Test Accelerator for Service Oriented Architectures
(SOA-Test Accelerator)

Instituto Superior Técnico – Universidade de Lisboa

João Filipe Garrett Paixão Florêncio
joao.florence@tecnico.ulisboa.pt

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 an 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.

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, commonly known as V&V [1], save time and money. Service-Oriented Architectures (SOA) [2], as implemented by web services, present features that add much complexity to the testing burden. SOA, by its dynamic and always-on nature makes most testing techniques not directly applicable to it [8]. 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. We are going to focus on the integration of several individual software modules, in our case web services to be combined and tested as a group. In order for we 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. 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 dramatic time and cost savings would be realized [12]. This is what we are aiming to do.

1.1 Goals 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. 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.

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.

2 Related Work

Tying the work we have done with previous research is not an easy job due to the fact that we are joining two different study areas, automatic test case generation and service architectures testing platforms. 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

Rayadurgam [12] worked on a method of automatically generate test-cases with model checkers, which 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 ask 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.

The approach chosen by Bai [15] relies on a 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 use in a given process. Unless the whole system at hand is a unique service, analysing its operation set does not suit integration testing.

2.1.1 Testing SOA

Ribarov et al [8] address testing in three major functions, unit, integration and functional system testing. In unit testing the authors propose the reutilization of work already done in components testing and 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 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] In Bartolini [10] is presented 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 ignoring internal structure, as our SOA-TA does.

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.

This method of addressing SOA testing shows good results, nevertheless the need to modify services code to allow its instrumentation, cannot be disregarded.

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 section, it claims to automate testing. This is, at some extent true since it uses a scripting language to make repetitive tasks automatic. However it only automates independent testing of services, not the whole architecture testing process.
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.

It materializes an idea very similar to the one shown by Rayadurgam [12] aforementioned, testing services on the interface level.

2.2.3 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. 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’s made its success. It does not support WSDL parsing, 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. 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.

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

One of the distinguishing features of SOA Test Accelerator is finding and certifying the minimum number of test scenarios and automatic providing with proper input. 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 section. 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. In the next section, we are going to clarify some testing concepts and standards.

3 Background

In this section, 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.

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 and Diagram

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 automata (no guards or actions), Mealy machines (outputs depend on current state and in-
puts), Moore machines (outputs depend on current state only), state charts (hierarchical states) etc. We have concluded 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.

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 $V \times V$

### 3.1.2 Switch Coverage

(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. SOA-TA supports 0 − switch coverage (or Chow − 0) and 1 − switch coverage (or Chow − 1) test generation.

**“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. 1, all transitions (edges) of the graph would be tested once.

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. 1, there are 6 transitions of length 2. To achieve Chow − 1 we would have to test 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:

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

### 4 Solution Description

In this section, we are going to provide a full description of our system. As mentioned before, sections following 4.1 describe consecutive running stages.

**Figure 2: General System Description**

**4.1 Modelling the process**

We decided to use BPMN to do the modelling process with the help of Bizagi Modeler, a freeware tool.

Figure 3 shows an example of a possible business process involving five different services, each with
4.2 Graph Generation

one or more operations executed. As Fig. 3 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 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 4.4, how decisive this edge notes will be. Once the process is created, we export the model to BPMN file, a XML-like file with all the data required to rebuild the model outside the tool.

4.1.1 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. 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.

4.2 Graph Generation

The graph generation is the following stage of SOATA. Using a XML parser, we will store the nodes and edges information from the BPMN file in a way suited to preform graph searches. Upon graph generation some alterations are made, two dummy nodes are added, the process start and end, and an edge connecting them. The necessity for such modifications which will explained in the following sections.

4.3 Test Path Generation

The goal of the stage of the process is to produce a test case for each business process scenario. 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.

Solving this problem within our graph we would have a path covering all transitions, all service operation calls. To solve this problem, we use the strategy documented on Costa [38]. For this strategy to work, based on finding Eulerian circuits, it requires that the graph must be strongly connected, i.e. all nodes of the graph must accessible from any one of them.

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

```csharp
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
```

**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
4.4 Input Data

the paths in which all pairs of consecutive edges are tested. Figures 4 and 5 depict an example of what this algorithm accomplishes.

![Figure 4: Example graph - 4 Before](image)

Figure 4: Example graph - 4 Before

![Figure 5: Example graph - 4 After](image)

Figure 5: Example graph - 4 After

4.4 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.

Our goal in this stage is to provide semi-automatically input values which will exercise the paths previously created. 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 4.1. 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. 6.

![Figure 6: Example graph - 5](image)

Figure 6: 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 within 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 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].

4.5 Test Script Generation

SOA – Test Accelerator generated scripts to run on Apache JMeter tool. 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.

4.6 Test Script Execution and Reading Results

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. There are several parameterers one can measure when executing tests on webservies, such as response data and headers success/failure boolean, timestamp, hostname, latency, number of threads, elapsed Time etc. SOA-TA is able to detail on the script which of this items to save and now it is configured to save response data, success, timestamp and the elapsed time.

5 Implementation

In this section, 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. 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.7 shows the flow of communications in SOA-TA. Here, as in any other web ap-
5.1 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. To interact with it, users would have to use the operations it provided. In our case, this is where we implemented path generation, which as been extensively described in section 4.3, algorithms to route inspection problem, data discovery, the parsers, script generation and Script Execution.

5.2 Database and Data Access

All the SOA-TA produced data is persisted in a relational database in Microsoft SQL Server. 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.

6 Results

In this section we are going to describe the evaluation performed on SOA-TA. As it was already discussed in previous sections, its operation depends on several components worthy of an independent assessment. 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.

6.1 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 4.1.1 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. 8 shows a first example of a small, yet real-life process description. This graph was extracted from a Tibco log from the TAP - Transportes Aéreos Portugueses, a Wintrust client. It portrays 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, SOA-TA generates the two distinct paths. The first: `retrieveSbrFeed`, `insertSKFast`, `retrieveFrequentFlyerCluster`, `modifyBooking`, `signInAmadeus`, `retrievePNR`, `addMultiElementsPNR`, `signOutAmadeus`. With the second changing only the last step, ending in the eighth step, `addMultiElementsPNR`. In a small and simple example like this it is trivial to see that are two paths exactly are required for us to test all transitions of size 1, 0 − Switch coverage. Fig. 9 shows a much more complex and also real-life process description. This graph represents the operations performed 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 1.
Regarding the time consumption, we have registered that for SOA-TA to generate all 7 paths, the one shown on 1 and six more to achieve the 1 – switch level, it took about 14 milliseconds, 4 on the first path and 10 on the remaining.

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation Name</th>
<th>Step</th>
<th>Operation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EmployeeTicket.</td>
<td>31</td>
<td>modifyTicket</td>
</tr>
<tr>
<td>2</td>
<td>EmployeeTicket.</td>
<td>32</td>
<td>displayTST</td>
</tr>
<tr>
<td>3</td>
<td>createBooking</td>
<td>33</td>
<td>ManualTicket</td>
</tr>
<tr>
<td>4</td>
<td>signInAmadeus</td>
<td>34</td>
<td>retrievePNR</td>
</tr>
<tr>
<td>5</td>
<td>retrievePNR</td>
<td>35</td>
<td>displayTST</td>
</tr>
<tr>
<td>6</td>
<td>addMultiElem</td>
<td>36</td>
<td>Cryptic</td>
</tr>
<tr>
<td>7</td>
<td>Cryptic</td>
<td>37</td>
<td>addMultiElem</td>
</tr>
<tr>
<td>8</td>
<td>signInAmadeus</td>
<td>38</td>
<td>signOut</td>
</tr>
<tr>
<td>9</td>
<td>retrievePNR</td>
<td>39</td>
<td>demandTickets</td>
</tr>
<tr>
<td>10</td>
<td>demandTicket</td>
<td>40</td>
<td>demandTicket</td>
</tr>
<tr>
<td>11</td>
<td>signInAmadeus</td>
<td>41</td>
<td>addMultiElem</td>
</tr>
<tr>
<td>12</td>
<td>modifyTicket</td>
<td>42</td>
<td>ManualTicket</td>
</tr>
<tr>
<td>13</td>
<td>displayTST</td>
<td>43</td>
<td>retrievePNR</td>
</tr>
<tr>
<td>14</td>
<td>signInAmadeus</td>
<td>44</td>
<td>confMlssTickets</td>
</tr>
<tr>
<td>15</td>
<td>addMultiElem</td>
<td>45</td>
<td>addMultiElem</td>
</tr>
<tr>
<td>16</td>
<td>retrievePNR</td>
<td>46</td>
<td>Cryptic</td>
</tr>
<tr>
<td>17</td>
<td>demandTicket</td>
<td>47</td>
<td>modifyTicket</td>
</tr>
<tr>
<td>18</td>
<td>signOut</td>
<td>48</td>
<td>confMlssTickets</td>
</tr>
<tr>
<td>19</td>
<td>retrievePNR</td>
<td>49</td>
<td>ManualTicket</td>
</tr>
<tr>
<td>20</td>
<td>demandTicket</td>
<td>50</td>
<td>modifyTicket</td>
</tr>
<tr>
<td>21</td>
<td>ManualTicket</td>
<td>51</td>
<td>Cryptic</td>
</tr>
<tr>
<td>22</td>
<td>retrievePNR</td>
<td>52</td>
<td>signOut</td>
</tr>
<tr>
<td>23</td>
<td>displayTST</td>
<td>53</td>
<td>retrievePNR</td>
</tr>
<tr>
<td>24</td>
<td>addMultiElem</td>
<td>54</td>
<td>confMlssTickets</td>
</tr>
<tr>
<td>25</td>
<td>addMultiElem</td>
<td>55</td>
<td>Cryptic</td>
</tr>
<tr>
<td>26</td>
<td>retrievePNR</td>
<td>56</td>
<td>Cryptic</td>
</tr>
<tr>
<td>27</td>
<td>displayTST</td>
<td>57</td>
<td>ManualTicket</td>
</tr>
<tr>
<td>28</td>
<td>signOut</td>
<td>58</td>
<td>retrievePNR</td>
</tr>
<tr>
<td>29</td>
<td>modifyTicket</td>
<td>59</td>
<td>Cryptic</td>
</tr>
<tr>
<td>30</td>
<td>addMultiElem</td>
<td>60</td>
<td>Cryptic</td>
</tr>
</tbody>
</table>

Table 1: TAP Staff Booking Generated Paths - Chow-0

We have also tested SOA-TA with process descriptions from Bizagi. Figure 10 shows an example of a simple process, however tasks 2 4 and 5 are on purpose connected in a loop. The generated path for this example are shown on table 2

![Figure 10: Process Example 2](image)

Table 2: Process Example 2 - Generated Paths

6.2 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 Apple 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, conversionRate. 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.”

This fictional user story would produce a process description like the one on Fig. 11. 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 at www.webservicex.net under the names “stockquote”, CurrencyConverter, length, Weather and in w3schools.com the temp-
convert.

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 3 4. 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 4, 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).

<table>
<thead>
<tr>
<th>Step</th>
<th>Path 1</th>
<th>Path 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GetQuote</td>
<td>GetQuote</td>
</tr>
<tr>
<td>2</td>
<td>ConversionRate</td>
<td>ConversionRate</td>
</tr>
<tr>
<td>3</td>
<td>ChangeLengthUnit</td>
<td>GetWeatherByZIP</td>
</tr>
<tr>
<td>4</td>
<td>ChangeLengthUnit</td>
<td>ConvertTemp</td>
</tr>
<tr>
<td>5</td>
<td>GetWeatherByZIP</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ConvertTemp</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Example 5 - Chow-0 Test Paths

<table>
<thead>
<tr>
<th>Step</th>
<th>Path 3</th>
<th>Path 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GetQuote</td>
<td>GetQuote</td>
</tr>
<tr>
<td>2</td>
<td>ConversionRate</td>
<td>ConversionRate</td>
</tr>
<tr>
<td>3</td>
<td>ChangeLengthUnit</td>
<td>ChangeLengthUnit</td>
</tr>
<tr>
<td>4</td>
<td>ChangeLengthUnit</td>
<td>GetWeatherByZIP</td>
</tr>
<tr>
<td>5</td>
<td>ChangeLengthUnit</td>
<td>ConvertTemp</td>
</tr>
<tr>
<td>6</td>
<td>GetWeatherByZIP</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ConvertTemp</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: 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 4.4, we use a SMT solver to aid us in this matter. We are going to follow closely input data generation for Path1 from table 3 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, ConvertUsdToEur, ChangeLengthUnit, ChangeLengthUnit, GetCityWeatherByZIP, ConvertTemp. Adding the restrictions on the edges, we get something like: GetStockValue (GetStockValueResult>400k) ConvertUsdToEur (ConvertUsdToEurResult>400k) ChangeLengthUnit, ChangeLengthUnit, GetCityWeatherByZIP, 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.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 756</td>
<td>2098046</td>
<td>21 951</td>
<td>2455438</td>
</tr>
<tr>
<td>60002</td>
<td>6711898</td>
<td>62 146</td>
<td>6951651</td>
</tr>
<tr>
<td>28 715</td>
<td>3212059</td>
<td>43 518</td>
<td>4867923</td>
</tr>
<tr>
<td>8 684</td>
<td>971392</td>
<td>85 889</td>
<td>9607543</td>
</tr>
<tr>
<td>42284</td>
<td>4729928</td>
<td>61 124</td>
<td>6837330</td>
</tr>
<tr>
<td>1 242</td>
<td>138930</td>
<td>63561</td>
<td>7109999</td>
</tr>
<tr>
<td>69 836</td>
<td>7811854</td>
<td>85331</td>
<td>9545134</td>
</tr>
<tr>
<td>24 801</td>
<td>2774239</td>
<td>32 055</td>
<td>3585672</td>
</tr>
</tbody>
</table>

Table 5: GetStockValue - Unit test results (16 samples)

With the results from table 5 from the GetStockValue operation, we would take all the outputs and feed it to ConvertUsdToEur. Table 6 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.

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 5 it will produce an implication and a logic OR restriction like the following:

- (=> (= GetStockValueInput 18756) (= GetStockValueOutput 2098046,16))


\[ \text{GetCityWeatherByZIP} \text{Step with 10023, a New York ZIP code, as input, returned 88° Fahrenheit, which after ConvertTemp converted to 31.11° Celsius. For all other paths built from this task description a similar process would take place.} \]

\[ \text{SOA-TA will then use all the restrictions given and will check the formula for satisfiability. If the formula is satisfiable, it can produce a model which will have possible value for all the unknown variables we might have. In this case, it produced the following values: GetStockValueInput = 18756, GetStockValueOutput = 2098046,16, ConvertUsdToEurInput = GetStockValueOutput, ConvertUsdToEurOutput = 1846280,621}. \]

\[ (\text{or } (\text{= GetStockValueInput 18756}) \]

\[ (\text{other for the relation between GetStockValueOutput and ConvertUsdToEurInput, like the following:}) \]

\[ (\text{= ConverteStockValueOutput 2098046,16}) \]

\[ (\text{= ConvertUsdToEurInput 1846280,621}) \]

\[ (\text{or } (\text{= ConvertUsdToEurInput 2098046,16}) \]

\[ (\text{= ConvertUsdToEurInput GetStockValueOutput}) \]

\[ \text{SOA-TA then use all the restrictions given and will check the formula for satisfiability. If the formula is satisfiable, it can produce a model which will have possible value for all the unknown variables we might have. In this case, it produced the following values: GetStockValueInput = 18756, GetStockValueOutput = 2098046,16, ConvertUsdToEurInput = GetStockValueOutput, ConvertUsdToEurOutput = 1846280,621}. \]

\[ \text{In the end of this phase we have all the inputs necessary to create the script, from the ones manually set as the distance in the ChangeLengthUnit step or weather values in GetCityWeatherByZIP which are less relevant in this case, to the automatically generated and validated by the SOA.} \]

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

\[ \text{To execute the script, as it was already mentioned in section 4.6 we start a command line process and provide the file path to be run. JMeter then, executes the script, saves the general results in a text file and the partial results of each step in a answer file. This file contains the usual XML contained in one SOAP response message. SOA parses this data and stores results in the database. For the example we were testing JMeter produced the following results file (the values are respectively the time Stamp, elapsed time, responseCode, responseMessage and success boolean):} \]

\[ \begin{array}{llll}
1442508850,783,200,OK,true \\
1442509634,827,200,OK,true \\
1442510462,827,200,OK,true \\
1442511290,769,200,OK,true \\
1442512163,873,200,OK,true \\
1442512955,792,200,OK,true \\
\end{array} \]

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

7 Conclusions

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.

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”.

References


