An automated unit testing framework for the OutSystems Platform

Armando João da Silva Gomes
Instituto Superior Técnico
Lisboa, Portugal
armando.gomes@tecnico.ulisboa.pt

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. 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.

ACM Classification Keywords
D.2.5 Software Engineering: Testing and Debugging

Author Keywords
OutSystems; software testing; automated testing; automated test generation; low-code.

INTRODUCTION

Building software is an expensive and demanding task. When companies cannot find exactly what they are looking for - usually, a commercial off-the-shelf 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[3]. 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[14]. 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.

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[20]. 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?

Two of those factors are software complexity and the current skill shortage. Determining the complexity in a software system may be considered a subjective term. A study by McNicholl and Magel in 1982[18] 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"[11], 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*.

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 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.

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[16]. Usually, platforms that focus only on the average employee are called no-code platforms, in which no coding is required and the 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.

Popularity of low-code platforms

According to Forrester, a research and advisory firm, there are three key factors that highlight the real value of these platforms[8]:

- **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 *departmental* (code for small, non-critical) applications. Many of the adopters of these platforms have built and use these applications corporate-wide or in multiple departments.

- **Contribute to Application Development and Delivery move to public clouds** According to recent studies[9], 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.

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[13] 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 life cycle 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.

Motivation

According to the World Quality Report 2017-18[6], in 2017 companies allocated 26% of their total IT budget to quality assurance (QA) processes. This value, despite being considerable, means a five percent decrease (31%) from 2016 and a nine percent decrease (35%) from 2015. In terms of the overall cost, 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[21] 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*[17]. It is stated that an issue introduced in the

requirements and design phase could cost from 1 to 100 orders of magnitude 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.

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?

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 that is easy to use and that allows an easy understanding of the current state of the system, following Nielsen's Heuristics[19].
- 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[1].

[1]Application Objects are elements of the application and are used as a measure for licensing purposes.

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. 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.

Despite several attempts, the work presented 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.

OUTSYSTEMS PLATFORM

The OutSystems platform is a low-code solution for the complete application development life cycle. 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 analyze 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.

Product overview

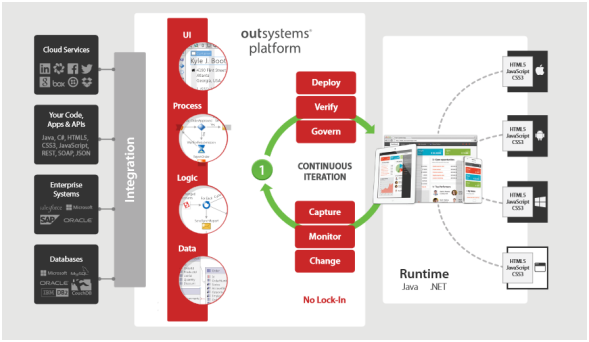

Figure 1. OutSystems platform high-level architecture. Source: OutSystems.com.

Internally, the OutSystems platform provides environments, tools and components that streamline software development, covering all life cycle 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.

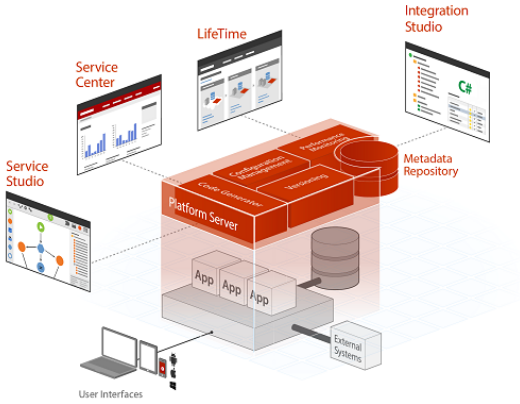

Figure 2. The various components of the platform server in the overall OutSystems platform architecture. Source: OutSystems.com.

When the development is finished and the application is ready for the next step of the life cycle, 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 modeling 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.

Architectural principles

OutSystems uses a pattern called 4-Layer Canvas (4LC) which helps the architect to design a Service-Oriented Architecture 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.

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, Business Process Technology 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.

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 CRUD operations or any other business logic, like calculating a currency exchange rate or a loan interest rate.

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.

Domain-Specific Languages (DSL)

Martin Fowler defines a DSL as "*a computer programming language of limited expressiveness focused on a particular domain*"[10]. 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 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 for defining and building DSLs.

The OutSystems Modeling Language 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[2] that generates the final code. This parser is also able to generate XML files - which are used in this work.

[2]The inner working of this parser is unknown since it's intellectual property of OutSystems and was not disclosed.

RELATED WORK

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. But how to test actions in the OutSystems platform?

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 among 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.

Control Flow Graph 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 **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.

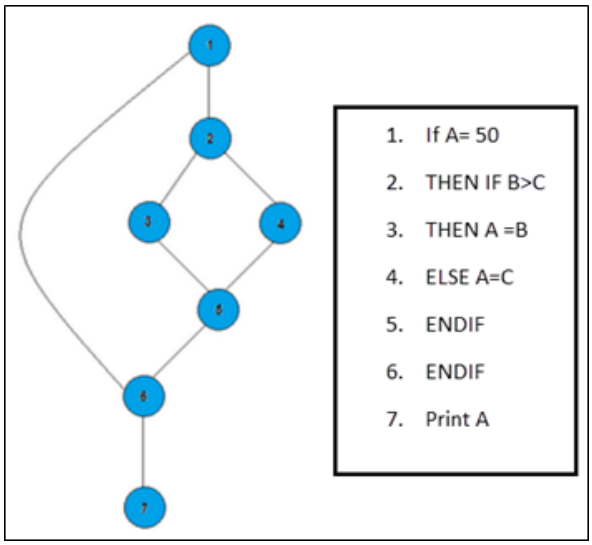

Figure 3. Control flow graph representation. Source: Guru99.com.

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.

Method scope test design patterns

There are several patterns to design method scope tests although only the two most relevant will be presented. Each one applies to a specific purpose implemented by the method under test (MUT).

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.

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.

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.

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[15]. It is not a recent technique, with articles proposing it in 1975[4] but has always faced dificulties 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[1] and, while *interpreting* a given application, it might not be possible to control environment interactions, causing consistency problems - KLEE[5] and Otter[22] are some tools used to mitigate issues related to environment interactions. Other challenges are memory handle, path explosion and constraint solving[2].

Constraint Solving

In order to understand exactly which values of input data are required so a given branch can be execised, 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[7]. An approach for automatically generate test data using constraing solving techniques was proposed by Gotlieb, et al.[12] One example of a solver implemented for the .NET platform is the Z3 Theorem Prover.

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.

Unit Testing Framework

The Unit Testing Framework 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 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 mimick 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*.

BDD Framework

The BDD Framework 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.

Test Framework

The Test Framework 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.
- **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 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.

Automator

The Test Automator is a browser (based on Selenium) and unit (based on web services) 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.

SOLUTION

Due to the nature of the OutSystems Modeling Language (OML) file along with restrictions of intellectual property, OutSystems has provided a representation of the OML file contents in XML format.

This XML file contains the same information as the OML file, in which properties of each object are defined as attributes while associated objects (like input and output variables, local variables, nodes and so on) are referenced as child nodes. This representation allows the system to deserialize the XML file

into a usable object and correctly parse the control flow graph for each user action.

The flow begins with the user uploading a valid XML file in which the system, namely the Parsing Engine, will read and produce an output with all relevant user actions along with the identified branches. Once imported, the user is redirected to the test suite list, specifically generated for the uploaded module.

Parsing Engine

The Parsing Engine is responsible by parsing the XML and calculate the branches. Since elements are linked on the XML file through connectors and those connectors point to a unique key, an auxiliary dictionary is used. Once all the deserialized nodes are converted into custom types, the linkage between elements is created and the branches are computed. In order to compute the branches a recursive function is used, with this function "exploding the paths" when the found element is either an *If* condition, a *Switch* condition or a *ForEach* loop.

User Interface

Once the user picks a test suite from the list, a screen listing basic information about the given action along with last execution is presented. Additionally, the user will see the identified branches. For each of those identified branches, the user will have the option to specify the contents for all of the input variables along with the expected results for the output variables. The inclusion of a solver has not been made during this development iteration.

The user will then have the option to execute an *ad hoc* test in which the results will be promptly presented on the screen. In the case everything works as expected, all fields will appear as green. If anything misses, the field will appear with an red warning and the received value is displayed.

Along with the possibility of setting up input and output variables, the user also has the possibility of mocking computed values. Computed values are values which affect a branch condition but are not part of the input parameter list. This allows the repeatability of tests, at least to some extent.

There are three types of objects that it is important to mock: session variables, site properties (global variables) and external calls (database or other actions). The actual mocking is made using the Dynamic Execution Engine, which is the engine responsible to dynamically invoke actions using reflection.

Regarding the mocking functionality, there are some limitations. While for session variables the values can be easily mocked, for site properties the scenario is a bit different. Although the mocking can be achieved, this is not a "pure" mocking *per se* - the actual value of the parameter is changed and concurrency issues can appear. Finally, for external calls (database and other actions), due to the nature of the generated code (static methods), it's not possible to mock the output of those methods. There are some commercial tools that support that but they all require the code to be running on the local machine and not on a server - being the latter the encountered scenario.

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. In order to retrieve the action through C#, the invocation of the Dynamic Execution Engine receives the name of the module along with the name of the action, which is then invoked using reflection.

Mocking

Due to the limitations presented before, only session variables and site properties will be mocked. 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*, as stated before, since the changes to the value of the *Site Property* are persisted between execution calls.

Method invocation and output parsing

By using a temporary list to store the method's signature (input and output parameters, since the latter are passed by reference), an array of objects is created with representations of the input parameters and *null* objects on the output parameters. Due to output parameters being passed by reference, the *InvokeMethod* method of the action's type (in C#) also requires a *ParameterModified* array. This array only contains boolean values, with the value *true* used for referenced parameters.

Once the invocation is successful, the temporary list with the method's signature is used again and the output parameters are extracted from the object array, being mapped to the output structure of the Dynamic Execution Engine method and, finally, with the results being displayed to the user on the user interface.

EVALUATION

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. The procedure for testing 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.

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.

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.

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.

CONCLUSION

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, the achieved results seem to show that this was a success and the developments should continue, ideally with OutSystems support.

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. 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 feature.

REFERENCES

1. Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. *Demand-Driven Compositional Symbolic Execution*. Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381. DOI: http://dx.doi.org/10.1007/978-3-540-78800-3_28
2. Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).
3. Michael Blechar. 2002. Build, Buy and Outsource Decision Factors. (Dec 2002).
4. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT&Mdash;a Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Not.* 10, 6 (April 1975), 234–245. DOI: http://dx.doi.org/10.1145/390016.808445
5. Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.
6. Capgemini and others. 2017. *World Quality Report 2017-18*. Technical Report. Capgemini.
7. LEONARDO DE MOURA and NIKOLAJ BJØRNER. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (2011), 69 – 77.
8. Forrester. *The Forrester Wave™: Low-Code Development Platforms For AD&D Pros, Q4 2017*. Technical Report. Forrester.
9. Forrester. *The Public Cloud Services Market Will Grow Rapidly To $236 Billion In 2020*. Technical Report. Forrester.
10. Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.
11. Robert L. Glass. 2002. Sorting out Software Complexity. *Commun. ACM* 45, 11 (Nov. 2002), 19–21. DOI: http://dx.doi.org/10.1145/581571.581584
12. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. 1998. Automatic Test Data Generation Using Constraint Solving Techniques. *SIGSOFT Softw. Eng. Notes* 23, 2 (March 1998), 53–62. DOI: http://dx.doi.org/10.1145/271775.271790
13. Jon Idle. 2016. Rapid Response. (Nov 2016).
14. Mark Keil, Duane Truex, and Richard Mixon. 1995. Effects of sunk cost and project completion on information technology project escalation. 42 (12 1995), 372 – 381.
15. James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. DOI: http://dx.doi.org/10.1145/360248.360252
16. ROB MARVIN. 2017. Five Top Low-Code Development Platforms. *PC Magazine* (2017), 69 – 76.
17. Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.
18. Daniel G. McNicholl and Ken Magel. 1982. The Subjective Nature of Programming Complexity. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems (CHI '82)*. ACM, New York, NY, USA, 229–234. DOI: http://dx.doi.org/10.1145/800049.801785
19. Jakob Nielsen. 1994. Enhancing the Explanatory Power of Usability Heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*. ACM, New York, NY, USA, 152–158. DOI: http://dx.doi.org/10.1145/191666.191729
20. The Standish Group. 1995. Chaos Report. (1995).
21. Ara C. Trembly. 2002. Software Bugs Cost Billions Annually. *National Underwriter / Life & Health Financial Services* 106, 31 (2002), 43.
22. Jonathan Turpie, Elnatan Reisner, Jeffrey Foster, and Michael Hicks. 2011. *MultiOtter: Multiprocess Symbolic Execution*. Technical Report. University of Maryland.