 867 923,48	4 283 772,662
971 392,24	854 825,171	9 607 543,54	8 454 638,315
4 729 928,07	4 162 336,706	6 837 330,64	6 016 850,963
138 930,12	122 258,506	7 109 999,45	6 256 799,519
7 811 854,96	6 874 432,365	9 545 134,08	8 399 717,991
2 774 239,86	2 441 331,077	3 585 672,30	3 155 391,624

Table 7.8: ConvertUsdToEur - Unit test results (Input from table 7.7)

With all these results gathered, SOA-TA will join this results in form of first-order logic restrictions also. For each entry in the table 7.7 it will produce an implication and a logic OR restriction like the following:

$(=> (= \ GetStockValueInput \ 18756) (= \ GetStockValueOutput \ 2098046,16))$

$(or \ (= \ GetStockValueInput \ 18756))$

The same for table 7.8 and one for the relation beetween GetStockValueOutput and ConvertUsdToEurInput, like the folowing:

$(=> (= \ ConvertUsdToEurInput \ 2098046,16) (= \ ConvertUsdToEurOutput \ 1846280,621))$

$(or \ (= \ ConvertUsdToEurInput \ 2098046,16))$

$(= \ ConvertUsdToEurInput \ GetStockValueOutput))$

SOA-TA will then use all the restrictions given and will check the formula for satisfiability. If the formula is satisfiable, it can produce a model which will have possible value for all the unknown variables we might have. In this case, it produced the values stated below.

- $GetStockValueInput = 18756$
- $GetStockValueOutput = 2098046,16$
- $ConvertUsdToEurInput = GetStockValueOutput$
- $ConvertUsdToEurInput = 2098046,16$
- $ConvertUsdToEurOutput = 1846280,621$

In the end of this phase we have all the inputs necessary to create the script, from the ones manually set as the distance in the ChangeLengthUnit step or weather values in

GetCityWeatherByZIP which are less relevant in this case, to the automatically generated and validated by SOA-TA.

The process of script generation is simple. All we do is gather all these inputs, outputs, service operations and WSDL descriptions in a XML file with specific JMeter rules. Each step is an instance of the "RPC/SOAP sampler" which carries a regular SOAP envelope. To save an operation result to use it as input to the next step, we make use of a regular expressions extractor and save the result in a JMeter variable.

To execute the script, as it was already mentioned in section 4.7 we start a command line process and provide the fie path to be run. JMeter then, executes the script, saves the general results in a text file and the partial results of each step in a answer file. This file contains the usual XML contained in one SOAP response message. SOA-TA parses this data and stores results in the database. For the example we were testing JMeter produced the following resuls file.

timeStamp,elapsed,responseCode,responseMessage,success

1442508850,783,200,OK,true

1442509634,827,200,OK,true

1442510462,827,200,OK,true

1442511290,769,200,OK,true

1442512163,873,200,OK,true

1442512955,792,200,OK,true

The $GetCityWeatherByZIP$ step with 10023, a New York ZIP code, as input, returned 88º Fahrenheit, which after $ConvertTemp$ converted to 31.11º Celsius. For all other paths built from this task description a similar process would take place.

8

Conclusions and Future Work

As we have seen, service oriented architectures present many challenges on what testing is concerned. On the other hand, its major spreading and appeal makes these issues unavoidable. Our work does not aim to be the solution to all the problems faced when testing this kind of systems but is surely a good option when the intention is to speed up the process whilst maintain all test robustness.

To our knowledge, there were no options on the market to automatically certify SOA tests according to coverage rules. Our work has allowed for completely inexperienced users to know what their tests are covering.

In the future probably we should focus our attention on allowing the user to model their system in any way they like, not only BPMN. Besides, the next expected version of SOA-TA could easily use more than the two levels of coverage now available, few changes will be necessary. In a mid-range future we could, with just a bit of effort, even transpose the SOA-TA concept to software testing in general, not only web services.

We have studied what other professionals have done and we are confident that the future of SOA testing will have a big part of automation in it. We also hope our work will serve academic research and commercial world interests. As Isaac Asimov once said,

"*If knowledge can create problems, it is not through ignorance that we can solve them*".

Bibliography

[1] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE software*, vol. 1, no. 1, p. 75, 1984.

[2] T. Erl, *Soa: principles of service design*. Prentice Hall Upper Saddle River, 2008, vol. 1.

[3] S. Oasis, "Reference model tc," *OASIS Reference Model for Service Oriented Architecture*, vol. 1, 2005.

[4] P. Offermann, M. Hoffmann, and U. Bub, "Benefits of soa: Evaluation of an implemented scenario against alternative architectures," in *Enterprise Distributed Object Computing Conference Workshops, 2009. EDOCW 2009. 13th*. IEEE, 2009, pp. 352–359.

[5] R. Heffner, C. Schwaber, J. Browne, T. Sheedy, G. Leganza, and J. Stone, "Planned soa usage grows faster than actual soa usage," *Forrester Research*, vol. 28, 2007.

[6] M. Hedin and IDC, *Worldwide SOA-Based Services 2007-2011 Forecast and Analysis: A Maturing SOA Market Fuels New and Different Demands for Professional Services*, Apr. 2007.

[7] S. Mulik, S. Ajgaonkar, and K. Sharma, "Where do you want to go in your soa adoption journey?" *IT Professional*, vol. 10, no. 3, pp. 36–39, 2008.

[8] L. Ribarov, I. Manova, and S. Ilieva, "Testing in a service-oriented world," 2007.

[9] G. Canfora and M. Di Penta, "Soa: Testing and self-checking," in *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, 2006, p. 3.

[10] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening soa testing," in *Proceedings of the the 7th joint meeting of the European software engineering confer-*

ence and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2009, pp. 161–170.

[11] F. A. Authority, "Software considerations in airborne systems and equipment certification, 1992. document no," RTCA/DO-178B, RTCA, Inc, Tech. Rep.

[12] S. Rayadurgam and M. P. Heimdahl, "Coverage based test-case generation using model checkers," in *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the.* IEEE, 2001, pp. 83–91.

[13] S. A. White, "Introduction to bpmn," *IBM Cooperation*, vol. 2, no. 0, p. 0, 2004.

[14] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking.* MIT press, 1999.

[15] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "Wsdl-based automatic test case generation for web services testing," in *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop.* IEEE, 2005, pp. 207–212.

[16] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland *et al.*, "Web services business process execution language version 2.0," *OASIS standard*, vol. 11, no. 120, p. 5, 2007.

[17] L. Reitman, J. Ward, and J. Wilber, "Service oriented architecture (soa) and specialized messaging patterns," *A technical White Paper published by Adobe Corporation USA*, 2007.

[18] P. Kalamegam and Z. Godandapani, "A survey on testing soa built using web services," *International Journal of Software Engineering and Its Applications*, vol. 6, no. 4, 2012.

[19] B. Beizer, *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons, Inc., 1995.

[20] T. Ostrand, "White-box testing," *Encyclopedia of Software Engineering*, 2002.

[21] W.-T. Tsai, R. Paul, W. Song, and Z. Cao, "Coyote: An xml-based framework for web services testing," in *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on.* IEEE, 2002, pp. 173–174.

[22] Hp uft data sheet. [Online]. Available: http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=4AA4-8360ENW

[23] Parasoft soatest data sheet. [Online]. Available: https://www.parasoft.com/product/soatest/

[24] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites.* Packt Publishing Ltd, 2008.

[25] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Software Engineering—ESEC/FSE'99.* Springer, 1999, pp. 146–162.

[26] R. Black, *Advanced Software Testing-Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst.* Rocky Nook, Inc., 2014.

[27] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.

[28] T. L. Booth, *Sequential machines and automata theory.* Wiley New York, 1967, vol. 3.

[29] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," DTIC Document, Tech. Rep., 1976.

[30] H. Imai and T. Asano, "Efficient algorithms for geometric graph search problems," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 478–494, 1986.

[31] E. R. Gansner, S. C. North, and K.-P. Vo, "Dag—a program that draws directed graphs," *Software: Practice and Experience*, vol. 18, no. 11, pp. 1047–1062, 1988.

[32] T. Takagi, N. Oyaizu, and Z. Furukawa, "Concurrent n-switch coverage criterion for generating test cases from place/transition nets," in *Computer and Information Science (ICIS), 2010 IEEE/ACIS 9th International Conference on.* IEEE, 2010, pp. 782–787.

[33] D. Schumm, D. Karastoyanova, F. Leymann, and J. Nitzsche, "On visualizing and modelling bpel with bpmn," in *Grid and Pervasive Computing Conference, 2009. GPC'09. Workshops at the.* IEEE, 2009, pp. 80–87.

[34] J. C. Recker, "Bpmn modeling–who, where, how and why," *BPTrends*, vol. 5, no. 3, pp. 1–8, 2008.

[35] M. Owen and J. Raj, "Bpmn and business process management," *Introduction to the New Business Process Modeling Standard*, 2003.

[36] P. E. P. Tibco and I. P. D. U. Guide, "Tibco software inc," 2001.

[37] H. A. Eiselt, M. Gendreau, and G. Laporte, "Arc routing problems, part i: The chinese postman problem," *Operations Research*, vol. 43, no. 2, pp. 231–242, 1995.

[38] J. C. C. Costa, "Coverage-directed observability-based validation method for embedded software," Ph.D. dissertation, INSTITUTO SUPERIOR TÉCNICO, 2010.

[39] R. E. Tarjan, "Applications of path compression on balanced trees," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 690–715, 1979.

[40] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[41] L. De Moura and N. Bjorner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[42] Jmeter api. [Online]. Available: http://jmeter.apache.org/api/

[43] G. Mulligan and D. Gračanin, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," in *Simulation Conference (WSC), Proceedings of the 2009 Winter*. IEEE, 2009, pp. 1423–1432.

[44] K. Wagh and R. Thool, "A comparative study of soap vs rest web services provisioning techniques for mobile host," *Journal of Information Engineering and Applications*, vol. 2, no. 5, pp. 12–16, 2012.

[45] J. de Halleux, "Quickgraph: A 100% c# graph library with graphviz support," 2007.

Attachments

Order	Name	Order	Name
1	performEmployeeTicketing	49	demandEmployeeTickets
2	performEmployeeTicketing	50	demandTicket
3	performEmployeeTicketing	51	signOutAmadeus
4	createBooking	52	demandEmployeeTickets
5	signInAmadeus	53	demandTicket
6	retrievePNR	54	signInAmadeus
7	displayTST	55	addMultiElementsPNR
8	addMultiElementsPNR	56	addMultiElementsPNR
9	addMultiElementsPNR	57	createManualTSTTicket
10	retrievePNR	58	retrievePNR
11	displayTST	59	commandCryptic
12	commandCryptic	60	createManualTSTTicket
13	addMultiElementsPNR	61	modifyTSTTicket
14	addMultiElementsPNR	62	displayTST
15	retrievePNR	63	commandCryptic
16	confirmEmployeeIssuedTickets	64	createManualTSTTicket
17	addMultiElementsPNR	65	retrievePNR
18	addMultiElementsPNR	66	demandTicket
19	addMultiElementsPNR	67	addMultiElementsPNR
20	signOutAmadeus	68	addMultiElementsPNR
21	retrievePNR	69	commandCryptic
22	addMultiElementsPNR	70	commandCryptic
23	addMultiElementsPNR	71	addMultiElementsPNR
24	modifyTSTTicket	72	modifyTSTTicket
25	addMultiElementsPNR	73	commandCryptic
26	addMultiElementsPNR	74	addMultiElementsPNR
27	retrievePNR	75	retrievePNR
28	commandCryptic	76	addMultiElementsPNR
29	addMultiElementsPNR	77	modifyTSTTicket
30	signOutAmadeus	78	commandCryptic
31	retrievePNR	79	createManualTSTTicket
32	displayTST	80	retrievePNR
33	createManualTSTTicket	81	demandTicket
34	retrievePNR	82	addMultiElementsPNR
35	displayTST	83	signOutAmadeus
36	signInAmadeus	84	demandEmployeeTickets
37	retrievePNR	85	demandTicket
38	demandTicket	86	createManualTSTTicket
39	signInAmadeus	87	retrievePNR
40	retrievePNR	88	demandTicket
41	addMultiElementsPNR	89	signOutAmadeus
42	signOutAmadeus	90	modifyTSTTicket
43	demandEmployeeTickets	91	addMultiElementsPNR
44	demandTicket	92	modifyTSTTicket
45	signInAmadeus	93	displayTST
46	modifyTSTTicket	94	commandCryptic
47	addMultiElementsPNR	95	signInAmadeus
48	signOutAmadeus	96	addMultiElementsPNR

Table 8.1: Attatchment A -Tap Staff Booking Chow-1 Path