An automated unit testing framework for the OutSystems Platform

Armando João da Silva Gomes

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisor: Prof. João Carlos Serrenho Dias Pereira

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. João Carlos Serrenho Dias Pereira
Member of the Committee: Prof. Paulo Jorge Fernandes Carreira

October 2018
Acknowledgments

This thesis is just another chapter in my life. Obviously, it’s a great chapter but it would not happen if I didn’t had conditions to be here. As such, I feel the need to thank a few people for all they have done so far:

• I would like to thank my parents for all the effort they have made to get me an academic degree. It’s because of them that, today, producing this work is a possibility.

• To my girlfriend Luísa, who has supported me so much, always incentivating me to go higher, to “just do it” and to never think there isn’t a way - there’s always a way!

• To my very good friends Fábio and Tiago, with whom I spent countless hours discussing about pretty much everything, from weekend’s football results to “find the missing comma in this huge query” on Slack, always being there for each other.

• My previous employer, Infosistema, for all the flexibility they have given me so I could attend the classes.

• From OutSystems, I would like to thank two specific people: António Barros and Vera Tiago. António’s experience and past stories were a huge contribution to the progress. To Vera, which I met while working for My Game Solutions, who gently nicknamed me of “Benjamin” when I enrolled into the MVP program and helped me countless times using the rubber duck debugging technique.

• I would like to thank each and every single person in the OutSystems MVP Slack channel, specially all the fellow MVPs that have contributed, in one way or another, to the development of this idea.

• I would also like to acknowledge my dissertation supervisor Prof. João Pereira for his insights, support and sharing of knowledge that has made this thesis possible.

To each and every one of you – Thank you.
Abstract

The rapid growth of the IT market needs and the increasing complexity of software projects led to a talent shortage. Companies in need of resources tend to hire either junior developers or people with no background in software development as well as choosing low-code platforms for development. The lack of skills - although softened due to the features of low-code platforms - lead to defective development, defective quality assurance process and, in the end, a defective software system. History has proven the financial and public impact of defective software systems and, as a result, companies will try to improve their quality assurance process. One of the methods to improve the quality assurance process is through automated software testing. The current offerings for the OutSystems platform lack the required functionality and/or require additional developments that represent a significant overhead to project planning - in terms of schedule and budget. In this work, a framework for automated test case generation in the OutSystems platform is proposed. This framework targets the Core Business layer in OutSystems proposed 4-Layer Canvas.

Keywords

OutSystems, software testing, automated testing, automated test generation, low-code
Resumo

O rápido crescimento de necessidades no mercado das tecnologias de informação aliado ao constante aumento de complexidade das aplicações conduziu à falta generalizada de talentos na área de IT. Por forma a colmatar estas necessidades, as empresas tendem a contratar programadores júniores ou então de outras áreas que não envolvam desenvolvimento aplicacional bem como a escolha de plataformas low-code. A falta de conhecimentos destes programadores, embora atenuada pelas funcionalidades destas plataformas, conduz a um processo de desenvolvimento deficiente, a um processo de controlo de qualidade deficiente e, no fim, a uma aplicação que não corresponde às expectativas. A história provou, no passado, quais os potenciais impactos em termos financeiros e de percepção pública e, por forma a evitar estes acontecimentos, as empresas tendem a investir no processo de controlo de qualidade. Um dos métodos utilizados para aumentar a qualidade do software produzido é com recurso a testes de software automatizados. As soluções existentes para a plataforma OutSystems não possuem as funcionalidades desejadas ou requerem desenvolvimentos adicionais que implicam um acréscimo de trabalho, impactando o calendário e o orçamento de projeto. Neste trabalho é posta uma ferramenta para geração automática de testes de software, ferramenta esta que foca-se na camada de Core Business, existente no padrão de arquitetura recomendado pela OutSystems - 4-Layer Canvas.

Palavras Chave

OutSystems, testes de software, testes automáticos, geração automática de testes, low-code
Contents

1 Introduction

1.1 Cost of building software ...................................................... 3
  1.1.1 Software Complexity and Skill Shortage ........................ 3
  1.1.2 Effort estimation, Scope Creep and Deadlines ................. 4
1.2 Low-code platforms .......................................................... 5
  1.2.1 What is a low-code platform? ...................................... 5
  1.2.2 Why so popular? ...................................................... 6
1.3 Motivation ............................................................................. 7
  1.3.1 The cost of software .................................................... 7
  1.3.2 How to assure software quality? ..................................... 7
  1.3.3 Human factors, skill shortage and automation ................. 8
  1.3.4 A faulty Quality Assurance (QA) process? ....................... 9
1.4 Problem ................................................................................. 9
1.5 Objectives ............................................................................. 9
  1.5.1 Why OutSystems? ....................................................... 10
    1.5.1.A The competition .................................................... 10
1.6 Organization of the Document ............................................. 11

2 OutSystems platform ............................................................ 13

2.1 Product overview .............................................................. 15
  2.1.1 Service Studio ............................................................ 17
2.2 Application Architecture .................................................... 19
  2.2.1 Architectural principles .............................................. 19
    2.2.1.A The layers .......................................................... 19
    2.2.1.B A typical architecture design .................................. 20
2.3 Core Layer ............................................................................ 20
2.4 OutSystems Modelling Language ....................................... 21
  2.4.1 Domain-Specific Languages ........................................ 21
3 Related work

3.1 Method scope testing

3.1.1 Control Flow Graph (CFG) and Coverage
3.1.2 Method scope test design patterns
  3.1.2.A Category-Partition
  3.1.2.B Recursive Function Test
  3.1.2.C Polymorphic Message Test
  3.1.2.D Combinational Function Test

3.2 Mocking

3.3 Automated Software Testing (AST) techniques

3.3.1 Static Analysis
3.3.2 Fuzzing
3.3.3 Random Testing
3.3.4 Symbolic Execution
  3.3.4.A Constraint Solving

3.4 Solutions for the OutSystems platform

3.4.1 Unit Testing Framework
3.4.2 BDD Framework
3.4.3 Test Framework
3.4.4 Test Automator
3.4.5 Previous academic research

3.5 Solutions for the .NET Framework

3.5.1 IntelliTest
3.5.2 Unit Test Boilerplate Generator

3.6 Test case generation in Domain Specific Languages

4 Solution

4.1 OutSystems Architecture

4.1.1 Assemblies generated during compilation
  4.1.1.A Changing the path

4.2 The OutSystems Modelling Language (OML) file in eXtensible Markup Language (XML)

format

4.2.1 Actions
4.2.2 Properties
4.2.3 Widgets and Connectors
List of Figures

2.1 OutSystems platform high-level architecture. Source: OutSystems.com. . . . . . . . . . . . 15
2.2 The various components of the platform server in the overall OutSystems platform archi-
tecture. Source: OutSystems.com. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3 A look into the Service Studio. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.4 The 4-Layer Canvas (4LC) is usually represented in a stacked format, with each layer
having its own color. Source: OutSystems.com. . . . . . . . . . . . . . . . . . . . . . . . . 21
2.5 A sample action in the OutSystems platform, which wraps the Fixer.io REST API for cur-
rency conversion. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.1 Control flow graph representation. Source: Guru99.com. . . . . . . . . . . . . . . . . . . . 27
4.1 A module viewpoint of the proposed architecture for the testing framework. . . . . . . . . 39
4.2 Service Studio . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3 Signature of the Parsing Engine action. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.4 OutSystems Parser project structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.5 OutTest dashboard . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.6 Importing an XML file into OutTest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.7 Test suite listing for a given XML file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.8 Test suite for Division action. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.9 Branches for Division action. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.10 Test executed and returned success. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.11 Test executed and returned error. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.12 Mocked value for a Session Variable. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.13 Mocked value for a Site Property. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.14 Dynamic Execution Engine signature . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.15 MockedParameters structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
List of Tables

3.1 Comparison of different OutSystem solutions ........................................... 34
Listings

4.1 Obtaining all user actions from a given DLL ............................................. 40
4.2 Retrieving method’s signature. ................................................................. 40
4.3 Mapping the output. ................................................................................. 40
4.4 XML root element. ................................................................................. 41
4.5 Action element structure and child nodes. .................................................. 42
4.6 Action element properties ...................................................................... 43
4.7 Aggregate XML element ......................................................................... 44
4.8 Generate XSD and C# class commands. ..................................................... 45
4.9 Deserializing the binary file. ................................................................. 46
4.10 Iterating the existing actions using foreach loops. .................................... 47
4.11 Parsing variables of a given action. ......................................................... 48
4.12 GetInputParameters method. ................................................................. 48
4.13 Parsing the existing nodes. ................................................................. 49
4.14 Parsing the OSNodeIF ........................................................................... 49
4.15 Computing the next node. ................................................................. 50
4.16 Computing the branches and respective conditions. ............................... 51
4.17 BuildNodeTree method body. ............................................................... 52
4.18 Mocking session variables and site properties. ......................................... 62
4.19 Filling the parameters before method invocation. .................................... 63
4.20 Parsing the output of action invocation. ................................................ 63
Acronyms

COTS  Commercial Off-The-Shelf
4LC   4-Layer Canvas
QA    Quality Assurance
UI    User Interface
XML   Extensible Markup Language
AST   Automated Software Testing
MST   Manual Software Testing
SOA   Service-Oriented Architecture
BPT   Business Process Technology
CRUD  Create, Read, Update and Delete
OML   OutSystems Modelling Language
DSL   Domain-Specific Language
IDE   Integrated Development Environment
XML   eXtensible Markup Language
CFG   Control Flow Graph
MUT   Method Under Test
SAT   boolean SATisfiability problem
SMT   Satisfiability Modulo Theory
IL    Intermediate Language
CIL   Common Intermediate Language
DLL   Dynamic-Link Library
XSD   XML Schema Definition
1

Introduction

Contents

1.1 Cost of building software ........................................ 3
1.2 Low-code platforms ............................................. 5
1.3 Motivation ....................................................... 7
1.4 Problem ........................................................ 9
1.5 Objectives ....................................................... 9
1.6 Organization of the Document ............................... 11
Building software is an expensive and demanding task. When companies cannot find exactly what they are looking for - usually, a Commercial Off-The-Shelf (COTS) solution -, they have no alternative than to build it from scratch. Whether companies tend to build it themselves or hire another company to do so (outsourcing), is a decision that might depend on the organizational maturity, application domain and in-house knowledge, flexibility, time constraints, value and overall productivity [1]. Additionally, when building in-house software, companies tend to fall into the sunk cost effect, in which decision-makers are heavily influenced by the investment already made [2]. In order to reduce costs and increase the productivity, companies tend to embrace low-code platforms and their bold promises of delivering software multiple times faster. This introductory chapter will identify some of the factors that influence the software cost and quality and how low-code platforms give response to current software development needs.

1.1 Cost of building software

So, why is the software development process so expensive? There are various reasons for that. The single biggest reason is the development team. According to Glassdoor, the average software engineer in Portugal costs around 25.000 to 30.000 EUR per year. If a company needs 10 people to develop a single project, just the development team costs can go over 250.000 or even 300.000 EUR, depending on the member profiles. Other factors include the complexity, the required time-to-market, the required expertise and so on. Finally, one additional factor that influence the cost is the failure rate. The Standish Group produces an yearly report with the state of the IT projects, with the first report published in 1994 [3]. This report already had very interesting mentions regarding IT application development: in the United States, the average number of projects per year was 175.000, leading to a total cost of over 250 billion USD. The impressive information comes next: of those 175.000 projects, around 31.1% of them were canceled before being concluded and 52.7% ended up costing more than 189% of their original estimates. In terms of successes, only 16.2% of the projects were being completed on-time and on-budget. The costs associated with building custom software don’t end once the software is built - applications usually require constant evolution and maintenance, which also helps in increasing overall costs. But which factors influence the software cost?

1.1.1 Software Complexity and Skill Shortage

Determining the complexity in a software system may be considered a subjective term. A study by McNicholl and Magel in 1982 [4] has shown that the perception of the complexity of software varies from individual to individual, that it may be influenced by the size of the result than the expended effort and, finally, that doesn’t seem to be any measurable individual characteristic that could allow the prediction of previous statements. In an article by Robert Glass titled “Sorting out software complexity” [5], the
The author has conducted a research and has reached two interesting findings: for each 25% increase in the problem complexity, there is a 100% increase in the complexity of the software solution and that explicit requirements explode by a factor of 50 or more into implicit (design) requirements as a software solution proceeds. With the added complexity to the software, it’s crucial that project managers are able to find the required talent with the proper skills. Unfortunately for said managers, there is not enough talent available - thus, a skill shortage. In other words, a skill shortage can be defined as a set of circumstances where it is difficult for an entity to identify and/or employ a professional with the required skills. The most recent study from ManpowerGroup, 2018 Talent Shortage Survey [6], has identified a 12-year high for the global talent shortages, with a considerable value of 45%. In Portugal, is referred that the talent shortage is at 46%, with IT being classified in 7th place in the most in-demand roles. Still related to Portugal, it is referred that the main reasons for the talent shortage are the lack of applicants (24%), the lack of experience (13%) and the lack of required hard skills (25%). Although the main reasons are pointed at candidates, a study conducted in 2018 also concludes that companies are also liable for these skill shortages [7]. Finally, it’s widely perceived that the testing job profile is less-preferred than a software-development profile, with test engineers being considered as a “second class citizens” [8].

1.1.2 Effort estimation, Scope Creep and Deadlines

Side-by-side with software complexity, the effort estimation plays a critical role in defining the success of a software system implementation. Estimation in agile development has been based on expert’s opinion, historic data [9], use case points and planning poker [10]. But what is, exactly, software estimation? Vyas defines [10] software estimation as the methodology for anticipating the practical measures which are effort and cost measured in individual hours and capital in the context of software. On a more “informal” definition, Steve McConnell defines estimate differently: it’s either a commitment or a plan to reach the target [11]. Although several estimation techniques exist, they often fail because they don’t consider all of the software engineering economics [12], with the most noticeable being the skill of the person executing the task. Also adding to the overall cost and complexity is the scope creep. The scope creep can be defined as the gradual increase of the project scope without a formal acceptance and a proper acknowledgement of associated costs, impacts in the schedule or other possible effects [13]. One of the most common suggestions to avoid the scope creep is to define from the start what is and is not within the scope, with every other additional request to be negotiated afterwards [14]. While some studies [15] [16] have shown that the scope creep is one of the primary causes for project failure, a study on the impact in terms of project quality [17] has shown that if the volatility of requirements is kept to a minimum, the

---

1 Explicit requirement could be considered a feature while an implicit requirement could be considered the “behind the scenes” tasks required to support said feature.
Quality Assurance (QA) and rework effort will be kept to a minimum as well, with the completion date to be slightly affected - if at all. On the contrary, if a uniform volatility is kept during the project course, QA effort, QA effectiveness and the rework rate are heavily affected 19%, 21% and 33% respectively, with the completion date being affected in 6%. On the other hand, deadlines tend to affect the quality of the QA process and, as a side effect, the quality of the produced software. A study conducted by Shah, et al. [18] has shown that quickly-approaching deadlines - or deadlines\(^2\) - tend to increase the pressure on the tester. The challenges for high-quality testing under pressure were identified as, mostly, lack of information and lack of proper communication. Some examples were that the required information to properly conduct the test was not available, testing teams did not know what was being fixed, issues only being communicated to the client closer to the deadline - and an additional effort from the team would be required - and so on. Additionally, the lack of acknowledgement of pressure situations, in which management neglects that the test engineers are working under heavy-pressure, was also referred.

1.2 Low-code platforms

The low-code market is on the rise and its worth is estimated to grow from 3.2 billion USD in 2016 to approximately 27.23 billion USD in 2022, a compound annual growth rate\(^3\) of almost 45%. In terms of valuation, the latest round of funding for OutSystems has introduced the company into the "unicorn list" - a list containing companies valued at more than one billion USD.

1.2.1 What is a low-code platform?

The term low-code is a new term for a concept that has been around for long. The notion of "power user" or "citizen developer", people who can optimize their workflow by themselves, inside companies is wildly known, with these platforms aiming at reducing the entry barrier for the average employee [19]. Usually, platforms that focus only on the average employee are called no-code platforms, in which no coding is required and all the available features are already present in it. On the other hand, the pure low-code platforms target traditional developers and IT people. A low-code platform allows a developer to build an application - usually web and/or mobile - faster due to the nature of the already available components. Additionally, if any specific requirement is needed, these platforms allow the developer to extend the platform with the given functionality - hence the difference between low-code and no-code.

\(^2\) Slang term for a deadline that an individual dreads

\(^3\) The compound annual growth rate (CAGR) is the mean annual growth rate of an investment over a specified period of time longer than one year.
1.2.2 Why so popular?

According to Forrester, a research and advisory firm, there are three key factors that highlight the real value of these platforms [20]:

• **Speed up application and innovation delivery** Based on Forrester July 2017 survey, 31 of 41 Application Development and Delivery (AD&D) leaders stated that traditional coding meant huge difficulties to meet business requirements on time while 21 of 41 also stated that the lack of flexibility was a challenge and, finally, 20 of 41 AD&D leaders stated that traditional coding means a long time to update existing applications. Related to the delivery speed on the low-code platforms, 40 out of 41 AD&D leaders stated that they have seen a notable or significant improvement while only one (1) AD&D leader stated that only a marginal improvement was seen. This data shows that low-code platforms allow teams to dramatically raise their productivity and ability to response in time to demands for business software.

• **Prove useful for large-scale applications** One common misconception about the low-code platforms is that are only good for departamental (code for small, non-critical) applications. Many of the adopters of these platforms have built and use these applications enterprisewide or in multiple departments.

• **Contribute to Application Development and Delivery move to public clouds** According to recent studies [21], the adoption of public cloud services - either from full applications (i.e. Atlassian’s JIRA) to infrastructure services (i.e. Amazon Web Services) - has been and is expected to keep growing, hitting a 23.6 billion USD valuation in 2020. Since these low-code platforms have publicly accessible clouds for development and delivery, they have emerged as a one of three major Platform-as-a-Service options.

Still related to speeding up the application delivery, low-code automation allows users to create and deploy high-quality web and mobile applications significantly faster – typically in weeks rather than months. According to OutSystems, gains in productivity can achieve up to 600% - or six times faster.\(^4\) Enabling IT departments to meet the growing demand for the rollout of new applications is only one of the benefits of low-code platforms since these platforms also streamline application maintenance and updates. This is critical, given today’s rapidly changing business needs. If the low-code platform is robust, it can also integrate readily with existing IT infrastructure meaning that migration and legacy enterprise systems are much less of a concern. In addition, users of these platforms can manage integration for easy access and replication, resulting in ongoing efficiency. An article written by Jon Idle [22] explains the rational of when and why use low-code platforms. According to the article, low-code platforms need to

be considered when there’s the need to build or manage multiple applications for a business in a quick manner, with proper lifecycle management and/or for multiple platforms. Besides the tangible results of developing in these platforms, there is also a greater stakeholder engagement and satisfaction - since they can iterate and see progress more frequently -, a lower risk and higher return on investment as well as, to certain extent, an elimination of the IT skills gap.

1.3 Motivation

According to the World Quality Report 2017-18 [23], in 2017 companies allocated 26% of their total IT budget to QA processes. This value, despite being considerable, means a five percent decrease (31%) from 2016 and a nine percent decrease (35%) from 2015.

1.3.1 The cost of software

Building software has an immediate and a future cost. While the former depends mainly on the duration, scope and resources, the latter depends not only on the same but it also depends on the effectiveness of the QA process. The cost of software testing has a visible face: people will often take the biggest share of the budget but sometimes there’s the need to license the bug tracking platform - like JIRA, for instance - or even some tools for testing - for instance, UI testing, load testing and others. But what about the “hidden costs” of the software testing? An article published in 2002 [24] concluded that bugs or errors had an annual cost of 59.5 billion USD. Another interesting fact is that almost 38% of that cost (22.5 billion USD) could be likely removed if the testing infrastructures would be improved. These costs represent if the errors would be found near to development stages - the sooner, the better - but, unfortunately, most of the errors were found late in the process. Related to when is an error found and the relative cost, Steve McConnel has detailed the subject in his book Code Complete [25]. It is stated that an issue introduced in the requirements and design phase could cost from 1 to 100 orders of magnitude\(^5\) while an error introduced in the development phase can cost from 1 to 25 orders of magnitude. It is safe to conclude that QA process plays a vital role in terms of a project or a company’s budget since it can save much more than it costs.

1.3.2 How to assure software quality?

Software quality can be a very broaden term. As per the IEEE Standard for Software Quality Assurance Processes [26], the process for quality assurance relies on a few tasks that need to be performed:

---

\(^5\)An order of magnitude is an approximate measure of the size of a number, equal to the logarithm (base 10) rounded to a whole number. In this case, if an error costs 1k dollars to fix in an earlier stage, it can cost up to 100k dollars in the later stages.
1. Identify criteria for product acceptance. The conditions may be derived from the contract, plan, documentation, quality assurance reports and other sources.

2. Determine whether the product conforms to the contract using techniques that include: reviewing, auditing, testing or evaluating the results of the previous.

3. Determine whether the product conforms to the documented acceptance requirements

4. Determine whether the acquirer has the means to determine the criteria to accept the product

From a development perspective, code review and software testing are the most used techniques and the ones that assure best results. Unfortunately, due to resourcing constraints - lack of resources with the proper skill set -, code review is neither executed or the people executing it doesn’t have the required skills to perform a critical analysis of others work. As such, software testing plays a (even more) fundamental role in the quality assurance process so a company doesn’t get affected - either financially or in its public perception. A good example of the financial impact was the failure in NASA’s Mars Climate Orbiter, in which a miscalculation caused by the usage of the wrong units lead to a 655 million USD loss. In respective of the non-direct financial impact, Jason Tee wrote an article stating several other impacts that a faulty software can cause: **loss of usage** - if the application is not usable, it won’t drive revenue to the business; **public perception** - depending on the perceived loss of use or revenue for users, the compensation may vary (if there is a financial loss) or bad reviews might stack up; **losses to competitors** - if the product is not working, users will start searching for alternatives in the market to satisfy their needs. These theories have been explored further with similar conclusions [27] [28].

### 1.3.3 Human factors, skill shortage and automation

Testing software can be a repetitive task that could be likely automated. The problem lies in the fact that automating those repetitive tasks can be more time-consuming than the execution of the task itself. In order to understand the human impact in the effectiveness of the QA process, some studies have identify if the motivation of the software tester had impact in the effectiveness of the QA process [29] [30]. While the same happens for software developers, a study shown that same factors affect software developers and software testers differently [31]. Considering this along with the current skill shortage, is automated testing the solution? Certainly not, at least for a short-term solution. It’s proven that automated testing can reduce some of the shortage, but it has to be seen as a long term goal although there is the need to make the testing process more efficient, predictable and effortless [32]. There are some myths surrounding the Automated Software Testing (AST) process: that it always results in improved software

---


quality; that every project or team can use automated tools and that AST is a “all or nothing” - either you do it only or not. Along with the myths, there are some facts like automated testing requires an initial investment, specialized training and that it doesn’t replace manual testing in its entirety [33]. A study from 2012, comparing AST to Manual Software Testing (MST) seems to validate these myths and facts [34].

1.3.4 A faulty QA process?

A faulty QA process can lead to higher costs in the end, making the “apparent expensive” costs of the QA process a tiny drop in the ocean. But does an extensive QA process leads to a better software? Bugayenko [27] thinks not and states that the hardest decision is to know when to stop. He defends that it’s better to launch a product that one has confidence in it than wasting time - and money - in trying to create a perfect product. A similar opinion is presented in a research note named “Impact of Patching on Software Quality” [35], although this is heavily influenced by the company position in the current market.

1.4 Problem

Due to the nature of low-code platforms, project managers look into their usage so they can reduce costs in two different vectors: development time is cut and, due to a lower learning curve, less skilled developers can be hired. Although it is expected that applications built with low-code platforms contain less bugs, they are not bug-free and, as a result, it is required to have a proper QA process in place. Despite providing huge benefits in terms of application development per se, they lack “low-code” testing capabilities: developers still need to manually implement their own unit tests. How can be assured that, due to the lack of proper QA skills of developers, the designed test suite is appropriate and the software is released with the desired quality?

1.5 Objectives

Considering the current talent shortage along with developers without the required skill set in order to assure the effectiveness of the QA process, the objective of this work is to produce a framework able to automatically generate and execute test suites for a given application built with the OutSystems platform. The main requirements for this framework are:

- Allow the developer to import and select the module under test.
- Generate full test suites for all the methods/actions present in a given module.
• Allow the developer to run the mentioned test suites, either on demand or in a given schedule.

• Provide execution reports for scheduled tests.

Additionally, it is important that this framework has a set of additional requirements in terms of performance, usability and size. These requirements are:

• The framework should provide an User Interface (UI) that is easy to use and that allows an easy understanding of the current state of the system, following Nielsen’s Heuristics [36].

• The framework should provide reasonable performance in terms of test execution, meaning that the productivity flow of a developer should not be affected by a given test execution - either on demand or scheduled.

• The framework should provide real-time results for a given test case/suite execution.

• The framework should be as small as possible in terms of OutSystems licensing, aiming for a reasonable 50 Application Objects in size\textsuperscript{8}.

1.5.1 Why OutSystems?

There are several reasons for choosing the OutSystems platform, with the most relevant being the daily development tool of the author. Additionally, the OutSystems platform is a world-renowned portuguese company that, in October 2018, has over 245 partners and more than 167.000 community members, being present in 52 countries and 22 different industries\textsuperscript{9}. In the low-code platform ecosystem, OutSystems is a founder and also considered a leader with a strong platform and big ambitions. The platform has received multiple awards, including the top rated low-code platform by TrustRadius and four times the CODiE award by SIIA: three for the best mobile application development platform and one for best could platform as a service. There are some features that highlight the advantages that the OutSystems Platform provides and make it a serious contender when a low-code or no-code platform is considered: rapid productivity, multi-channel development, reduced costs and no lock-in approach. A more detailed presentation of the OutSystems Platform is done in Chapter 2.

1.5.1.A The competition

The competition in the low-code market has increased in the past few years. In the Forrester Wave\textsuperscript{TM} report [20], OutSystems scored 4.35 out of 5 leaving behind other giants like Mendix (4.16) - which was recently acquired by Siemens -, Salesforce (3.31) or ServiceNow (2.96). Despite version 11 being

\textsuperscript{8}Application Objects are elements of the application and are used as a measure for licensing purposes.

\textsuperscript{9}About OutSystems, https://www.outsystems.com/company/about/
recently released, the most notable addition (in version 10) to the platform is the ability to use low-code for mobile application development. Despite all the praise, the OutSystems platform is not the *de facto* choice when the subject is low-code implementation because the pricing structure makes it unattainable to some companies. Not only that, but a compilation of the best low-code platforms by PC Magazine [19] has not included the OutSystems platform in the list. Their reasoning was based on two profiles - the *Average Joe* business user\(^\text{10}\) profile and the developer profile - in which OutSystems only excels in one: the developer.

Despite several attempts, the work presented in this thesis was not supported by OutSystems, limiting the end result to some extent. There were, however, some resources provided in order to streamline some of the progress.

### 1.6 Organization of the Document

This thesis is organized as follows: Chapter 1 presents the introduction, the motivation and the problem at discussion. In Chapter 2 the OutSystems platform is presented. Chapter 3 presents the related work in the field of study while Chapter 4 presents the decisions and outcomes of the developments. Finally, in Chapter 5, it is presented the results of the evaluation process while in Chapter 6 an overview of current and future work is discussed.

\(^{10}\text{The *Average Joe* business user is someone belonging to the company with no special development training.}\)
2

OutSystems platform

Contents

2.1 Product overview ................................................. 15
2.2 Application Architecture ......................................... 19
2.3 Core Layer ......................................................... 20
2.4 OutSystems Modelling Language ................................. 21
The OutSystems platform is a low-code solution for the complete application development lifecycle. OutSystems achieves this by allowing developers and teams to respond to critical business requests for demanding market needs. A development team can design, develop, assure quality and analyse and manage an application, independently of the size of it. As such, the platform provides a front to end development solution while also allowing deployment and maintenance needs. In the end, the goal of the OutSystems platform is to provide reduced risks and costs associated to software development by implementing a continuous delivery approach.

2.1 Product overview

![OutSystems platform high-level architecture](Figure 2.1: OutSystems platform high-level architecture. Source: OutSystems.com.)

Internally, the OutSystems platform provides environments, tools and components that streamline software development, covering all lifecycle steps. In terms of the environments, the two most important are:

- **Visual development environment - Service Studio** Service Studio is the IDE used to create all parts of the application stack: database structure, application logic, user interfaces, business processes, external integrations and security policies.

- **Integration environment - Integration Studio** Integration Studio is an environment used for component creation that extend the platform, i.e., external databases, specific snippets of code and
third-party systems.

The developments done within these two environments will reside in a third environment, the platform server. The OutSystems platform server is the server component and the core of the platform. This server does the process of generation and optimization of code as well as the compilation and deployment of it. Besides this, it is also responsible for batch jobs as well as application logs.

![The various components of the platform server in the overall OutSystems platform architecture. Source: OutSystems.com.](image)

**Figure 2.2:** The various components of the platform server in the overall OutSystems platform architecture. Source: OutSystems.com.

When the development is finished and the application is ready for the next step of the lifecycle, there are two additional resources that aid in the team: **Service Center**, which is a management console for the operational aspects of the environment - i.e. connection strings, web service endpoints - and **LifeTime**, which is a console for centralized management for development, quality assurance and production environments.

In terms of the development task, the visual development environment (Service Studio) allows developers to drag and drop elements to create the UI as well as business logic and data structures. Adding to this ease of use, Service Studio has a full reference-checking and self-healing engine that prevents
errors in the applications when changes are made. Although there is a reasonable set of available elements that developers can use, if these elements are not sufficient to achieve the desired result, they can go on and extend the platform with their own code.

The continuous delivery approach is, in simple terms, achieved by the click of a button: 1-click publish. When this functionality is used, the development environment allows the developer to manually or automatically merge his changes into the published version, replacing the current code and making it available right away. When the merge is complete - if necessary -, various services are invoked:

- **Code Generator** The code generator service converts the OutSystems modelling language modeled in the development environment and generates the native .NET (or Java) code. The reason for this is that generated applications are optimized for performance and security by executing on top of standard environments.

- **Deployment Service** Once the code generator finishes its job, the deployment services deploy the generated code into a standard web application server. This standard server is, usually, either Microsoft Internet Information Services or RedHat's JBoss. This service, during the deployment procedure, ensures that the application is consistently installed across all servers.

- **Application Services** During the whole process of generating, deploying and executing the application, the application services provide and manage the execution of batch jobs and logging features for the platform/applications. These logging features could go from simple audit messages to errors or performance indicators.

Along with this process of generating and compiling the application, an application model is created and is stored in the same platform server, using a built-in version control system and a dependency analysis is made. In the event that another application might be affected by the changes being applied, the system notifies the team accordingly.

### 2.1.1 Service Studio

The Service Studio (which can be seen on 2.3 ) is, as referred previously, the visual development environment and is the most used development tool inside the OutSystems ecosystem. Developers using the OutSystems platform will spend between 90 and 100% of their time using this tool. Service Studio’s main functionality is split into four major blocks:

- **On the left side** there is the widget bar. This bar contains all the available widgets for use within the given context. On the image, the widgets shown are the ones available in the context of creating a web screen. In the context of creating an action, the widget list is different.
On the center there is the screen that is being built. Here lies a graphical representation of the end result, with this tool using a what you see is what you get methodology.

On the right side there are two functionalities. On the top right, there’s the tree structure which allows the user to either see the current structure of a given page (not shown) or allows the user to see defined/referenced actions (shown). Additionally, this block will show the existing entities, existing processes and so on. On the bottom right lies the property pane. This pane contains the properties applicable to a given object.

Additionally, on the top right corner of the screen there are four tabs: Processes, Interface, Logic and Data. Processes allow the definition of BPM! (BPM!)-like processes, using OutSystems own Business Process Technology (BPT) while Interface allows the definition of screens, blocks, images, themes and multilingual locales. The Data tab contains the entities - database tables -, structures, session variables, site properties - global variables - and external resources - JavaScript libraries, for instance.

The Logic tab is the central point of this work. In the Logic tab, the developer can define and/or import server - or user - actions, can consume or expose integrations (REST or SOAP), can define access roles and, finally, can specify custom exceptions.

Within the scope of the OutSystems platform, there are two types of actions: client and server
actions. The client actions are applicable to mobile development only while the server actions are applicable to both mobile and web development. Considering only the web development component, these actions are then split within two sub-groups: screen and user actions. The screen actions are actions tied to a particular screen functionality, like a button or a link. User actions - commonly referred to server actions -, are the others that are not tied to any other element but themselves - much like a regular method in object-oriented programming.

The goal of this work is to provide a framework in which it is possible to test these user actions without any additional development - since screen actions are tied to UI interactions and, as such, can be manually tested without additional developments.

2.2 Application Architecture

When developing a project with the OutSystems platform, there are a set of “field tested” suggestions that allow the application’s architecture to be considered a good one. These suggestions are not mandatory per se but all the auxiliary tools for architecture validation are based on them - which means that, by not following them, the developer is setting himself for very difficult times ahead.

2.2.1 Architectural principles

OutSystems uses a pattern called 4-Layer Canvas (4LC) which helps the architect to design a Service-Oriented Architecture (SOA) in a simple manner to attain simple goals: it suggests - and encourages - the abstraction of services and the correct isolation of distinct functional modules in order to promote module reusing.

2.2.1.A The layers

As the name of the architecture pattern suggests, there are four different layers, each with its own purpose. From bottom to top:

- The **Library Layer** contains business-agnostic services to extend the framework/functionality with highly reusable assets, UI Patterns, connectors to external platforms and systems and, additionally, integration of native code (C# in case of the web component, Objective-C/Swift and Java for the mobile part).

- The **Core Layer** aims to provide the isolation of business modules, exposing reusable entities, business rules and business widgets.
• The **End User Layer** provides user interfaces and processes (BPM processes or, in OutSystems terminology, BPT processes), reusing the modules from the Library and Core layers in order to implement the user stories.

• Finally, the **Orchestration Layer** contains processes, dashboard and home pages in order to provide a unique and unified user experience.

Additionally to these layers, there are a set of rules used to validate if the architecture meets the suggested pattern. These rules are the following:

• **No upward references** A module from a lower layer can’t reference a module from a higher layer. For instance, a module from the Library layer (lower) can’t reference a module from the End User Layer (higher). Failing to respect this rule can imply the existence of a circular reference, in which the modules won’t be up-to-date but also can imply that other modules will require additional information due to unnecessary references.

• **No side references among End User or Orchestration modules** Having proper isolation between modules allows different lifecycles for different sponsors/teams. Also, it's possible that a reference between these modules bring along a significant set of dependencies from lower layers.

• **No cycles among Cores ou Libraries** Cycles bring undesirable and unexpected impacts, leading to a code that it’s harder to maintain and to scale.

### 2.2.1.B A typical architecture design

OutSystems recommends the usage of the 4LC in two different stages of the architecture design. The first one is when there’s the need to identify concepts, with these being either business concepts, technical concepts or integration needs. The second one is during the module design phase, modules which implement the given concepts - identified in the first stage. This is an iterative process: identifying concepts lead to module definition which may lead to more concepts and so on.

### 2.3 Core Layer

The Core Layer, Core Business Layer or even Core Services layer is the most important layer in an OutSystems application architecture. The decisions made during the design and, most important, the coding that is done during the development phase, hugely influence not only the outcome of the project but other factors like scalability, reusability and performance.

A typical module in this layer will be composed of entities - which is an element that allows information to be persisted in the database - and server actions, which can do Create, Read, Update and Delete
(CRUD) operations or any other business logic, like calculating a currency exchange rate or a loan interest rate. In Figure 2.5 is depicted a server action focused on a specific business logic.

This Core Layer is the central part of this work. The testing framework that was produced aims to allow the unit testing of these server actions without requiring any further developments. As of now, in order to test these actions it needs to happen one of two things: "wait" until the user interface is implemented so the logic can be tested - typifying as integration testing instead of unit testing - or implement a test case using regular development practices. In Chapter 3, existing frameworks that take advantage of additional developments will be presented.

2.4 OutSystems Modelling Language

As seen on Figure 2.5, the OutSystems Modelling Language (OML) is composed by a set of widgets that allow the developer to define the action flow. The OML is a Domain-Specific Language.

2.4.1 Domain-Specific Languages

Martin Fowler defines a Domain-Specific Language (DSL) as "a computer programming language of limited expressiveness focused on a particular domain" [37]. Moreover, he identifies four key elements to this definition:

- **Computer programming language** A DSL is used by humans to define instructions, therefore being a language easily understandable for humans but executable by a computer.

- **Language nature** A DSL should have a sense of fluency where the meaning comes not only from
Figure 2.5: A sample action in the OutSystems platform, which wraps the Fixer.io REST API for currency conversion.

the individual elements but also from the way they are connected.

• **Limited expressiveness** A DSL should support only the minimum required of features to support its domain.
• **Domain focus** A DSL should be focused on a small domain, potentiating its worthfulness - this being as a consequence of the limited expressiveness.

Beyond the definition, he also defines three main categories in which DSL may fit: external DSL, internal DSL and language workbench.

The **external DSL** is defined as a separate language from the main language of the application it works with. It has its own language and it's parsed by a code on the host application. The **internal DSL** is a particular way of using a general-purpose language, in which only a subset of the language’s features are used and, finally, a **language workbench** is an Integrated Development Environment (IDE) for defining and building DSLs.

The **OML** is an **external** DSL since it uses a separate language from the application it works with (HTML/C# for the generated code) and uses a parser\(^1\) that generates the final code. This parser is also able to generate eXtensible Markup Language (XML) files - which are used in this work.

### 2.4.2 OML and Code Generation

The code generator referred in Section 2.1 is based on a proprietary tool which converts the OML into C# code (or Java, up until platform 10). Additionally, this tool also has the capability to convert OML files into XML ones. A more detailed overview and usage of this XML file is presented in Chapter 4.

---

\(^1\)The inner working of this parser is unknown since it's intelectual property of OutSystems and was not disclosed.
## Related work

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Method scope testing</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Mocking</td>
<td>30</td>
</tr>
<tr>
<td>3.3 AST techniques</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Solutions for the OutSystems platform</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Solutions for the .NET Framework</td>
<td>35</td>
</tr>
<tr>
<td>3.6 Test case generation in Domain Specific Languages</td>
<td>36</td>
</tr>
</tbody>
</table>
As of October 2018, there is no solution that allows the automated test suite generation for User Actions in the OutSystems platform. There are, however, other existing solutions aimed at providing testing capabilities in the platform. In this chapter, relevant work will be presented along with the theory behind method testing as discussed on Robert V. Binder’s book Testing Object-Oriented Systems [38].

3.1 Method scope testing

The method scope testing is a deeper level of the class scope testing since here the parts that interoperate are statements. In other words, a method scope testing focuses on testing the relationship amongs statements. When testing classes, the common procedure is to send messages to one method at a time. Due to the nature of the OutSystems platform, user actions could be also called methods - thus being this mapping the bridge between object-oriented testing and OutSystems testing. A module, on the other hand, could also be considered as a class, although the similarities could not be so straightforward. Binder states that although only one method at a time is tested, methods cannot exist apart from a class. Again, due the nature of the OutSystems platform, this is not entirely applicable. There is no need to have an implicit cooperation between methods, "living" each one on their own. This doesn’t mean, though, that user actions cannot rely on other user actions - which is very common, indeed.

3.1.1 Control Flow Graph (CFG) and Coverage

The Control Flow Graph is a representation of predicates - single or multiple conditions which evaluate to true or false - using graph notation. Each of the nodes represent a block of instructions while the edges represent a predicate. Computing predicates allow the developer to identify which values trigger the given branches and, thus, allowing to achieve acceptable levels coverage. Coverage models include

![Control Flow Graph](image)

Figure 3.1: Control flow graph representation. Source: Guru99.com.
Statement Coverage, Branch Coverage, Multiple-condition Coverage Object Coded Coverage, Basis-Path Coverage and Data Flow Coverage. For the scope of this work, it is important to understand Statement and Branch coverage models.

The statement coverage model, or C1, is calculated based on the number of statements of a given method that have been executed at least once. The branch coverage model, or C2, uses the number of branches that are executed, even if some statements in those branches are not executed. According to Binder, in large systems achievable coverage lies between 80 and 85%, with the remaining 15 to 20% being due to unreachable code.

3.1.2 Method scope test design patterns

There are several patterns to design method scope tests. Each one applies to a specific purpose implemented by the Method Under Test (MUT).

3.1.2.A Category-Partition

The category-partition test pattern might be used on any given method or function in any given scope, being appropriate for any method that implements one or more independent functions. The intent of this pattern is to design a test suite based on input/output analysis. This pattern assumes that faults are related to value combination of message parameters and instance variables. If faults are only visible through certain sequences, these pattern might not reveal them.

The strategy involves identifying all functions of the MUT followed by the identification of input and output parameters for each function, along with categories. Then, partition each category into choices considering existing constraints on said choices. Finally, the test cases are generated by enumerating all the possible combinations and the expected results are developed using an appropriate oracle. Although being a fairly straightforward technique, the identification of categories and choices is a subjective process and the size of the generated test suite may become quite large.

3.1.2.B Recursive Function Test

The recursive function test pattern is appropriate for methods that are recursive to implement a search, an evaluate, a self-referencing formula or any other similar algorithm. Despite the functionality of these methods has an equivalent loop, the procedure is different than testing loops. Low-coverage testing on recursive functions may hide a lot of bugs. Usually the fault model when the arguments are unusual, the stack space is insufficient or the base case is not reached/detected.

In terms of strategy, this model involves the definition of the base case, the recursive case, the preconditions for the initial call, the post-condition and, finally, all descent-phase calls and ascent-phase
bindings. The test suite should then try to violate the pre-condition in the initial call and at least once in the descent phase; attempt to violate the post-condition at least once in the ascent-phase and test boundary cases on depth. Additionally, it should attempt to force all exceptions in server or environment objects on which the method depends. In the end, a recursive function test pattern requires no more than two dozen test cases and reveals faults that result in incorrect method evaluation for a given message or state.

3.1.2.C Polymorphic Message Test

The polymorphic message test pattern is used to augment the test suite for a client of a polymorphic server. This is specially important when a polymorphic method can be binded to several implementations. The strategy for this pattern relies on identifying the number of candidate bindings for each message sent to the polymorphic object and then expand each segment with a multiway branch subgraph by adding two nodes: one for the branch node and other for the sequential node. Finally, add a catch-all node to represent binding errors. This pattern reveals bugs in client class usage but should not be used by itself only. Additionally, in order to be effective, the analysis of the class hierarchy is required.

3.1.2.D Combinational Function Test

The combinational function test pattern is more appropriate for methods that implement complex algorithms, business rules or other case-based logic or, in other words, it aims to design test suites for behaviors based on state or message parameters.

The fault model of this pattern could be when incorrect values are assigned to decision variables, incorrect or missing operators/variables in predicates, incorrect structure in a predicate, incorrect or missing default case or default action and many others.

The strategy for this pattern is to create a decision table that lists which different input combinations will result in different actions, along with the expected response of the MUT. Each of the entries in the input combinations should only be boolean-type expressions and each combination of input variables and respective action is called a variant. In the case that decision variables are not boolean, additional tests should be made to exercise the boundaries.

This pattern highlights incorrect response actions to test messages and often reveals design errors and omissions. Additionally, faults resulting from the order of message to external dependencies or corruption of object variables hidden by the MUT interface may not be shown.
3.2 Mocking

When the MUT has any external dependencies, it is useful to use mocking so the developer can only focus on the code being tested and don’t get concerned about the behavior of said dependencies. Mocking works by replacing dependencies with closely controlled objects that simulate the expected behavior. There are three types - or levels - of mocking:

- **Fakes** allow the developer to replace the actual code of the external dependency by implementing the same interface but without further interaction with other objects. The results from said implementation are hardcoded and the need for new fakes increase as the test suite increase. Big test suites mean a fake structure that is hard to understand and maintain.

- **Stubs** is an evolution from fakes. While using fakes require implementing actual code for the external dependency, using stubs with a mocking framework allow the developer to create a stub with minimal amount of effort.

- **Mocks** is the most evolved mocking type. It is very similar to stubs - providing the same advantages - but allow the developer to set expectations on a given method - developer can define the output value for each method call, for instance. While stubs are “blind” substitutes, mocks allow the developer to verify its usage.

In order to use these types of mocking, a developer uses a mocking framework. Mocking frameworks are not substitutes of unit testing frameworks but rather complement them by isolating dependencies. In terms of offerings, there are several free and commercial mocking frameworks. In terms of the free offering, JustMock Lite, NSubstitute, Moq and Rhino Mocks are the most publicized. Microsoft Fakes, included in Visual Studio is also a good option. In terms of commercial solutions, JustMock and Typemock seem to be the most complete offerings.

3.3 AST techniques

In the field of automated software testing, there are several techniques that can be applied. These techniques often vary in terms of cost (either time, budget or computational) or effectiveness. Simpler techniques - random testing and fuzzing - often produce less results but cost less, while symbolic execution may be the more complex of the presented techniques.

3.3.1 Static Analysis

The Static Analysis technique relies on analysing the code - either source or object code - without actually executing it. [39] It could be considered an automated way to do code review, for which the
results may vary from simple potential coding errors to a full model validation. It works by determining the runtime properties of the software automatically, without requiring any user input or code interaction. In order to achieve this, an abstraction representation of the application behavior is created and its states are analysed. [40] This technique varies from model checking technique since it doesn’t need to explore relevant states of the running software. [?] There is a huge offer in terms of Static Analysis tools, with an honorable mention of open-source Eclipse, Jetbrain’s IntelliJ IDEA as well as several extensions for Microsoft’s Visual Studio.

3.3.2 Fuzzing

Fuzzing or Fuzz Testing is a automated test technique where malformed, non-valid or random inputs are intentionally fed into the software in order to find flaws. [?] It is considered an effective technique to find security flaws in software although unexpected crashes and potential leaks in memory could also be found by it. There are some categories in which a fuzzer may fall in. It could be based on generation or based on mutation, whether the inputs are generated or mutated from existing ones; it could be a dumb or smart fuzzer, depending if it’s aware of the input structure and, finally, it could be white-box, grey-box or black-box, depending if it is aware of the program structure (white-box) or not (black-box). The list of available tools for fuzz testing is not that big, when compared to Static Analysis. According to OWASP¹, fuzzing tools are split into types, namely: mutational fuzzers, frameworks and domain-specific fuzzers. There are also commercial tools for fuzz testing.

3.3.3 Random Testing

Another software testing technique is random testing in which independent inputs are generated randomly - hence the name [?]. Once executed, the result of the execution is compared to specifications in order to assess if the test passed or failed. In the likely event that specifications do not exist, exceptions raised by the software under test are used to determine if the program runs successfully or not. Although it is not a very advanced technique, it’s quite useful to prevent testing bias. Besides that, it’s easy to use and costs close to nothing. On the downside, random testing has been proved to only find simple bugs and the results, when compared to other testing techniques, are much worse. One of the most well-known test tools that implement random testing is QuickCheck², developed originally in Haskell by Koen Claessen and John Hughes.

¹Fuzzing, https://www.owasp.org/index.php/Fuzzing
²QuickCheck, http://www.cse.chalmers.se/rjmh/QuickCheck/
3.3.4 Symbolic Execution

The symbolic execution testing method is a technique where a program is analyzed to determine which set of inputs trigger a given part of a program [41]. It is not a recent technique, with articles proposing it in 1975 [42] but has always faced difficulties related to constraint satisfiability. The computed paths are created using an interpreter that, by using symbolic values instead of actual values, obtains a set of expressions for the program input variables that will hit possible outcomes of conditional branches. There are some known limitations to this technique, though. For large programs, it might not be possible to execute all feasible paths [43] and, while interpreting a given application, it might not be possible to control environment interactions, causing consistency problems - KLEE [44] and Otter [45] are some tools used to mitigate issues related to environment interactions. Other challenges are memory handle, path explosion and constraint solving [46].

3.3.4.A Constraint Solving

In order to understand exactly which values of input data are required so a given branch can be excised, it is often used a constraint solver on the computed paths. A constraint solver is a decision procedure for logical formulas expressing problems. Two common solvers are the boolean SATisfiability problem (SAT) and the Satisfiability Modulo Theory (SMT) solvers. The SMT solver is an extension to the SAT in which instead of being based on boolean logic, it evolved into first-order logic [47]. An approach for automatically generate test data using constraining solving techniques was proposed by Gotlieb, et al. [48]. One example of a solver implemented for the .NET platform is the Z3

3.4 Solutions for the OutSystems platform

As stated previously, there are no solutions for the OutSystems platform for which no further developments are required in order to test user actions. In this section, the four available options are presented, along with a comparison table.

3.4.1 Unit Testing Framework

The Unit Testing Framework\(^1\) is a framework for unit test definition created by Andrew Burgess and published, for the first time, in June 3rd, 2013. The last update was published by Paulo Ramos in April 5th, 2016 and it has not been updated for the last installments of the Platform (v10 and v11). This framework provides a set of functions that the developer can use when designing the tests. As

---

\(^1\)https://github.com/Z3Prover/z3

\(^2\)https://www.outsystems.com/forge/Component_Details.aspx?ProjectId=387
suggested, this framework does not provide automated test case generation since it targets a "design-time" level instead of a "run-time" level. This can mimic what is done using a Test Driven Development [?] approach in other programming languages like Java or C#. The main difference is that it requires the signature of the function to be created beforehand, in order to be imported into the test case. The set of functions offered by this framework is a standard one, offering functions for the typical assert equals and assert fail/pass as well as others like assert is true/false, assert pass and, finally, assert contains text.

3.4.2 BDD Framework

The BDD Framework\(^5\) is a testing framework that provides a set of tools for easily creating Behavior Driven Development tests for an OutSystems application. The Behavior Driven Development is the base concept of the application, following the principles of the Given, When and Then. This framework allows the user to define the scenarios and then each step of the scenario. These steps require a specific development for each one, an activity that can be time consuming and might require additional development depending on the pre or post-conditions that are required. As such, depending on the type of the tests that the developer wants to do, designing and implementing the test might require additional developments to assure required data is present and the test can proceed. Similarly to the Unit Testing Framework, the BDD Framework also provides common assert functions, namely: assert, assert fail, assert true/false, assert value. Unfortunately, it has some limitations that difficult it's adoption: designing the tests can consume a lot of Application Objects; it doesn’t support scheduling of tests and it doesn’t provide an overall report based on test execution. It also doesn’t support automated test generation. An analysis on the recent information and blog posts by OutSystems on the software testing subject leads to conclude that testing using Behavior Driven Development is the suggested way for OutSystems applications.

3.4.3 Test Framework

The Test Framework\(^6\) is an open source application that offers simple management and automated execution of OutSystems Unit and API tests. With seamless integration with BDD Framework and Unit Testing Framework, it allows automated regression tests setup and execution. This component has been developed by Indigo and supports the following set of features:

- Manage Test Suites, Cases and Steps Define test scenarios, for both Unit and API tests, and it executes them manually or daily at a given schedule.

\(^5\)https://www.outsystems.com/forge/component/1201/bddframework/
\(^6\)https://www.outsystems.com/forge/2464/
• **Test execution classification** Classify test executions as Broken or Defect, to help teams focus on fixing tests, or actually fixing the wrong functionality identified by the test.

• **Introspection and execution of BDD and UTF Tests** Automatically imports and runs BDD or Unit Test Framework tests periodically, validating results on every run.

• **Quality overview** Monitor the evolution of defined test suites, gaining a clear understanding of whether tests are not designed for maintainability, or if the applications are increasingly having more quality and less regressions.

The documentation for this Test Framework is provided by OutSystems\(^7\) and it doesn’t seem to have huge adoption - only 131 downloads since the release, in September 27, 2017. Also, the Application Object consumption is of 83 AO’s, which is a significant amount for the OutSystems Platform - although AO’s are usually unlimited in development environments. On a usage-basis, this tool requires BDD Framework to manage and execute the tests, seeming like an extension to the said framework. This tool, like the others, doesn’t support automated test generation.

### 3.4.4 Test Automator

The Test Automator\(^8\) is a browser (based on Selenium) and unit (based on webservice) regression testing tool for the OutSystems Platform. It was built for OutSystems Platform version 7. The last update for this component was released in November 16th, 2015 by Paulo Garrudo and, in total, it counts 686 downloads since its initial release. This tool allows the usage of Selenium test scripts as well as some OutSystems logic though it’s most focused for UI testing instead of logical testing. This logical testing capabilities are well limited - it requires to be exposed on a web service and it has to follow a certain structure. Eligible logic methods should have no input parameters and should return only one output parameter, parameter that needs to be a single boolean or text.

<table>
<thead>
<tr>
<th>Feature</th>
<th>UTF</th>
<th>BDD Framework</th>
<th>Test Framework</th>
<th>Test Automator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Unit Tests</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Provides Scheduling</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Execution Reports</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AO Impact</td>
<td>High</td>
<td>High</td>
<td>Very High</td>
<td>Low</td>
</tr>
</tbody>
</table>

**Table 3.1:** Comparison of different OutSystem solutions


\(^8\)https://www.outsystems.com/forge/component/82/test-automator/
3.4.5 Previous academic research

This thesis is not the first attempt at improving testing capabilities in the OutSystems platform. A previous Master of Science degree thesis by Gustavo Guerra [49] proposed the creation of a new set of widgets to support unit tests. Although the work mentions that the proposal was considered very good which would later be included in one of the subsequent releases of the product, as of October 2018 it is still not included in past or present releases, even considering the brand new version 11.

Another thesis for the same Master programme by Ricardo Neto [50] has proposed a different approach in order to support automated test execution in the OutSystems platform using the Selenium WebDriver. Although it was also intended to be built a test framework for the OutSystems platform, this attempt was targeting HTML pages - the UI - instead of the Core Business Layer. Automated test generation was not considered due to the task complexity.

3.5 Solutions for the .NET Framework

There are some solutions targeted at the .NET Framework - which is the base of the OutSystems platform. The drawback is that these solutions require full access to the source code - which, as of now, is a feature only available for customers who stop using the platform.

3.5.1 IntelliTest

IntelliTest9 explores the .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional branch in the code. For example, if statements, assertions, and all operations that can throw exceptions are analyzed. This analysis is used to generate test data for a parameterized unit test for each of the methods, creating unit tests with high code coverage. When IntelliTest is executed, the developer can easily see which tests are failing and add any necessary code to fix them. He can select which of the generated tests to save into a test project to provide a regression suite. As the code is changed, IntelliTest shall be rerun to keep the generated tests in sync with code changes. Its limitations is that is only available for the Enterprise Edition of Visual Studio 2015 (and later); only supports C# code and do not support x64 configurations. It also has the ability to be extended and support emitting tests in MSTest, MTest v2, NUnit and xUnit format.

---

3.5.2 Unit Test Boilerplate Generator

The Unit Test Boilerplate Generator\textsuperscript{10}, generates a unit test boilerplate for a given C# class, setting up mocks for all dependencies. Currently it supports the Visual Studio, NUnit and xUnit testing framework and supports the following mock frameworks: Moq, AutoMoq, NSubstitute, SimpleStubs and Rhino Mocks. It also allow to customize the unit test output based on template mechanism.

3.6 Test case generation in Domain Specific Languages

Having the software testing based in DSL is not something new. A research made in 2016 by Paul Pocatilu and Alin Zamfirou has attained a test case generator based on a DSL created by them for the Android platform \cite{51}. Similarly, Sroka et al. have implemented a DSL to produce specification-based tests \cite{52}. Finally, Liguo Yu has proposed a prototyping-based testing model built in a DSL \cite{53}.

\footnote{Unit Test Boilerplate Generator, https://marketplace.visualstudio.com/items?itemName=RandomEngy.UnitTestBoilerplateGenerator#overview}
Contents

4.1 OutSystems Architecture .................................................. 39
4.2 The OML file in XML format .............................................. 41
4.3 Module import and selection - Parsine Engine ....................... 45
4.4 OutTest - User Interface .................................................. 53
4.5 Executing test cases - Dynamic Execution Engine .................. 60
4.6 Remaining items on the architecture ................................. 64
In chapter 1 the objectives for this work were identified: build a framework for automated test case generation in the OutSystems platform. A list of features to achieve the overall goal could be defined as:

- Allow the developer to import and select the module under test.
- Generate full test suites for all the methods/actions present in a given module.
- Allow the developer to run the mentioned test suites, either on demand or in a given schedule.
- Provide execution reports for scheduled tests.

4.1 OutSystems Architecture

The proposed architecture for the test framework in the OutSystems platform relied on the implementation of thirteen different modules as seen on Figure 4.1. These thirteen modules would be split by all the four layers present in the 4LC pattern, with the ones on the Library Layer to be built purely in C# using OutSystems extensions.

![Figure 4.1: A module viewpoint of the proposed architecture for the testing framework.](image)

Due to the limited access to an OutSystems platform server - namely, the Personal Edition -, the initial approach was based on parsing the Intermediate Language (IL) code emitted by invoking GetI-
The `LAsByteArray()` method on a `MethodBody` object. This last object would be a representation of the method, obtained through reflection on a given assembly - which is mapped against an OutSystems module.

### 4.1.1 Assemblies generated during compilation

During the compilation process (or 1-click publish), each module usually produces two assemblies: an assembly containing all the methods, processes and screens and a different one containing additional information related to screen flows.

In order to obtain the actions defined on those assemblies, the invoked code is present on Listing 4.1.

**Listing 4.1**: Obtaining all user actions from a given DLL.

```csharp
Assembly eSpaceAssembly = AppDomain.CurrentDomain.GetAssemblies().ToList().First(x => x.GetName().Name.Equals(sseSpaceName, StringComparison.InvariantCultureIgnoreCase));
Type actionsType = eSpaceAssembly.GetType(string.Format("ss\{0\}.Actions", sseSpaceName));
```

Then, for each method, it’s possible to retrieve its signature by using the snippet present on Listing 4.2.

**Listing 4.2**: Retrieving method’s signature.

```csharp
actionsType.GetMethods().Where(x => x.Name.Equals(string.Format("Action\{0\}". ssActionName))).ToList().ForEach(x => x.GetParameters().ToList().ForEach(y => tempList.Add(y.Name, y.ParameterType.ToString())));
```

Finally, since this is running on an OutSystems extension, the output must be mapped to the `out` parameter - as seen on Listing 4.3.

**Listing 4.3**: Mapping the output.

```csharp
RCAttributeDefinitionRecord attr = new RCAttributeDefinitionRecord();
attr. ssSTAttributeDefinition. ssName = (y.IsOut == true) ? y.Name.Substring(8) : y.Name.Substring(7);
attr. ssSTAttributeDefinition. ssType = (y.IsOut == true) ? y.ParameterType.ToString().Substring(7).Replace('&', ' ').Trim() : y.ParameterType.ToString().Substring(7);
attr. ssSTAttributeDefinition. ssIsOutput = y.IsOut;
action.ssSTDefinedAction.ssAttributes.Append(attr);
```

### 4.1.1.A Changing the path

Due to various concerns - namely in terms of time, budget and future support - the idea of using the emitted IL through reflection was ditched and was proposed to raise the development to an higher level. Raising the level would allow some abstraction to possible Common Intermediate Language (CIL) changes in the .NET Framework which would not only make the development process easier but also
would provide better information to the users. From an user perspective, it’s better to see something like: \( \text{Dividend} > 0 \) and \( \text{Divisor} > 0 \) than \( a > 0 \) and \( b > 0 \).

After several attempts, it was possible to obtain the representation of the OML file in XML format, which has all the information and relationship amongst widgets. In the following section, the structure of the XML file will be discussed.

### 4.2 The OML file in XML format

As said previously, the XML file contains all the information and the relationship amongst widgets. There are essentially two patterns in the XML structure: element properties are included as attributes of the element and any sub-element is included as a child element. On Figure 4.2, it is possible to see which properties are applicable to the ActionSamples module (see bottom right pane). These values are then translated into the XML file, as seen on Listing 4.4. One thing that actually it is possible to see is that some values are present on the XML file but they do not exist in the Service Studio representation. A possible reason is that the tool produces output for both types of modules (web or mobile) and, if the value doesn’t exist in the OML file, it just fills with the default value. That would explain fields like UpgradeRequired or DefaultTransition.
4.2.1 Actions

For the scope of this work, the focus is on the ActionsFolder element in the XML file. This is the parent element that has all the Folders - if any - containing the user actions - which is what it's intended to parse.

This action folder has zero or more elements of type ActionFolder which will have, each one, zero or more elements of the Action type. The Action element is composed by four lists: the Node list type, which is always present and contains the widgets present in the given action, the InputVariables list type, which contains the input variable definition and two other lists, for output variables (OutputVariables) and for local variables (LocalVariables). A variable list (input, output and local) only exist if the action has them - Listing 4.5 shows a simplified version.

4.2.2 Properties

As specified before, each element has a set of attributes that represent the values on the properties pane in the Service Studio. For the Actions scope, the list of parameters is the following:

- **Key**: unique identifier of the action.
- **CreatedBy**: the credentials of the user who created the action in the first place.
• **LastModifiedBy**: the credentials of the user who changed the action in the last place.

• **LastModifiedDate**: date of the latest change to the action. Could be the creation date, if the same.

• **Name**: name of the action.

• **Public**: if the action is accessible from outside of the module or not.

• **IconLength**: property related to the icon of an action. If this value is zero, then the action has the default icon.

By using the **LastModifiedDate** field of the action - see Listing 4.6 -, it is possible to understand if the action was changed and a subsequent alert should be issued to the user.

### Listing 4.6: Action element properties

```xml
<Action Key="Action:zcN0Nu7j5kCtRjLH4cTSWW" CreatedBy="armandojsgomes@gmail.com" LastModifiedBy="armandojsgomes@gmail.com" LastModifiedDate="2018-08-12 23:03:44" Name="Division" IconLength="0"/>
```

Other widgets may have more or less properties, depending on the cases. For instance, all the graphical elements used for development contain two additional attributes, namely the X and Y position on the screen.

#### 4.2.3 Widgets and Connectors

Widgets, by default, follow the same pattern than actions and modules: all of the information about the widget is stored in the attributes of the element. Additionally, all of the widgets (except the **End**, **Download** and **RaiseException** widgets) contain a list of connectors. Referring to Figure 4.2, connectors are represented by the arrows.

The vast majority of widgets only have one connector entry in the connector’s list. A given connector only contains two attributes: the unique key and the target, which value is the key of the target node. The others have two or more connectors: **If**, **Switch** and **ForEach**. The **If** widget has exactly two connectors, for the **true** and **false** branches. The **ForEach** also has two connectors, for the logic inside the loop and the next instruction once the loop has ended. Finally, the **Switch** node has many connectors as cases plus one - for the default (**Otherwise**) case. As an additional note, the **Switch** widget might have only one connector (**Otherwise**) and still be valid - obviously, the odds of founding this are really low and, if found, represent a programming mistake.

The widgets **End**, **Download** and **RaiseException** represent the end of a given flow, thus not having any connector.
4.2.4 Specific functionality

Properties and connectors follow the same pattern and are present in the vast majority of the widgets. But what about specific functionality?

An Assign widget - which assigns values to variables -, has a list of Assignments, in which each Assignment has a Variable-Value pair. The ExecuteAction - which executes another user action - has a list of Arguments, with a Parameter-Value pair. Other examples are the Switch widget, which has a Condition-Connector pair and the Aggregate widget.

The Aggregate widget is possibly the most complex widget. It contains a Table element that contains another element called DataTable and, finally, a child element called TableOperations. Inside the TableOperations element, the sources (tables) are identified (AddSource node) and the JOIN, FILTER and SORT conditions are specified. A simplified sample, with some attributes removed, of an Aggregate can be seen on Listing 4.7.

Listing 4.7: Aggregate XML element

```xml
<Aggregate Key="Aggregate:FQSWuETbX0mTYmfNyh6e4Q" Name="GetSampleEntities">
  <Connectors>
    <Connector Key="Connector:XeHdYZNgSUy4yOsV7BnOAA" Target="End:Eqf3b_BOEi4A7x8Qq2maA"/>
  </Connectors>
  <Table>
    <DataTable Key="DataTable:FQSWuETbX0mTYmfNyh6e4Q.#Table" RootOperation="CombineSources:7oA_CNrHCKicAZfl0qNrmw" Name="SampleEntities">
      <TableOperations>
        <CombineSources Key="CombineSources:7oA_CNrHCKicAZfl0qNrmw" Name="SampleEntities">
          <Sources>
            <DataSource Key="DataSource:2Sk9Z+PC+0WoC2pi4u3nWQ" Name="SampleEntity" SourceOperation="AddSource:FbMGU163jUK5Mg1AN77uZw"/>
          </Sources>
          <Filters>
            <Filter Key="Filter:STYA5LQd00evGYg+x9L3hg" Condition="SampleEntity.Description like "&quot;%&quot; + Session.SampleEntities_SearchKeyword + "%""/>
          </Filters>
          <Sorts>
            <AttributeSort Key="AttributeSort:PB61E69ky0aY+Vrkh8OLVg" IsDynamic="Yes" Attribute="List_SortColumn_GetOrderBy/SampleEntityTable.Id, &quot;{SampleEntity}.[Description]"&quot;/>
          </Sorts>
        </CombineSources>
      </TableOperations>
      <DataTable>
        <AddSource Key="AddSource:FbMGU163jUK5Mg1AN77uZw" Source="Entity:Rw5bqtJQ2ESUka9r6uw"/>
      </DataTable>
      <ImplicitParameters>
        <ImplicitParameter Key="ImplicitParameter:byeThom750wZZFS5W1pKQ" DataType="Text" Value="List_SortColumn_GetOrderBy/SampleEntityTable.Id, &quot;{SampleEntity}.[Description]"&quot;/>
        <ImplicitParameter Key="ImplicitParameter:a7mbo2H5qbrj29TthyWA" DataType="Text" Value="Session.SampleEntities_SearchKeyword"/>
      </ImplicitParameters>
    </Table>
  </dataTable>
</Aggregate>
```
4.2.5 Generating the XML Schema Definition (XSD) and the C# class

In order to use C# default capabilities to deserialize the XML file into a C# class, two steps were required: create the XSD schema and, from the schema, create the C# class. That could be achieved using the xsd.exe tool included in the .NET Framework.

4.2.5.A Jagged Arrays

A jagged array is an array whose elements are arrays. Due to the nature of some elements present on the XML file, it was required to generate the XSD and the corresponding C# class with specific steps as stated on Microsoft's website\(^1\). These steps involved the creation of a Dynamic-Link Library (DLL) file specifically for this purpose along with a XML file, making the executed command as:

```
Listing 4.8: Generate XSD and C# class commands.

xsd ActionSamples.XML.xml
xsd /p:GenerateClasses.xml ActionSamples.XML.xsd
```

4.3 Module import and selection - Parsine Engine

With the generated C# class for XML deserialization, the Parsing Engine is built using an OutSystems extension. This extension, as shown on Figure 4.3, receives the uploaded binary (the XML file) and returns the following information:

- **eSpaceName** which is the module’s name.

- **ActionList** is a structure composed by other structures. It contains the Action name, when the last modification occurred and two lists: parameter and branch list. The parameter list contains the name of the parameter, its type (input, output, local, computed), the data type - i.e. integer, decimal, currency and so on -, if it’s mandatory and when the last modification occurred. As for the branch list, only the computed condition for the branch is returned.

- **Error** which is a structure composed by Error Code and Message.

In order to correctly parse the elements of the deserialized XML file, a specific DLL named OutSystems Parser was built using Visual Studio.

\(^1\)https://support.microsoft.com/pt-pt/help/2967637/jagged-arrays-that-contain-xml-element-tags-are-projected-when-you-gen
4.3.1 OutSystems Parser DLL

On Figure 4.4 it is possible to see the project structure for the OutSystems Parser DLL. Essentially, it is composed of three major types:

- The **Espace** class, at the root of the project, is the generated class using the XSD tool.

- The classes inside the **Widgets folder** are the different types of widgets possible to use within the OutSystems platform.

- The classes inside the **Global folder** - inside the Widgets folder - contain "internal" representations of the widgets and respective functionality for the OutTest framework.

All of the classes inside the **widgets** folder extend the **Widgets.Global.Node** class. In writing form, a **ServerAction** has a list of **Parameters** and **Nodes**, in which each one has a list of **connectors**. The purpose of the **ConnectorReference** class will be discussed on the **Parsing nodes** subsection.

Related to the initial parsing of the binary received as an input parameter, this is done using a static function from OutSystemsParser.dll - as seen on Figure Listing 4.9:

**Listing 4.9**: Deserializing the binary file.
Figure 4.4: OutSystems Parser project structure.

```
ssActionList = new RLActionRecordList();
ssError = new RCErrorRecord();
ssENameSpace = "";
ESpace eSpace;
#region Parse File
try
{
    eSpace = ParseEspace(System.Text.Encoding.UTF8.GetString(ssBinaryFile));
    ssNameSpace = eSpace.Name;
}
catch(Exception ex)
{
    SetError(1, ex.ToString(), ssError);
    return;
}
#endregion

Once this deserialization process is done, the next step is to iterate the ActionFolders and, for each, iterate existing Actions:

Listing 4.10: Iterating the existing actions using foreach loops.

```
foreach(ESpaceActionFoldersActionFolder actionFolder in eSpace.ActionFolders.ActionFolder)
{
    foreach(OSServerAction action in actionFolder.Actions.Action)
    {
        RCActionRecord actionRecord = new RCActionRecord();
        ServerAction serverAction = new ServerAction(action.Key, action.Name);
        actionRecord.ssSTAction.ssName = action.Name;
    }
```
In Listing 4.10 is shown the process used to iterate the ActionFolders and respective actions. Then, for each action, a new RCActionRecord is created - the RCActionRecord is used for output purposes - and the non-structure values are set: name and the date of last modification. Additionally, the remaining output variables are also initialized.

4.3.2 Parsing variables

For each action, beyond creating the RCActionRecord and initializing the default values for the branch and parameter list, it is important to parse any existing parameter - either input, output or local. In Listing 4.11 it's shown the method invocation so the variables can be parsed - for the three types - while in Listing 4.12 it's shown in detail the body of the GetInputParameters method. The GetOutputParameters and GetLocalVariables method body are very similar.

Listing 4.11: Parsing variables of a given action.

```csharp
#region Action Parameters
if (action.InputParameters != null)
    GetInputParameters(action, serverAction, actionRecord);
if (action.OutputParameters != null)
    GetOutputParameters(action, serverAction, actionRecord);
if (action.LocalVariables != null)
    GetLocalVariables(action, serverAction, actionRecord);
#endregion
```

Listing 4.12: GetInputParameters method.

```csharp
static void GetInputParameters(OSServerAction action, ServerAction serverAction, RCActionRecord actionRecord)
{
    List<Parameter> output = new List<Parameter>();
    foreach (ActionInputParameter inputParameter in action.InputParameters.InputParameter)
    {
        Parameter param = new Parameter(inputParameter.Key, inputParameter.Name,
                                         inputParameter.DataType,
                                         inputParameter.IsMandatory != null && inputParameter.IsMandatory.ToUpper().Equals("YES"));
        RCPParameterRecord parameterRec = new RCPParameterRecord();
        parameterRec.ssSTParameter.ssName = inputParameter.Name;
        parameterRec.ssSTParameter.ssDataType = inputParameter.DataType;
        parameterRec.ssSTParameter.ssLastErrorMod = DateTime.ParseExact(inputParameter.LastModifiedDate, "yyyy-MM-dd HH:mm:ss",
                                                                        System.Globalization.CultureInfo.InvariantCulture); 
        actionRecord.ssSTAction.ssParameterList.Add(parameterRec);
        serverAction.AddParameter(param);
    }
}
```
Once the values are correctly parsed, they are added to an instance of the ServerAction type, received as an input.

### 4.3.3 Parsing nodes

Once variables are parsed, the next step is to parse nodes. The process is achieved using a foreach loop as well, this time iterating the node items present on current action’s node list - as seen on Listing 4.13.

**Listing 4.13: Parsing the existing nodes.**

```csharp
foreach (object node in action.Nodes.Items) {
    Type type = node.GetType();
    if (type == typeof(OSNodeStart)) {
        serverAction.AddNode(ParseNodeStart((OSNodeStart) node));
    } else if (type == typeof(OSNodeEnd)) {
        serverAction.AddNode(ParseNodeEnd((OSNodeEnd) node));
    } else if (type == typeof(OSNodeAssign)) {
        serverAction.AddNode(ParseNodeAssign((OSNodeAssign) node));
    }
    (...)
}
```

Listing 4.13 is a short representation of the full snippet of code, due to the possible 15 types of nodes. The OSNode prefix represents the module deserialization to “native” types - not the encapsulation presented on the OutSystemsParser DLL. It was considered the usage of the visitor design pattern but due to time constraints, the refactoring was not possible.

For each type of OSNode types, a ParseNode method is invoked. All these methods are similar, changing the way specifics are parsed. The ParseNodeIF method can be seen on Listing 4.14.

**Listing 4.14: Parsing the OSNodeIF.**

```csharp
static If ParseNodeIf(OSNodeIF ifNode) {
    If @if = new If(ifNode.Key, ifNode.Condition);
    foreach (object condition in ifNode.Connectors.Items) {
        if (condition.GetType() == typeof(OSNodeIfTrueBranch)) {
            OSNodeIfTrueBranch trueCondition = (OSNodeIfTrueBranch)condition;
            @if.AddConnectorReference(ifNode.Key, trueCondition.Target, true);
        } else {
            OSNodeIfFalseBranch trueCondition = (OSNodeIfFalseBranch)condition;
            @if.AddConnectorReference(ifNode.Key, trueCondition.Target, false);
        }
    }
    return @if;
}
```
Regarding the \textit{ParseNodeIf} method, it is important to highlight the following: when parsing a given node, the connector(s) target(s) might not have been parsed yet. As such, an auxiliary class is used - \textit{ConnectorReference} - which, instead of pointing to another Node (object), it just points to a string.

### 4.3.4 Computing the next node

Overall, the process of computing the next node is just to "convert" an object of type \textit{ConnectorReference} to the \textit{Connector} type. The difference between both is simple: \textit{ConnectorReference} points to a string - the key of the target - while the \textit{Connector} points to a given node - an object.

As such, the first step is to populate an auxiliary dictionary of key-object, with the key being the unique identifier of the node and the object being the node itself in a \textit{foreach} loop.

Then, the next step is to iterate the nodes already added to the \textit{ServerAction} object - \textit{OutSystemsParser.dll} type - and, depending on the type of node, compute the connector(s). Instead of using a \textit{foreach} loop, a \textit{for} loop is used. The reason for the change is that there is the need to replace the objects on the list being iterated, as seen on Listing 4.15.

This listing is a shortened version of the full method but, essentially, what is done is that for specific types of objects (\textit{ForEach}, \textit{Switch} and \textit{If}), additional logic is required to compute the targets, either due to being a loop or having multiple conditions. Then, if the current node is none of these special cases, we validate if its type is \textit{RaiseError}. If found a \textit{RaiseError} node, a validation takes place to see if in the scope of the given action, the exception is handled. If so, that is considered the next node in the flow. If not, it ends there. Finally, for "regular" node - which doesn’t fit the special cases -, a simple mapping is done.

Listing 4.15: Computing the next node.

```csharp
for ( int iterator = 0; iterator < serverAction.GetNodes().Count; iterator++)
{
    Node node = serverAction.GetNodes()[iterator];
    if (node.GetType() == typeof(ForEach))
    {
        ForEach temp = ((ForEach)node);
        temp.AddConnector(temp.GetCycleReference().GetTarget(),
                          nodeList[temp.GetCycleReference().GetTarget()], true);
        temp.AddConnector(temp.GetContinueReference().GetTarget(),
                          nodeList[temp.GetContinueReference().GetTarget()], false);
        temp.Clean();
        serverAction.GetNodes()[iterator] = temp;
    }
    (...
    else
    { }
```
4.3.5 Creating branches

With the "next node" available and the control flow graph completed, it is possible to identify which conditions lead to each path and, as a result, identify which test cases will achieve branch coverage and, likely, statement coverage.

In order to compute the required conditions, two additional variables are required. The first is a List of Node Lists. This is used to identify the full "path" of each condition. The second is a dictionary linking these paths to the given computed condition. Given that the first element in each action is always a Start node, it is obtained manually it is passed into the BuildNodeTree method - as seen on Listing 4.16.

```
Listing 4.16: Computing the branches and respective conditions.
```

```
List<List<Node>> branches = new List<List<Node>>();
Dictionary<List<Node>, string> conditions = new Dictionary<List<Node>, string>();
Node currentNode = serverAction.GetNodes().OfType<Start>().Single();
BuildNodeTree(branches, new List<Node>{currentNode}, currentNode, new List<ForEach>(), conditions);
foreach (List<Node> branch in branches)
{
    RCBranhRecord branchRecord = new RCBranhRecord();
    try
    {
        branchRecord.ssSTBranch.ssCondition = conditions[branch];
    }
    catch (Exception)
    {
        branchRecord.ssSTBranch.ssCondition = "Unique branch."
    }
    actionRecord.ssSTAction.ssBranchList.Add(branchRecord);
}
ssActionList.Add(actionRecord);
```

The BuildNodeTree method receives a set of input parameters, each with their specific purpose:

- The first parameter is the already computed branches. This is a list that is changed inside the BuildNodeTree method and is used into subsequent calls.

- The second parameter is the current List<Node> object to be computed. In the first invocation, it only has the Start node.

\(^2\text{Inline if conditions are not being considered as of now}\)
• The third parameter is the start node. It is used to compute the next node and proceed with the branch calculation.

• The fourth parameter is a list of ForEach nodes. This is required since the logic inside a loop will always end in the ForEach node. As such, during branch calculation, it's important to understand if the cycle connector was already used so it can proceed to the exit connector.

• The fifth and final parameter is a dictionary containing the mapping between a listing of Node List and the computed condition.

The full body of BuildNodeTree method is available in Listing 4.17. This is a recursive function that only stops when all the branches have reached their End node - with this End node being the final node of the flow.

Listing 4.17: BuildNodeTree method body.

```csharp
static void BuildNodeTree(List<List<Node>> branches, List<Node> currentBranch, Node startNode,
List<ForEach> executedForEaches, Dictionary<List<Node>, string> conditions)
{
    if (startNode.GetConnectorCount() == 1)
    {
        Node nextNode = startNode.GetNextNode();
        if (nextNode.GetType() != typeof(End))
        {
            currentBranch.Add(nextNode);
            BuildNodeTree(branches, currentBranch, nextNode, executedForEaches, conditions);
        }
        else
        {
            currentBranch.Add(nextNode);
            branches.Add(currentBranch);
        }
    }
    else if (startNode.GetConnectorCount() > 1)
    {
        if (startNode.GetType() == typeof(ForEach) && !executedForEaches.Contains(startNode))
        {
            ForEach forEachNode = ((ForEach)startNode);
            executedForEaches.Add(forEachNode);
            List<Node> futureBranch = Node.CloneNodeList(currentBranch);
            AddCondition(conditions, futureBranch, forEachNode.GetCycleCondition(), currentBranch);
            BuildNodeTree(branches, futureBranch, forEachNode.GetCycleNode(), executedForEaches,
            conditions);
            AddCondition(conditions, currentBranch, string.Format("[0] is empty", forEachNode.GetCycleCondition()));
            BuildNodeTree(branches, currentBranch, forEachNode.GetContinueNode(), executedForEaches,
            conditions);
        }
        else if (startNode.GetType() == typeof(ForEach) && executedForEaches.Contains(startNode))
        {
            ForEach forEachNode = ((ForEach)startNode);
            currentBranch.Add(forEachNode.GetContinueNode());
            BuildNodeTree(branches, currentBranch, forEachNode.GetContinueNode(), executedForEaches,
            conditions);
        }
        else if (startNode.GetType() == typeof(Switch))
        {
            Switch switchNode = (Switch)startNode;
            BuildNodeTree(branches, currentBranch, switchNode.GetContinueNode(), executedForEaches,
            conditions);
        }
        else
        {
            throw new Exception("Invalid node type.");
        }
    }
    else
    {
        throw new Exception("Invalid node type.");
    }
}
```
Switch switchNode = ((Switch)startNode);
foreach (SwitchCondition sc in switchNode.GetConditions())
{
    List<Node> futureBranch = Node.CloneNodeList(currentBranch);
    futureBranch.Add(sc.GetNode());
    AddCondition(conditions, futureBranch, sc.GetCondition(), currentBranch);
    BuildNodeTree(branches, futureBranch, sc.GetNode(), executedForEaches, conditions);
}
currentBranch.Add(switchNode.GetOtherwiseCondition().GetNode());
AddCondition(conditions, currentBranch, switchNode.GetOtherwiseCondition().GetCondition());
BuildNodeTree(branches, currentBranch, switchNode.GetOtherwiseCondition().GetNode(),
    executedForEaches, conditions);
else
{
    If @if = ((If)startNode);  
    List<Node> falseBranch = Node.CloneNodeList(currentBranch);
    falseBranch.Add(@if.GetFalseBranchNode());
    AddCondition(conditions, falseBranch, string.Format("(!{0})", @if.GetCondition()),
        currentBranch);
    currentBranch.Add(@if.GetTrueBranchNode());
    AddCondition(conditions, currentBranch, @if.GetCondition());
    BuildNodeTree(branches, falseBranch, @if.GetFalseBranchNode(), executedForEaches,
        conditions);
    BuildNodeTree(branches, currentBranch, @if.GetTrueBranchNode(), executedForEaches,
        conditions);
}
else
{
    branches.Add(currentBranch);
}

The logic is quite simple. Using the method’s start node, a check is made to see if the current node has more than one connector. If not, the next node is retrieved and, depending whether it’s an End node or not, the method is terminated or invoked again - but only after adding the next node into the current branch.

If the connector count is greater than one, then an additional validation is made. For nodes of type Switch and If, the BuildNodeTree function is invoked again for each of the possible connectors. In the case the node is of type ForEach, then another validation takes place: does the current node has already been used? If not, the path explodes into two - one for the logic inside the ForEach and other for the case in which the list used in the ForEach is empty. If the ForEach node has already been iterated, then the “continue” node is retrieved - which is the same as if the list was empty - and the remaining path is computed.

4.4 OutTest - User Interface

When the user decides to use the OutTest framework, the initial tasks the user needs to carry on is to login. Once logged in, the user will be redirected to the dashboard - or landing page - where it is possible to see some information regarding the the latest and scheduled test executions.
4.4.1 Landing Page

The landing page is a dashboard that contains the following information, from left to right, top to bottom:

- Latest and next scheduled test execution.
- Number of test cases executed per day and module
- Graph with regressions and a burn up information
- Latest test executions with errors

![Figure 4.5: OutTest dashboard](image)

From the dashboard it is also possible to navigate through the menus or use the quick links for a specific action. As seen in Figure 4.5, this is a reference screen with a read-only perspective on the overall state of the system.

4.4.2 A typical test suite

Creating a test suite requires that a valid XML file is imported - see Figure 4.6. Once imported, the user is presented with the current test suite list for the imported module - as seen on Figure 4.7.

This list contains the name of the user action under test, the module (eSpace) to which it belongs and the number of test cases for that action. The number of test cases, by default, is the number of
different branches present on the user action. Additionally, the status of the test suite and a link for the execution is presented.

4.4.3 Executing one test

The test suite for the Division user action - shown on Figure 4.8 - lists the following information: action details, which is the same as presented on the list before along with the values present on the user action and, finally, the identified branches - see Figure 4.9.

For each identified branch, the user has the possibility to specify the input values, the expected output values and additionally, computed values. In the computed values section the user is allowed to specify mocked values. This will be detailed in the Mocking values subsection.

On Figure 4.10 and 4.11 it is possible to see how the screen behaves when the test is manually
executed and the result is either a success or a fail. In the case of a failure, an error mark is displayed and the received result is shown to the user. This way, the user can quickly understand what's going on.

Due to security constraints in place related with Application Domains, it is not possible to execute actions from one module in another module. In a more concrete example, the OutTest module can't directly execute actions in the ActionSamples module through the Dynamic Execution Engine. In order to bypass this limitation, the developer needs to "copy and paste" a pre-built web service definition so the OutTest framework can execute the actions. The URL for that service should be /<ModuleName>/rest/Runner/Invoke. This web service is already prepared for execution, reducing the overhead. In the Chapter 6 a possible improvement to overcome this limitation is proposed.

4.4.3 A Solving the constraints

Due to time limitations, it was not possible to include the implementation of a solver considering the constraint expressions generated by the OutSystems platform. Although it was not implemented when this document was written, the Z3 Theorem Prover from Microsoft is being evaluated for inclusion³.

---

³https://github.com/Z3Prover/z3
Figure 4.9: Branches for Division action.

Figure 4.10: Test executed and returned success.
4.4.4 Mocking values

In order to assure the repeatability of tests, computed values - values which influence the outcome but are not input parameters - need to be mocked. These values can be one from either type: Session Variables, Site Properties or External Calls.

4.4.4.A Session Variables

Figure 4.12 shows the execution of a test using a mocked value. In the computer parameters part, the name of the session variable is defined along with the data type and the parameter type. This will be mapped into the Dynamic Execution Engine invocation. Since our mocked value is different than expected, an execution error is shown.

4.4.4.B Site Properties

In the same fashion as Session Variables, the user has the possibility to "mock" Site Property values. Identical behavior can be seen on Figure 4.13.

It is important to refer that this is not a "pure" mock like session variables. A more detailed explanation about the mocking process for site properties is given in the Dynamic Execution Engine section.
4.4.4.C External Calls (Database and other Actions)

When the code generator integrated with the OutSystems Platform Server generates the code, static methods are created. Unfortunately for the scope of this work, it has been proven unfeasible to mock such static methods, even using commercial tools like JustMock. Despite being able to mock static methods, it requires that the invocation is done within the scope of Visual Studio - which is not the case, since the code is running on the server. Unless an improvement to the code generator is made - proposed in Chapter 6 -, it's not feasible to advance on this vector.
4.5 Executing test cases - Dynamic Execution Engine

The Dynamic Execution Engine is the piece responsible to execute a given test case using the parameters defined. The process for executing the test case relies on four steps: obtaining the action to be executed, mocking applicable values, setting up input/output variables and, finally, invoke and parse the output.

![Figure 4.14: Dynamic Execution Engine signature](image)

4.5.1 Getting the action

As seen on Figure 4.14, when invoking the Dynamic Execution Engine the developer must provide the name of the module \(eSpaceName\), the name of the action to be executed \(ActionName\) and then a structure which contains three items: a list of parameters (key-value pairs, mandatory), an error message for output - since the same structure is used for output - and a MockedParameters list. The full details of this list can be seen on Figure 4.15.

Each record of this type contains six parameters:
Figure 4.15: MockedParameters structure

- **ListName** which is the name of the list, if applicable - for instance, the output of an aggregate or SQL query.

- **AttributeName** which is the name of the used attribute. Can be either from a list - i.e. `List.Current.AttributeName` - or a site property/session variable.

- **Value** is the value that the variable should return.

- **IsEmpty** represents if the variable should be mocked as an empty list.

- **IsSessionVariable** if it is a session variable.

- **IsSiteProperty** if it is a site property.

In order to retrieve the action through C#, the code presented in Listing 4.1 is used. Once the method is correctly obtained, the Dynamic Execution Engine proceeds to the next step: mocking values.
### 4.5.2 Mocking values

Considering the limitations described in previous section, the mocking process is implemented for *Session Variables* and *Site Properties*. This is achieved by iterating over the *MockedParameters* input structure. Listing 4.18 shows the full snippet of code.

```csharp
#region Setup Mocking
if (ssInputParameters.ssSTInvokeStructure.ssMockedParameters != null &&
    ssInputParameters.ssSTInvokeStructure.ssMockedParameters.Length > 0)
{
    foreach (RCMockedParametersRecord mockingRecord in
        ssInputParameters.ssSTInvokeStructure.ssMockedParameters)
    {
        if (mockingRecord.ssSTMockedParameters.ssIsSessionVariable)
        {
            AppInfo appInfo = (AppInfo)eSpaceAssembly.GetType(string.Format("ss\{0\}.Global", sseSpaceName)).GetProperty("App").GetValue(null);
            appInfo.OsContext.Session[string.Format("\{0\}.\{1\}", sseSpaceName, mockingRecord.ssSTMockedParameters.ssAttributeName)] =
                mockingRecord.ssSTMockedParameters.ssValue;
        }
        else if (mockingRecord.ssSTMockedParameters.ssIsSiteProperty)
        {
            SitePropertiesInfo spi =
                (SitePropertiesInfo)eSpaceAssembly.GetType(string.Format("ss\{0\}.Global", sseSpaceName)).GetProperty("SiteProperties").GetValue(null);
            spi[mockingRecord.ssSTMockedParameters.ssAttributeName] =
                mockingRecord.ssSTMockedParameters.ssValue;
        }
    }
#endregion
```

When iterating over each *MockerParameter* record in the input structure, an additional check is made to see if it’s a *Session Variable* or a *Site Property*.

For a *Session Variable*, the value of the *App* property present in the *ss(eSpaceName).Global* type contains the *AppInfo* variable. This *AppInfo* variable contains a property named *OsContext* which, in turn, contains a *Session* dictionary. By adding a value to this dictionary - which key follows the eSpace.Attribute naming convention - it is possible to mock the value for the running session. Since this action is invoked with a different *OsContext* object every time, there are no concurrency issues.

For *Site Properties*, a similar approach is taken but instead of obtaining an *AppInfo* object, the code retrieves the *SitePropertiesInfo* object. This object is a dictionary and, contrary to the *AppInfo* object, it only contains values for the given module. As such, a direct assignment can be done through *dictionary*[attribute] = *value*. In terms of concurrency, there are some limitations here. While for *Session Variables* the *OsContext* object is created before each invocation, for *Site Properties* it isn’t mocking *per se*. Actually, the code changes the value of the *Site Property* as if it was done manually, being persisted between execution calls.
4.5.3 Setting up variables

With the mocked values in place, it is time to fill the object array containing all parameters. In order to achieve correct variable setup, a two-phase process occurs.

The first phase involves iterating across the temporary list containing the parameter order and either add the value or add a null object. The second phase involves modifying some parameters so that their value can be set by the invoked method. This is caused by the method being defined with output parameters passed by reference. Listing 4.19 shows the complete snippet of code.

Listing 4.19: Filling the parameters before method invocation.

```csharp
foreach (string key in tempList.Keys)
{
    if (key.Contains("outParam"))
        argumentList.Add(null);
    else
    {
        foreach (RCParameterRecord parameterRecord in ssInputParameters.ssSTInvokeStructure.ssParameters)
        {
            if (parameterRecord.ssSTParameter.ssName.Equals(key.Substring(7))
                argumentList.Add(Convert.ChangeType(
                    parameterRecord.ssSTParameter.ssValue,
                    Type.GetType(tempList[string.Format("inParam{0}",
                        parameterRecord.ssSTParameter.ssName)])));
            break;
        }
    }
}
object [] argumentArray = argumentList.ToArray();
ParameterModifier parameterModified = new ParameterModifier(argumentArray.Length);
int i = 0;
for (i = 0; i < argumentArray.Length; i++)
{
    if (argumentArray[i] == null)
        parameterModified[i] = true;
}
ParameterModifier[] mods = { parameterModified };
```

4.5.4 Invocation and output

The last step is to invoke the action with the specified parameters and return the output into the OutTest framework. Listing 4.20 shows how this is achieved.

Listing 4.20: Parsing the output of action invocation.

```csharp
actionsType.InvokeMember(string.Format("Action{0}". ssActionName),
    BindingFlags.InvokeMethod | BindingFlags.Static | BindingFlags.Public,
    null, null,
    argumentArray,
    mods,
    null,
```
The `InvokeMember` method is a method from the `Type` class, which represents type declarations: class type, value types, enumeration types, interface types and so on. This specific invocation uses the method’s name - `ActionForEach` for instance - , a set of binding flags - which defines the way member and type search is conducted by reflection - and two other parameters: `argumentArray` which has all the variables (including input and output) and, finally, the `mods` parameter. The `mods` parameter is a `ParameterModified` struct that defines which parameters are passed by reference. Due to how the code is generated by the OutSystems platform, all the output parameters are passed by reference.

Once the invocation was successful and knowing the order of the parameters hasn’t changed, it is possible to iterate through a temporary dictionary containing parameter’s order and type and map that against the `argumentArray`. Since the generated code prefixes input parameters as `inParam` and output parameters as `outParam`, that string is used to check whether the parameter is an output one. If so, then an output record is created and added to the output structure.

### 4.6 Remaining items on the architecture

Regarding the non-discussed items that were present in the proposed OutSystems architecture, some of the elements were discarded due to their nature (Test Generator Engine and Mocking Engine), along with CoreUser and CoreSchedule - which were merged into the CoreTest module. The proposed end user modules are integrated in the OutTest module.
## Evaluation

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Demographics</td>
<td>67</td>
</tr>
<tr>
<td>5.2 Procedure</td>
<td>67</td>
</tr>
<tr>
<td>5.3 Results</td>
<td>68</td>
</tr>
<tr>
<td>5.4 Financial analysis</td>
<td>68</td>
</tr>
<tr>
<td>5.5 Overall comments</td>
<td>68</td>
</tr>
<tr>
<td>5.6 Objective evaluation</td>
<td>69</td>
</tr>
</tbody>
</table>
The framework was evaluated during a period of two days, using a predefined set of functions. The lack of “real world” examples was due to intellectual property rights over existing projects along with the unnecessary overhead to create a working application. Summing up, the aim was to focus on concrete scenarios which would be identified as usual bugs - and that would be created by a junior developer.

5.1 Demographics

The evaluation of the produced framework was conducted using five junior developers from two different companies - one from Portugal and one from the United States of America. Due to the nature of the results, both companies requested to remain anonymous. Regarding the developers, it was their first professional role and their experience with the platform was low. For two of those five developers, their field of studies was not related to computer science neither programming - which is the result of the current skill shortage.

5.2 Procedure

The procedure implemented was the following:

1. Developers were given a working eSpace with 4 existing bugs (unknown to developers) and were requested to test the actions presented on that eSpace. The total time taken would be recorded and found errors were reported.

2. Developers were then given an XML version from the same working eSpace and were requested to import it into the OutTest framework and build the required tests. The total time taken would be recorded and found errors were reported.

3. Once these two tasks were completed, a modified version of the working eSpace was given to the developers and they were asked to find the manually created bugs. Time taken would be recorded and compared between both solutions. This modified version had two additional bugs.

4. Once found, developers were asked to fix them.

5. Finally, developers were invited to share their thoughts about their performance using each method and overall comments on the built framework.
5.3 Results

- Regarding test suite creation time, developers took on average 80% less time creating the test suite using the OutTest framework instead of the manual procedure (1.5 hours instead of 7.5).

- Regarding original bug discovery using manual methods, only 75% of the bugs were discovered versus the 100% discovery using OutTest.

- Regarding bug discovery using modified versions, only two developers were able to correctly identify the two new bugs using manual methods while all the developers were able to find them using OutTest.

- In terms of timings for bug discovery in the modified versions, an average of 30 minutes using OutTest was required versus an average of 80 minutes when doing manually.

- As for bug fixing time, the average time taken was one hour for both bugs.

5.4 Financial analysis

Considering an average cost of EUR15/Hour/Developer and considering a team of five junior developers, the usage of the OutTest framework yielded an average saving of EUR90 per developer just for test case design. Additionally, an approximate EUR15 was saved when developers had to re-test.

Considering the team a whole (five people), using manual methods would have cost approximately EUR887 to the company, without having pristine actions - 7.5h for the test creation, an estimated 2 hours to fix the original four bugs, 1.33h for the bug discovery in the modified versions and an additional hour for fixing those. If the usage of the OutTest framework was considered, the total cost would have been EUR375, a saving of almost 58%!

The costs presented only take into account direct costs, although the results show an increase in productivity and an overall increase in the quality.

5.5 Overall comments

At the end of the tests, both developers and managers were requested to give their opinion on the framework.

The developers stated that they felt an increase in their productivity and that their work had become "less boring". Additionally, they felt that the usage of a framework like this would improve their code quality and the overall relationship with peers, managers and clients - since less bugs would be found.
As of managers, they stated that the financial gain was considerable along with the increase in the quality of the deliverables, which would lead into better a relationship with the customer. Unfortunately, the lack of possibility to simulate outside calls (database and/or other actions) would restrict the overall applicability of this framework in "real-world" projects.

5.6 Objective evaluation

Considering the final result along with the initial objectives, it is possible to say that not all of the objectives were completely satisfied. In terms of the main requirements, the lack of mocking capabilities for external calls is a considerable limitation in order to allow the user to generate full test suites for all the methods/actions present in a given module. In terms of additional requirements, a better looking UI should be considered. In terms of size related to Application Objects count, the produced framework had a total of 40, 20% below the maximum defined value.
6 Conclusion

Contents

6.1 Conclusions ......................................................... 73
6.2 Current limitations and future work .............................. 73
6.1 Conclusions

The increased complexity in software along with the current talent shortage led to companies having to hire professionals outside of the IT market, leading to junior developers with less skills. Having less skills and, more important, less relevant quality assurance skills, the quality of deliverables is at skate, leading to delayed projects, financial losses and public image degradation.

In order to overcome the repetitiveness and "boredom" of the testing tasks, automated software testing can be implemented in companies, although it requires initial investment and certain skills. The process of automated testing can be done using various techniques like Fuzzing, Random testing, static analysis, symbolic execution and so on.

The objective of this work was to create a testing framework for the OutSystems platform so user actions - usually linked to business logic - could be tested without further developments.

Despite this work not being supported by OutSystems and the existing limitations (discussed next), the achieved results seem to show that this was a success and the developments should continue, ideally with OutSystems support.

6.2 Current limitations and future work

As stated previously, there are some limitations to this work that influence the overall functionality and applicability.

The first limitation is the need to have an XML version of the module (eSpace) so it can be imported. As of now, this feature is not available to a regular OutSystems developer or customer, with the file being only accessible through OutSystems employees. A possible workaround is to have the XML file to be automatically generated by the Platform Server and also automatically imported into OutTest.

Regarding the limitation of function invocation due to the Application Domains, it was suggested that a possible solution would be to automatically reference every action a "super" module and automatically compile it - without user intervention. This could, eventually, bypass the need of the web service implementation.

Regarding the mocking limitation for Site Properties, the suggestion is that the value is obtained through a function, and that function also receives an OsContext object.

Regarding external calls limitation - and explaining the need of the OsContext in the Site Property function -, it would be helpful if the generated code would include a functionality which would check the said OsContext.Session dictionary for a given value - TestMode, for instance - and if the value was true, it would then return the object present in another entry of the dictionary - TestResult, for instance. This way, mocking would be possible.
Regarding future work, the inclusion of a solver could be proven a huge leap forward along with a redesigned and more effective UI. In terms of the extensions, refactoring of the code to properly implement design patterns along with performance optimization is considered.

Finally, several former employees of OutSystems along with current customers have shown interest in the produced framework, either from a "usage perspective" (customers) or to "join development efforts" (former employees).

As an informal feedback, all the senior developers that were asked about which feature would be more important for them - either mocking of external actions or a solver for the conditions - have selected the mocking since it would allow them to reproduce erroneous behaviors from productive environments in non-productive environments. They have considered that, despite being a plus having a tool to calculate branch's input values, they would prefer to do it manually - at least until the calculation was proven 100% accurate and efficient.
Bibliography


