



Mariana Araújo Cabeda

Bachelor in Computer Science

Automated Test Generation Based on an Applicational Model

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Miguel Carlos Pacheco Afonso Goulão,
Assistant Professor, Faculdade de Ciências
e Tecnologia da Universidade Nova de Lisboa

Co-adviser: Pedro Lema Santos, Quality Owner,
OutSystems

Examination Committee

Chairperson: Doutor António Maria L. C. Alarcão Ravara
Rapporteur: Doutor João Carlos Pascoal de Faria



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Automated Test Generation Based on an Applicational Model

Copyright © Mariana Araújo Cabeda, Faculty of Sciences and Technology, NOVA University Lisbon.

The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

ACKNOWLEDGEMENTS

I would like to start off by thanking the Faculty of Sciences and Technology from the New University of Lisbon and, in particular, the Informatics Department, for over the course of these last five years, giving me the fundamental tools that will now allow me to start my professional career. Thank you to OutSystems for providing a scholarship for this dissertation.

I definitely need to thank my advisers, Miguel Goulão and Pedro Santos, for mentoring me through this work. This dissertation is a result of your efforts as well, so thank you.

Thank you to everyone at OutSystems for all the support presented. I've felt like part of the team since day one and had a blast going to work every day. The way everyone is always willing to help is like no other and I've learned so much by working with all of you.

Also, thank you to my friends and colleagues, in particular to my fellow thesis companions, Giuliano and Miguel, for the morning coffees and the after lunch snooker games, this has been an intensive work year but we also found time to have some fun and cut the stress off a bit.

Last, but most importantly, I must thank my parents. I would need much more than this one page to thank you, so I'll summarize. Thank you for all the incredible sacrifices you made in order to provide me with the choice to be who I wanted, setting up no boundaries and no limits for what I could achieve, teaching me about independence, hard work and to never be afraid of anything. I will forever be grateful to you. To my grandmother, who I know would be incredibly proud of this moment. To my dog Tobias, who has been a source of happiness and destruction in our house. And also, I suppose, for no particular reason, to my brother.

“The most exciting phrase to hear in science, the one that heralds discoveries, is not ‘Eureka!’ but ‘Now that’s funny. . .’”

– Isaac Asimov

ABSTRACT

Context: As testing is an extremely costly and time-consuming process, tools to automatically generate test cases have been proposed throughout the literature. OutSystems provides a software development environment where with the aid of the visual OutSystems language, developers can create their applications in an agile form, thus improving their productivity.

Problem: As OutSystems aims at accelerating software development, automating the test case generation activity would bring great value to their clients.

Objectives: The main objectives of this work are to: develop an algorithm that generates, automatically, test cases for OutSystems applications and evaluates the coverage they provide to the code, according to a set of criteria.

Methods: The OutSystems language is represented as a graph to which developers can then add pieces of code by dragging nodes to the screen and connecting them to the graph. The methodology applied in this work consists in traversing these graphs with depth and breadth-first search algorithms, employing a boundary-value analysis to identify the test inputs and a cause-effect graphing to reduce the number of redundant inputs generated. To evaluate these test inputs, coverage criteria regarding the control flow of data are analysed according to node, branch, condition, modified condition-decision and multiple condition coverage.

Results: This tool is able to generate test inputs that cover 100% of reachable code and the methodologies employed help greatly in reducing the inputs generated, as well as displaying a minimum set of test inputs with which the developer is already able to cover all traversable code. Usability tests also yield very optimistic feedback from users.

Conclusions: This work's objectives were fully met, seen as we have a running tool able to act upon a subset of the OutSystems applicational model. This work provides crucial information for assessing the quality of OutSystems applications, with value for OutSystems developers, in the form of efficiency and visibility.

Keywords: Software testing, Software test automation, Software test coverage, OutSystems, Visual Programming Language, OutSystems applicational model.

RESUMO

Contexto: Uma vez que testar *software* é um processo extremamente dispendioso e demorado, ferramentas que automatizam a geração de casos de teste têm vindo a ser propostas ao longo da literatura. A OutSystems oferece um ambiente de desenvolvimento de *software* onde, com a ajuda da linguagem visual OutSystems, os *developers* conseguem criar as suas aplicações de forma ágil, melhorando a sua produtividade.

Problema: Sendo que a OutSystems pretende acelerar o desenvolvimento de *software*, automatizar a tarefa de geração de casos de teste teria imenso valor para os seus clientes.

Objetivos: Idealizar e desenvolver um algoritmo que gera, automaticamente, casos de teste para aplicações OutSystems, suportando a avaliação de cobertura que esses testes oferecem ao código, com um conjunto de critérios.

Métodos: A linguagem OutSystems é representada através de um grafo ao qual os *developers* podem adicionar pedaços de código arrastando nós para o ecrã e conectando-os ao grafo. A metodologia utilizada neste trabalho consiste em atravessar estes grafos, com recurso à pesquisa em largura e profundidade, aplicando uma análise dos valores de fronteira para identificar os valores de input, e grafismo de causa efeito de forma a reduzir o número de inputs redundantes gerados. Para avaliar estes valores de input, os critérios de cobertura sobre o fluxo de controlo de dados são avaliados sobre: nós, ramos, condições, condições-decisões e múltiplas condições.

Resultados: Esta ferramenta gera valores de input que cobrem 100% do código alcançável e as metodologias aplicadas reduzem o número de inputs gerados, apresentando ainda um conjunto mínimo com o qual o *developer* é capaz de cobrir todo o código alcançável. Os testes de usabilidade obtiveram feedback positivo por parte dos utilizadores.

Conclusões: Os objetivos deste trabalho foram prontamente alcançados, uma vez que temos uma ferramenta capaz de agir sobre um subconjunto do modelo da OutSystems. Este trabalho oferece informação crucial para avaliar a qualidade de aplicações OutSystems, com valor para os *developers* OutSystems, sob a forma de eficiência e visibilidade.

Palavras-chave: Testes de *Software*, Automatização de Testes, Cobertura de Testes, OutSystems, Linguagem de Programação Visual, Modelo aplicacional da OutSystems.

CONTENTS

List of Figures	xvii
List of Tables	xix
Listings	xxi
Acronyms	xxiii
1 Introduction	1
1.1 Context and description	1
1.2 Motivation	2
1.3 Objectives	3
1.4 Key contributions	3
1.5 Structure	4
2 Background	5
2.1 OutSystems	5
2.1.1 OutSystems platform	5
2.1.2 Language	7
2.1.3 Application testing	10
2.2 Testing overview	11
2.2.1 Testing techniques	11
2.2.2 Testing activities	12
2.2.3 Testing levels	12
2.2.4 Test design techniques	13
2.2.5 Manual and automatic testing	15
2.3 Test automation	15
2.4 Coverage criteria	17
2.4.1 Control flow coverage	17
2.4.2 Data flow coverage	19
2.4.3 Summary	22
2.5 Testing over graphs	23
2.5.1 Introduction to graph theory	23

2.5.2	Graph traversal	24
3	Related work	27
3.1	Tools and techniques	27
3.1.1	Code-based testing	27
3.1.2	Model-based testing	29
3.2	Prioritization of test cases	32
3.3	Summary	33
4	Implementation	35
4.1	Algorithm	35
4.1.1	Architecture	37
4.1.2	The test object	37
4.1.3	Data types and expressions	38
4.1.4	Graph traversal	40
4.1.5	Process nodes	41
4.1.6	Coverage evaluation	55
4.1.7	Expected output	61
4.1.8	Warnings evaluation	62
4.1.9	Test case prioritization	64
4.1.10	Optimizations	65
4.2	PoC with dummy model	65
4.3	Tool applied to the OutSystems model	68
5	Evaluation	71
5.1	Algorithm execution	71
5.2	Usability experiment	74
5.2.1	SUS	74
5.2.2	Results analysis	76
6	Conclusions	79
6.1	Contributions	80
6.2	Future work	80
	Bibliography	83
A	Detailed test results	93
A.1	Algorithm execution	93
A.2	Usability experiment	95
A.3	Graph and questions	95
A.3.1	Graph A	95
A.3.2	Graph B	97

B Documents referenced	99
C VL/HCC paper	107
I OS Language Overview	111
II Test Automation Tools	117
II.1 Tools	117
II.1.1 Proprietary software tools	117
II.1.2 Open source software tools	118

LIST OF FIGURES

2.1	Overview of the main components of the OutSystems platform and the architecture of the Platform Server [52].	6
2.2	OutSystems platform overview [76].	8
2.3	Example of a code trace from Service Studio.	8
2.4	Different testing levels [9].	13
2.5	Subsumption relations among control flow coverage criteria [5].	20
2.6	Subsumption relations among coverage criteria regarding uses and defs [85].	22
2.7	BFS execution example.	25
2.8	DFS execution example.	26
4.1	Example of a graph procedure in OutSystems to be traversed over the course of this chapter.	36
4.2	Structure of the developed algorithm.	37
4.3	OutSystems language nodes representing the different variable types.	38
4.4	Graph traversal shows a depth-first behaviour from the standpoint of each singular thread.	40
4.5	Graph traversal shows a breadth-first behaviour from the standpoint of the entire algorithm where the different colours represent multiple threads.	41
4.6	Traversal starts processing the <i>Start</i> node.	42
4.7	Processing an <i>Assign</i> node.	44
4.8	Processing an <i>If</i> node.	47
4.9	Graph after processing the <i>If</i> node.	48
4.10	Processing a <i>Switch</i> node.	51
4.11	Graph after processing the <i>Switch</i> node.	51
4.12	Graph after processing the node of identifier 5.	52
4.13	Processing a <i>Unsupported</i> node.	53
4.14	The graph and algorithm's state after traversal of node with identifier 7.	53
4.15	The graph and algorithm's state after traversal of node with identifier 9.	54
4.16	The graph and algorithm's state while processing the last couple nodes and after the end of traversal of this graph.	55
4.17	Final test objects after graph traversal.	56

4.18	Graph characteristics: set of all nodes, branches, simple conditions and decisions found in the graph.	56
4.19	Nodes covered by each of the test objects generated by this algorithm.	57
4.20	Branches covered by each of the test objects generated by this algorithm.	58
4.21	Test cases per condition.	59
4.22	Truth tables for each decision, showing if there is a test object that is able to vouch for said entry.	60
4.23	Test cases per decision.	60
4.24	Dead code and paths covered by each test object.	63
4.25	The order by which the code is covered if test cases from each path are executed sequentially from left to right.	66
4.26	XML file snippet from the graph being used as example.	66
4.27	Structure of this PoC.	67
4.28	Proof of concept (PoC) interface screenshot.	68
4.29	Structure of the tool implemented for ServiceStudio.	69
4.30	Screenshot of ServiceStudio where the command to call the tool can be seen.	69
4.31	Tool window with results from this example. a) shows the initial state of the window and b) after expanding all fields.	70
5.1	Scatter plot comparing the amount of top test cases presented against the total test cases generated by this tool.	72
5.2	Percentage represented by the top test cases over the total amount of test cases generated.	73
5.3	Scatter plot comparing the amount of total test cases generated for this tool against what would be generated if not for cause-effect graphing.	73
5.4	Percentage of correct answers for both methods.	75
5.5	SUS distribution.	77
A.1	Top test cases results.	93
A.2	Total test cases generated.	94
A.3	Total test cases generated without cause-effect graphing.	94
A.4	Graph A.	96
A.5	Graph B.	97
I.1	Processes tab in Service Studio.	111
I.2	Interface tab in Service Studio.	112
I.3	Logic tab in Service Studio.	113
I.4	Data tab in Service Studio.	116

LIST OF TABLES

2.1	List of nodes to be analysed for the test case generation with a simple description and the number of incoming and outgoing nodes.	9
2.2	Overview of all coverage criteria defined in this chapter.	23
3.1	Quick overview of the work presented in this chapter.	34
4.1	Expected values for the output variable.	62
4.2	Values used to identify the test object who covers more code and will be the first presented to the developer.	65
4.3	Results from the next step of prioritization.	65
5.1	Risk evaluation according to cyclomatic complexity [41].	72
5.2	For each question on the usability tests, the correct answer rate for both the original method (manual analysis) and for the tool.	75
5.3	SUS meaning [7, 13].	76
5.4	Mean SUS answer for each question.	76
5.5	SUS descriptive statistics.	76
A.1	Detailed results obtained for each tested graph.	95
B.1	List of documents utilized in the making of this report.	99
II.1	List of proprietary software test automation tools [92].	117
II.2	List of some of the popular open sources software test automation tools [92].	118

LISTINGS

1	Breadth-First algorithm [98]	24
2	Depth-First algorithm [98]	26
3	Process data types	39
5	Process <i>Start</i> node	42
6	Process <i>Assign</i>	43
8	Auxiliary functions	46
9	Process <i>Switch</i>	49
10	Process <i>Unsupported</i>	52
11	Process <i>End</i>	54

ACRONYMS

BDD	Behavior-driven development.
BFS	Breadth-first search.
BPT	Business Process Technology.
DFS	Depth-first search.
FAQ	Frequently Asked Questions.
JAXB	Java Architecture for XML Binding.
JSON	JavaScript Object Notation.
JSP	JavaServer Pages.
OCL	Object Constrained Language.
PoC	Proof of concept.
SDG	Sequence Diagram Graph.
SUS	System Usability Scale.
TDD	Test-driven development.
TPLs	Textual Programming Languages.
UI	User Interface.
UML	Unified Modeling Language.
VL/HCC	IEEE Symposium on Visual Languages and Human-Centric Computing.
VPLs	Visual Programming Languages.
XML	eXtensible Markup Language.

INTRODUCTION

This chapter will focus on introducing this thesis, starting with a description and contextualization of the problem along with its motivations, followed by the objectives and contributions. Lastly, the chapter concludes with an overview regarding the structure of the remaining document.

1.1 Context and description

Software testing is a main process of software development. It is a quality control activity performed during the entire software development life-cycle and also during software maintenance [92]. Two testing approaches that can be taken are manual or automated.

In manual testing, the activities are executed by a human sitting in front of a computer carefully going through the application, trying various usage and input combinations, comparing the results to what the expected behaviour should be, reporting defects. Manual tests are repeated often during development cycles for source code changes and other situations like multiple operating environments and hardware configurations [92].

According to Dustin et al. [24] the definition for software test automation refers to the automation of software testing activities including the development and execution of test scripts, validation of testing requirements, and the use of automated testing tools. One clear beneficial aspect of automation is its ability to run more tests with less time when compared to manual testing, which increases productivity.

In this context, there are already a number of tools that allow automated testing over [Textual Programming Languages \(TPLs\)](#) (a list of some of those tools can be consulted in [Annex II](#)), but the same variety does not apply to [Visual Programming Languages \(VPLs\)](#).

VPLs refer to any programming language that lets users create programs by manipulating the program elements graphically rather than by specifying them in a text editor

of source code [48]. With advantages including making programming more accessible, in particular, to reduce the difficulties that beginners face when they start programming, pure VPLs also come with their own set of struggles, particularly when problems demand more flexibility than visual programming could offer.

With this in mind, tools such as OutSystems use a pragmatic mix of both visual and text-based programming, allowing developers to create software visually by drawing interaction flows, UIs and the relationships between objects, but also supplementing it with hand-written code where that's the best thing to do. This model is well suited to the needs of modern software development. Low-code tools reduce the complexity of software development bringing us to a world where a single developer can create rich and complex systems in an agile¹ way, without the need to learn all the underlying technologies [88].

When confronted with the task of testing, the users developing in OutSystems have access to some tools that automate the process of test case execution but, in order to expedite the testing process, there is still a need for a tool that automizes the generation of the test cases themselves.

1.2 Motivation

A program is tested in order to add some value to it. This value comes in the form of quality and reliability, meaning that errors can be found and afterwards removed. This way, a program is tested, not only to show its behaviour but also to pinpoint and find as many errors as possible. Thus, there should be an initial assumption that the program contains errors and then test it [61].

Therefore, a valid definition for testing is [61]:

*Testing is the process of executing a program
with the intent of finding bugs.*

As humans, we tend to be goal-oriented, so properly defining our goals has a major psychological effect [61]. If our goal is to prove that a program has no errors, we will steer subconsciously toward this goal, selecting test data that, will probably never cause the program to fail. On the other hand, if our goal is to show that a program is defective, our test data will have a higher probability in successfully finding errors. The latter approach will add more value to the program than the former [61].

A test case that finds a new bug can hardly be considered unsuccessful; rather, it has proven to be a valuable investment. An unsuccessful test case is one that causes a program to produce the correct result without finding any errors [61].

¹Agile is a methodology that anticipates the need for flexibility and applies a level of pragmatism to the delivery of the finished product. The OutSystems Agile Methodology addresses the need for speed and continuous change, delivering applications that truly respond to business needs [81].

Almost two decades ago, Weyuker et al. [105] pointed out that the most skilled software testers used to change jobs within their companies because a career in software testing was not considered advantageous enough for most professionals, raising the question of “*how demotivated has a software tester to be to abandon their career and follow another path in software development process?*”.

A recent survey [90] concerning work-related factors that influence the motivation of software testers shows that they are strongly motivated by a variety of work, creative tasks, recognition for their work and activities that allow them to acquire new knowledge.

Nowadays, there is a high need for quick-paced delivery of features and software to customers, so automating tests is of the utmost importance. One of its several advantages is that it releases the software testers of the tedious task of repeating the same assignment over and over again, freeing up testers to other activities and allowing for a variety in the work as well as opening space for creativity, being these some of the factors concluded in [90] said to improve software testers motivation at work.

As OutSystems aims at rapid application development, the automation of the test case generation activity, based on their applicational model, along with coverage evaluation, will be of great value to developers using OutSystems.

1.3 Objectives

The main objective of this work was to ideate and develop an algorithm to allow the generation of test scripts in an automated manner over the low-code OutSystems language, not including the actual execution of said tests. This language is defined as being a visual programming language, with some accents of textual language whenever that is the best approach, in order to have the conveniences of VPLs, but still be a functional and scalable language.

Another goal was to provide the ability to evaluate the coverage that the proposed solution can provide the code, according to several coverage criteria that will be detailed in the following chapters.

Finally, the viability of implementing this tool on top of the OutSystems main platform, the Service Studio, was analysed, evaluating the solution in terms of performance as the code grows in complexity since this will be the primary challenge as the test cases are expected to grow exponentially.

1.4 Key contributions

At the end of this thesis, there is a running tool, related to a sub-set of the OutSystems applicational model, able to automatically generate the test scripts for the selected code, and this tool is running on top of the OutSystems’ development environment. This work lays the foundation to have test coverage on applications made in OutSystems.

This feature will bring value to people developing applications in OutSystems, both internally, within the company, and also externally, to customers.

Complementing this tool, its implementation is documented in a published paper [15] (see [Appendix B](#)) for the [IEEE Symposium on Visual Languages and Human-Centric Computing \(VL/HCC\)](#) of 2018.

Outside of OutSystems, it is also expected that this work will contribute to further advance the state of the art of automated testing techniques, namely over [VPLs](#).

1.5 Structure

The remainder of this thesis is organized as follows:

- [Chapter 2 - Background](#): here the focus is on the research that was performed, being the main topics the OutSystems platform, testing, automated testing, coverage criteria over graphs and graph traversal algorithms;
- [Chapter 3 - Related work](#): this chapter presents some tools and techniques that relate to the context of this thesis, in this case, focused on the automatic generation of tests;
- [Chapter 4 - Implementation](#): depicts the algorithm conceptualized as well as details the implementation of both the [PoC](#) and the tool applied to the OutSystems model;
- [Chapter 5 - Evaluation](#): presents the results obtained for this work, both by the execution of the tool as well as from usability tests performed;
- [Chapter 6 - Conclusions](#): concludes this dissertation with a quick overview of the work produced as well as identifies what can be the future of the tool hereby presented;
- [Appendix A - Detailed test results](#): further details the results recorded and the structure of the usability tests performed;
- [Appendix B - Documents referenced](#): this appendix depicts all documents referenced throughout the present report, informing the main topics it approaches and the chapters it is referenced on;
- [Appendix C - VL/HCC paper](#): shows the paper submitted and accepted for the [VL/HCC](#) conference of 2018 focused on the work produced by this dissertation;
- [Annex I - Test Automation Tools](#): contains a high-level overview of the OutSystems language;
- [Annex II - Test Automation Tools](#): here are listed some tools that already exist for test automation, both open-source and proprietary.

BACKGROUND

This chapter provides background context related to this dissertation, covering: the OutSystems platform, software testing overview (both manual and automated), coverage criteria applicable over graphs, graph theory and finally, graph traversal algorithms.

2.1 OutSystems

OutSystems is a software company whose primary focus relies on improving the productivity and simplifying the day-to-day life of IT professionals. The OutSystems platform was developed with a strong focus on performance, scalability, and high-availability and can be used to create web and mobile applications. The OutSystems language allows users to develop at a higher abstraction level, without the need to worry over low-level details related to creating and publishing applications [52, 82].

2.1.1 OutSystems platform

Figure 2.1 depicts the main components of the OutSystems platform and the architecture of the Platform Server where the components of the typical 4-tier web application architecture are represented in white, which the OutSystems platform complements with an extra set of services and repositories displayed in red. Service Studio and Integration Studio, the two products which compose OutSystems Development Environment, are desktop tools that interact with the Platform Server via Web services [52].

2.1.1.1 Service Studio

Service Studio is an environment where business developers assemble and change web and mobile business applications using visual models with a drag-and-drop paradigm.

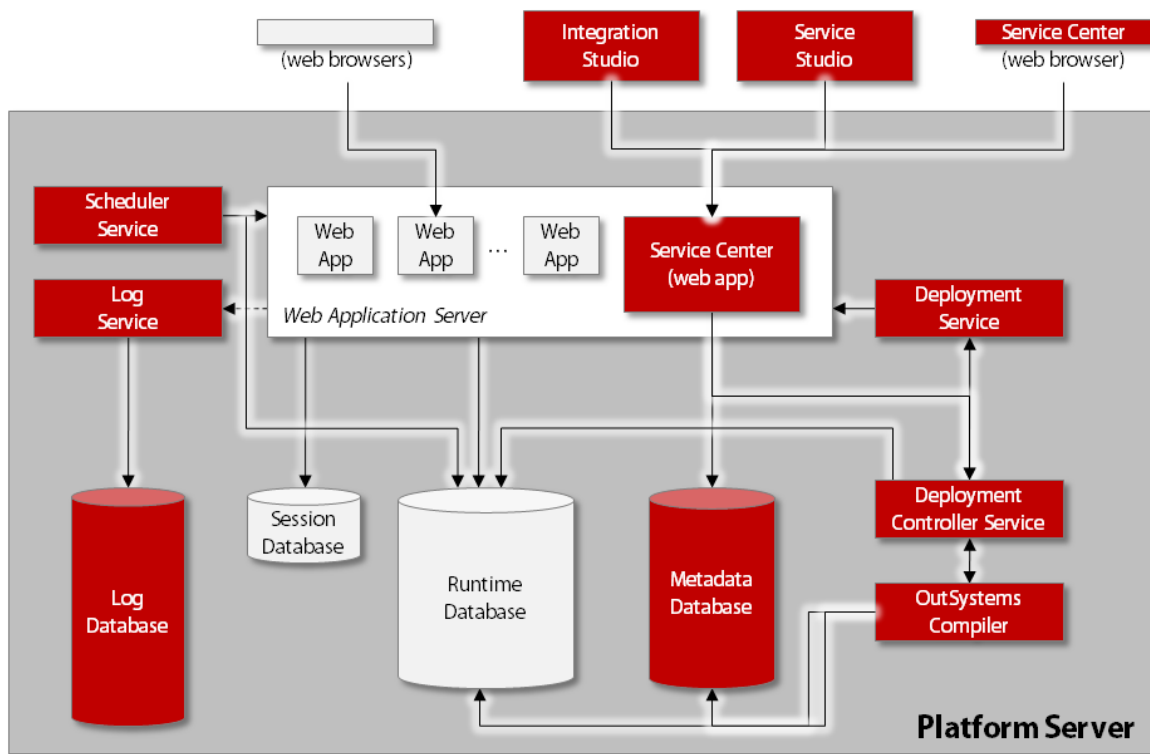


Figure 2.1: Overview of the main components of the OutSystems platform and the architecture of the Platform Server [52].

This tool also enables the modeling of UIs, Business Processes, Business Logic, Databases, Integration Components, SAP BAPIs [91], SOAP and REST Web Services, Security Rules, and Scheduling activities. Service Studio embeds a full-reference checking and self-healing engine that assures error-free, robust change across all application components.

When the developer publishes an application, Service Studio saves a document with the application model and sends it to the Platform Server [52].

2.1.1.2 Integration Studio

In Integration Studio, developers can create components to integrate existing third-party systems, microservices, and databases, or even extend OutSystems with their own code.

After the deployment of these components, they will be available to reuse by all applications built with OutSystems. Developers use Visual Studio to code integration components and can take advantage of existing ASP.NET [58] libraries.

When publishing a component, the development environment compiles it with a standard ASP.NET [58] compiler. The generated DLLs are sent to the Platform Server [76].

2.1.1.3 Platform Server

The Platform Server components take care of all the steps required to generate, build, package, and deploy native ASP.NET [58] and Java web applications on top of a Microsoft

stack, Oracle WebLogic [67] or JBOSS [40], using a set of specialized services [76]:

- *Code generator*: takes the application modeled in the IDE and generates native ASP.NET [58] code, allowing the generation of applications that are optimized for performance, are secure and run on top of standard technologies;
- *Deployment services*: these services deploy the generated ASP.NET [58] application to a standard web application server, ensuring an application is consistently installed on each front-end of the server farm;
- *Application services*: manage the execution of scheduled batch jobs, providing asynchronous logging services to store events like errors, audits performance metrics.

2.1.2 Language

As it has already been briefly mentioned in previous sections, the OutSystems language allows developers to experience the benefits of VPLs but also takes into account the fact that VPLs have a set of disadvantages associated, and the most concerning are [97]:

- *Extensibility*: visual languages allow developers to do a limited set of things easily, but the edge cases are too difficult or even impossible to achieve. Tools should give more power, instead of limiting the developers;
- *Slow code*: every developer who has faced performance problems knows how hard they are to diagnose and overcome. Visual languages can be leaky abstractions, generating slow code which is impossible to optimize.

With all of this in mind, the OutSystems language uses a pragmatic mix of both visual and text-based programming, supplementing the visual with text-written code where that is the best thing to do.

Figure 2.2 shows a visual representation of the OutSystems platform, where from Service Studio (2.1.1.1), the code created passes on to a Code Generator encapsulated in a file. This contains all the graph information and the application logic. This code containing elements which constitute OutSystems' model, is the input to the algorithm meant to be developed during the course of this thesis.

The model for Service Studio comprises of a multitude of components that include a set of UI elements (widgets - text, container, link, etc), processes (Business Process Technology (BPT) - activities, timers, etc), amongst others (themes, entities, etc). Annex I contains a high-level overview of the OutSystems language.

For the course of this work, only the application logic behind client/server actions and events will be considered.

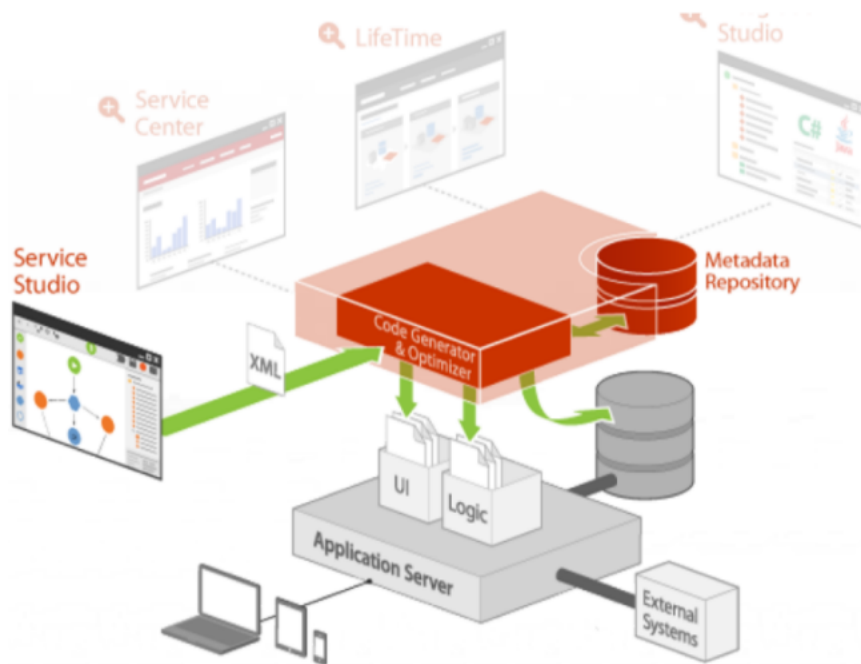


Figure 2.2: OutSystems platform overview [76].

2.1.2.1 Nodes

Figure 2.3 shows a logic flow in the OutSystems language, illustrated by a set of nodes that come together to form a graph representing a given method, function, procedure, etc. These nodes represent blocks of code that the developer can add to the graph as desired.

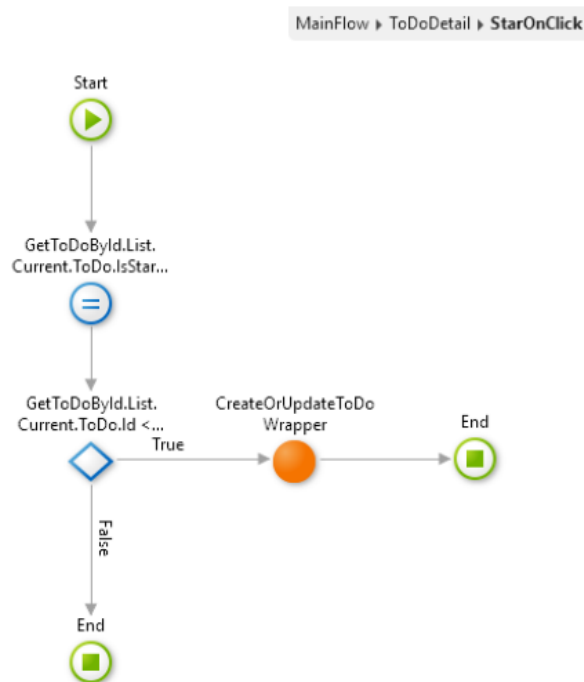
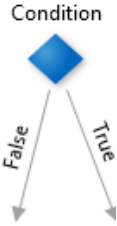
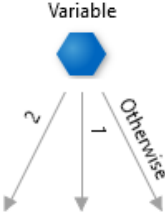




Figure 2.3: Example of a code trace from Service Studio.

Considering the extension and complexity of the model provided by OutSystems, only a subset of the existing nodes will be evaluated in this thesis, as the time allocated for this work would not suffice. Table 2.1 displays a list of the most pertinent information regarding the selected node types: visual representation, behavior description and number of incoming and outgoing branches. Although incomplete, this list comprises a very interesting set of elements which are the most important for exercising decision flows, a starting point for future work.

Table 2.1: List of nodes to be analysed for the test case generation with a simple description and the number of incoming and outgoing nodes.

	Description	In : Out
If	 <p>This node presents the typical behaviour of an if code block, with his Boolean condition and the outgoing flows according to the condition result, weather it is True or False. Contrary to some programming languages where the False branch can be omitted, in OutSystems, both branches are defined.</p>	1:2
Switch	 <p>This switch block analysis each decision and the flow will continue through the decision it verifies as true.</p>	1:N
Assign	 <p>This block assigns a value (or object) to a variable. Each assign node can comprise of multiple assignments</p>	1:1
ServerAction	 <p>This block represents the collapse of another graph that will contain code corresponding to an action to be executed on the server-side of the application.</p>	1:1

2.1.3 Application testing

Due to the nature of visual development and continuous integrity validation built into OutSystems¹, users do not need to worry about the underlying technical challenges, given the abstractions that OutSystems provides, which makes it less error prone [79].

Regardless, testing is a fundamental part of any software. OutSystems' approach is to keep their platform open so it is compatible with the tools developers typically use. This way, testing is integrated in the continuous delivery cycle so there are not losses in productivity [71].

There are some tools available that allow the execution of tests for applications created in OutSystems. Something that is still missing, is a tool that can generate the test cases themselves, offering a coverage analysis and this, what the present dissertation proposes.

2.1.3.1 Unit Testing Framework

The Unit Testing Framework [84] provides a complete framework for implementing, executing and managing unit tests.

Teams find this approach particularly effective for calculation engines and business service components. Having a good set of unit tests for a system can help greatly when it is time to change or refactor a system [79].

2.1.3.2 Behavior Driven Development Framework

Test-driven development (TDD) is an evolutionary approach that requires the writing of automated tests prior to developing functional code in small, rapid iterations [47].

Behavior-driven development (BDD) was originally developed by North [63] in response to issues in TDD. It is focused on defining fine-grained specifications of the behaviour of the targeting system, in a way that they can be automated, allowing developers to focus the creation of tests for the most critical use cases [79]. The aim goal of BDD is to get executable specifications of a system [63, 99].

The BDD Framework [83] is an open source component the developer can adapt to its own needs, but already provides: creation of test scenarios and steps conformant to BDD principles; support for multiple tests in the same page and the final statistics (number of successful/failed tests); amongst other features.

2.1.3.3 Functional, UI and Regression Testing

For functional and regression testing in web applications, OutSystems recommends the use of Selenium [95], but any strategy currently used to test traditional web applications apply as well.

¹Continuous Integration is a software development practice where work is integrated frequently. Each integration is verified by an automated build to detect integration errors as quickly as possible [32]. For this, OutSystems employed a validation engine - TrueChange [78] -, that assures robust and error-free changes across all applications and their modules.

Additionally, there is Test Automator [101], a browser (Selenium-based) and unit (WebService-based) regression testing tool that helps guarantee the quality of solution development by automating the execution of tests over the application.

2.2 Testing overview

Developing a large software system is an extremely complex and error-prone process. A fault might occur at any stage of development and it must be identified and treated as early as possible in order to stop its propagation. Quality engineers must be involved in the development process since the very early stages up beyond the product deployment all the way through maintenance and post-mortem analysis [8].

Testing plays an important role in achieving and assessing the quality of a software product. On the one hand, we are able to improve the quality of the products as we continuously test, find a defect and fix it, all during development. On the other hand, we also assess how good our system is when we perform system-level tests before releasing a product [50].

2.2.1 Testing techniques

The term testing refers to a full range of test techniques, even quite different from one another, and embraces a variety of aims [9].

2.2.1.1 Static techniques

The distinction between static and dynamic techniques depends on whether the software is executed or not. Static techniques are based solely on the (manual or automated) examination of project documentation, of software models and code, and other related information about requirements and design. Thus static techniques can be employed all along development, and their earlier usage is, of course, highly desirable.

Traditional static techniques include [9]:

- *Software inspection*: a very popular software inspection process was created by Michael Fagan and is referred to as the Fagan inspection. It is a structured process of trying to find defects in development documents such as programming code, specifications, designs, and others during various phases of the software development process. Fagan Inspection defines a process as a certain activity with a pre-specified entry and exit criteria and then, inspections can be used to validate if the output of the process complies with the exit criteria specified for the process [28, 29];
- *Algorithm analysis and tracing*: the process in which the complexity of algorithms employed and the worst-case, average-case and probabilistic analysis evaluations can be derived.

2.2.1.2 Dynamic techniques

Dynamic techniques obtain information about a program by observing executions. In contrast to static analysis, which examines a program's text to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examining the running program. While dynamic analysis cannot prove that a program satisfies a property, it can detect violations of properties and also provide useful information to programmers about the behavior of their programs [6, 9].

2.2.2 Testing activities

In order to test a program, one must perform a sequence of testing activities, namely [50]:

1. *Identify an objective to be tested*: this will define the intention, or purpose, of the designing of the test cases to ensure said objective is successfully met by the software;
2. *Select inputs*: the selection of test inputs can be based on the requirements specification of the software, the source code or our expectations. Test inputs are selected by keeping the test objective in mind;
3. *Compute expected outcome*: without running the program, one must already have an understanding of what the expected outcome should be, and that can be done from an overall, high-level understanding of the test objective and the specification of the program under test;
4. *Execute the program*: we must setup the execution environment accordingly and execute the program with the selected inputs, observing the actual outputs;
5. *Analyse the test result*: finally, having executed the test, the last testing activity is to analyse the result, comparing the actual outcome with the expected one and assigning a verdict to the program.

2.2.3 Testing levels

Testing can be performed at different levels involving the complete system, or only parts of it, throughout its lifecycle. A software system goes through several stages of testing before it is actually deployed [9], some of which are characterized next:

1. *Unit level*: here, individual program units, such as procedures, functions, methods or classes are tested, in isolation;
2. *Integration level*: in general terms, integration is the activity of aggregating software pieces to create a larger component. Integration testing aims at testing these larger components to guarantee that the pieces that were tested in isolation can now work together as a whole;

3. *System level*: this level includes a wide spectrum of testing, being a critical phase in a software development process because of the need to meet a tight schedule close to the delivery date, to discover most faults and to verify that fixes are working and have not resulted in new faults;
4. *Acceptance level*: after the completion of the system-level testing, the product is delivered to the customers that perform their own series of tests, based on their expectations for the system. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is the objective of system-testing.

The first three levels of testing are performed by a number of different stakeholders in the development organization, whereas acceptance testing is usually performed by the customers.

There is a type of testing that is usually performed throughout the lifecycle of a system, whenever a component of the system is modified, called regression testing. The main idea is to ensure that the modification did not introduce any new faults in the portion that was not subject to modification.

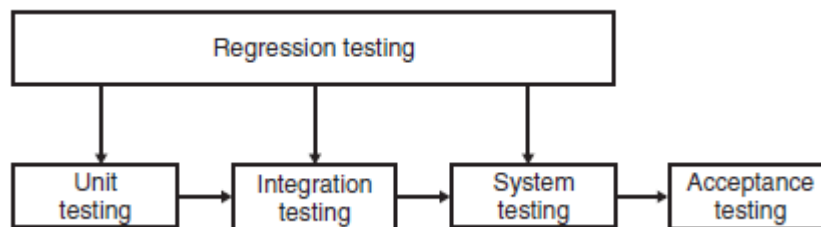


Figure 2.4: Different testing levels [9].

2.2.4 Test design techniques

Three broad concepts in testing, based on the sources of information for test design, are white-box, black-box, and grey-box testing [25].

2.2.4.1 White-box

In white-box or structural testing, the source code for the system is available to the tester and the test cases are selected based on the implementation of the software. The objective here is to execute specific parts of the software, such as specific statements, branches or paths. The expected results are evaluated under a set of coverage criteria, just as the ones depicted in [section 2.4](#).

2.2.4.2 Black-box

In black-box or functional testing, the tester does not have access to the internal details of the software that is thus treated as a black box. The test bases are selected based on the requirements or design specification of the system under test. Functional testing emphasis relies on the external behavior of the software.

Commonly used black-box methodologies are detailed next.

Equivalence Partitioning

Equivalence partitioning is a technique in which the input domain of a problem is divided into a finite number of *equivalence classes* such that it can be reasonably assumed that a test derived from a value of each class is equivalent to a test derived from any other value from the same class.

This technique aims at defining test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed, as some of them would prove to be redundant [25, 61].

Boundary-value analysis

Boundary-value analysis focuses more on testing on boundaries, or where they are chosen. It includes minimum, maximum, just inside/outside boundaries, error values, and typical values [25]. It differs from equivalence partitioning in two aspects [61]:

1. Instead of selecting any element in an equivalence class as being representative, boundary-value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test;
2. Instead of just focusing on the input conditions, test cases are also derived by considering the result space (output equivalence classes).

Cause-effect graphing

In this technique, testing starts by creating a graph and establishing the relationship between effects and its causes. Identity, negation, logic OR and logic AND are the four basic symbols which express the interdependency between cause and effect [25].

Cause-effect graphing has a beneficial side effect in pointing out incompleteness and ambiguities in the specification.

Error guessing

The basic idea behind *error guessing* consists in writing test cases based on possible or error-prone situations. These can be identified both by intuition and experience and include classic cases as the number 0 as input or “none” and “one” when a variable number of inputs or outputs can be present. Another idea is to identify test cases associated with assumptions that the programmer might have made when reading the specification [61].

2.2.4.3 Grey-box

Grey-box testing can be derived from the previous two and is a technique to test the application with limited knowledge of the internal working of an application and also has the knowledge of fundamental aspects of the system.

Commonly employed grey-box methodologies are depicted next.

Matrix Testing

This testing technique starts by defining all the variables that exist in the software. Each variable will have an inherent technical and business risk associated and can be used with different frequency during its' lifecycle. All this information is summarized in tables from which then the design of test cases is derived [25].

Regression Testing

In this technique, if there are new changes made to the software, there is a selection of already executed test cases that are then re-executed in order to check if the change in the previous version has regressed other aspects of the program in the new version [25]. This technique is done to make sure that new code changes do not have undesired side effects over already existing functionalities.

2.2.5 Manual and automatic testing

Testing work can be roughly divided into automatic and manual testing, as follows [34]:

- *Manual testing*: a human tester takes the role of an end user and executes the features of a given software under test to ensure its behaviour is as expected;
- *Automatic testing*: the use of special software to control the execution of tests and the comparison of actual outcomes with predicted outcomes.

2.3 Test automation

It is unproductive to test software manually, since the thousands of scenarios human testers generate are vulnerable to inaccurate results, and manual tests are also slow and difficult to repeat. In addition, a manual approach might not be the most effective in finding certain defects. Therefore, test automation aims to have the actions that enhance the quality of the product done constantly in an effort to make the software as error-free as possible by the time it goes out on the market.

These actions may include: development of test cases; selection of inputs and computation of outputs; evaluations after the scenarios are run, among others. Thanks to test automation, total product quality and efficiency can be greatly increased in the major software development processes, with versions delivered much faster, less staff assigned for manual testing and fewer software errors [27].

But, although we may see test automation as a solution to decrease testing costs and to reduce cycle time in software development, if it is not applied at the right time and context with the appropriate approach, it might fail [34].

In test automation, we have four basic components: testers, (test automation) tools, test cases, and the system under test. Test engineers interact with the test automation tools and develop test cases which are then executed using the chosen test automation tool. The tests exercise the system under test and the tool provides test reports for humans to interpret. Even though the introduction of test automation often increases the cost of creating tests, the cost of re-running them decreases [42].

Test automation provides us with a great amount of advantages [22, 92]:

- *Saves time and money*: tests have to be repeated during development cycles to ensure software quality and so every time source code is modified, such tests must be repeated to ensure no bugs were accidentally introduced into the code. Repeating the tests manually is costly and time-consuming, as testers could be focused on other tasks such as dealing with more complex features. After their creation, automated tests can be run multiple times with decreased cost and at a much faster pace compared to manual tests as it can reduce the time to run said tests from days to hours, and saving time translates directly into saving money;
- *Improves consistency of test results*: even the most expert tester will make mistakes during monotonous manual testing. Automated tests perform the same steps precisely every time they are executed and never forget to record detailed results;
- *Provides the ability to perform tests that are very difficult to execute manually thus increasing test coverage*: automated software testing tools can look inside an application and see memory contents, data tables, file contents, and internal program states to determine if the product is behaving as expected, easily executing thousands of different complex test cases during every test run providing coverage that is impossible with manual tests. These tools can also simulate tens, hundreds or even thousands of virtual users interacting with the network, software and web applications, something that was also extremely difficult to do with manual testing;
- *Team morale improves*: as there is no more need to spend time executing repetitive tasks, automated software testing gives teams time to spend on more challenging and rewarding projects, allowing team members to improve their skill sets and confidence and, in turn, they pass those gains on to their organization.

There are also some risks/difficulties associated with the automation of software testing activities [22, 92]:

- *Tools cost and learning curve*: testing automation lays heavily on top of the tools that are used. A careful consideration needs to be made in whether to purchase a licence for a proprietary software tool, or adapting a pre-existing open source one

or even, developing a completely different type of tool. Along with this, learning how to manipulate the chosen tool also introduces an extra cost. A list with both proprietary and open source tools is available in [Annex II](#);

- *Unrealistic expectations from the tool*: having unrealistic expectations from the tool is a risk that may lead to schedule and cost seepage;
- *Maintenance of the automation scripts*: maintaining the scripts may prove to be expensive. For a large project, the volume of test data might be high and it requires a good structure to maintain all the test data.

2.4 Coverage criteria

Software test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage.

In terms of coverage over graphs, it is usual to divide these criteria into two types: *control flow coverage criteria* and *data flow coverage criteria*. *Control flow* criteria uses the control structure of a program to develop the test cases. *Data flow* criteria are based on the flow of data through the software artifact represented by the graph.

A general definition for graph coverage can be enunciated as follows: given a set TR of test requirements for a graph criterion C , a test set T satisfies C on graph G if, and only if, for every test requirement tr in TR , there is at least one test path p in $path(T)$ such that p meets tr [5].

2.4.1 Control flow coverage

Control flow coverage, or structural coverage as it is also called in literature, refer to white-box methodologies and they include: node, branch and path coverage [5, 50, 104].

Node coverage

Node coverage - also referred to statement or segment coverage -, refers to executing individual program statements and observing the outcome. We say we have 100% node coverage if all the statements have been executed at least once in all of our test cases combined. Complete node coverage is the weakest coverage criterion in program testing.

Before moving on to the next few coverage criteria, it might be useful to define and distinguish the concepts of condition and decision:

- Condition is a simple Boolean expression, meaning that it cannot be broken down into simpler Boolean expressions. For example, the following expressions are conditions:
 - 1: $A == B$
 - 2: C
- Decision is a Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. For example, the following expressions are decisions:
 - 1: $A \ \&\& \ B$
 - 2: $(A \ \&\& \ B) \ || \ C$

Decision Coverage

A decision - also known as edge or branch - is an outgoing edge from a node. Having full decision coverage means selecting a number of paths such that every decision is included in at least one path.

Condition coverage

Full condition coverage states that each possible outcome of each condition occurs at least in one test case, meaning that all conditions are, at least one time, true and false.

Modified condition-decision coverage

After the introduction of decision and condition coverage, modified condition-decision coverage refers that every condition in a decision that has been shown to independently affect that decision's outcome must be tested by varying just that condition while holding fixed all other possible conditions.

Multiple condition coverage

In multiple condition coverage, or conditional combination coverage, all possible combinations of conditions should be evaluated.

Multiple condition coverage implies branch, condition, condition-decision and modified condition-decision coverage, which will have an exponential blow-up ($2^{\text{number_of_conditions}}$) and some combinations may be infeasible.

Path Coverage

Here, all possible paths in the graph are taken into account. However, a program may contain a large or even infinite number of paths. A very common example is when we have cycles within the graph, where complete path coverage is not feasible since each cycle iteration will originate a new path.

In order to try to get as close as possible to complete path coverage, there are also other path coverage criteria based definitions. The most common include [5]:

- **Prime Path Coverage**

Before defining what a prime path is, there is a need to introduce the definition of a simple path.

A path from n_i to n_j is said to be simple if no node appears more than once in the path, with the exception of the first and last nodes that may be identical. This means that simple paths have no internal loops, although the entire path itself may be a loop. These paths are very useful given that any path can be created by composing simple paths.

Even small programs may have a very large number of simple paths. For a simple path coverage criterion we would like to avoid enumerating the entire set of simple paths and, instead, list only the maximal length simple paths, to which we call a prime path [5].

A path from n_i to n_j is called a prime path if it is a simple path and does not appear as a proper sub-path of any other simple path.

Therefore, the prime path coverage criterion is defined as having all prime paths show up in at least one test case. This way, we can say we have reached 100% prime path coverage.

- **Simple Round Trip Coverage**

- A round trip path is a prime path of nonzero length that starts and ends at the same node. In simple round trip coverage, it is required that the test cases contain, at least, one round-trip path for each reachable node in the graph that begins and ends a round-trip path [5].

- **Complete Round Trip Coverage**

- In complete round trip coverage, it is required that all round-trip paths for each reachable node in the graph are tested.

- **Specified Path Coverage**

In specified path coverage, instead of requiring all paths, it is only considered a specified set of paths. For example, these paths might be given by a customer in the form of usage scenarios.

Figure 2.5 presents the subsumption relations between the control flow coverage criteria previously introduced, i.e., indicates which criteria are assumed to be covered once other criterion is reached. For example, when decision coverage is reached, it is implied that node coverage is also covered, and so on.

2.4.2 Data flow coverage

Data flow analysis focuses on how variables are bound to values, and how these variables are to be used. Just as one would not feel confident about a program without executing

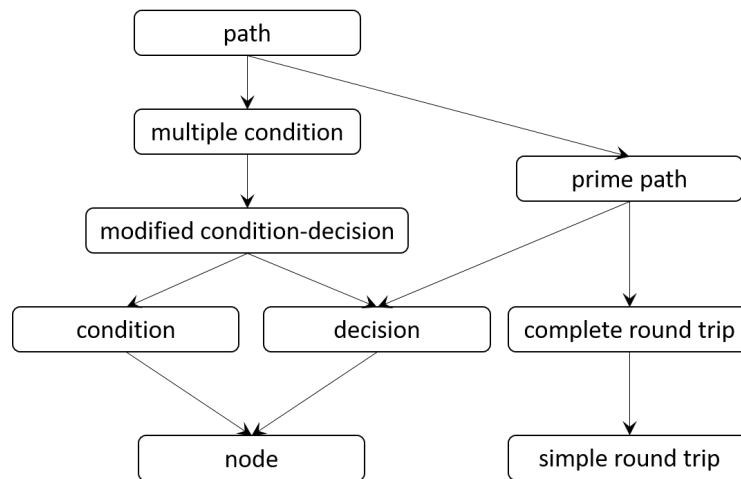


Figure 2.5: Subsumption relations among control flow coverage criteria [5].

every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation. Traversing all paths does not guarantee that all errors will be detected. We must be aware that path selection criteria cannot ensure that a set of test data capable of uncovering all errors will be chosen, as it was demonstrated in [85, 86].

2.4.2.1 The Rapps and Weyuker Family of Criteria

Both Rapps and Weyuker have defined a family of coverage criteria based on definitions and uses of variables. The first reports of said criteria can be consulted in [85, 86].

Their idea was that in order to test a software correctly, we should focus on the flows of data values and try to ensure that the values created at one point in the program are not only created but also used correctly. This is done by focusing on *definitions* and *uses* of values. A *definition* (*def*) is a location where a value for a variable is stored into memory. A *use* is a location where a variable's value is accessed. These data flow testing criteria use the fact that values are carried from *defs* to *uses*. These are called the *du-pairs* - also known as *definition-uses*, *def-use*, and *du* associations and the idea is to exercise *du-pairs* in various ways [5].

Each variable occurrence is classified as being a definitional, computation-use, or predicate-use occurrence. Those are referred to as *def*, *c-use* and *p-use*, respectively. Since *defs* have already been defined, the definitions of both *c* and *p-use* are as follow:

- *p-uses* occurs when a variable is used to evaluate wheater a predicate is true or false;
- *c-uses* occurs when a variable is used to compute the value of other variables, or output values.

Here are some examples [85, 86]:

- $y = f(x_1, \dots, x_n) \Rightarrow c\text{-use}$ of variables x_1, \dots, x_n and def of y ;
- $read\ x_1, \dots, x_n \Rightarrow def$ of variables x_1, \dots, x_n ;
- $print\ x_1, \dots, x_n \Rightarrow c\text{-uses}$ of variables x_1, \dots, x_n ;
- $if\ p(x_1, \dots, x_n)\ then\ goto\ m \Rightarrow p\text{-uses}$ of variables x_1, \dots, x_n .

An important concept when discussing data flow criteria is that a *def* of a variable may or may not reach a particular *use*. The most obvious reason would be because no path goes from the definition to any use. But, a more subtle reason is that the variable's value may be changed by another *def* before it reaches the *use*. Thus, a path from l_i to l_j is said to be *def-clear* with respect to variable v if no location between l_i and l_j changes the value [85].

Next, the definitions of All-Defs, All-P-Uses, All-C-Uses/Some-P-Uses, All-P-Uses/Some-C-Uses, All-Uses, and All-du-Paths Coverage will be introduced.

All-Defs Coverage

This criterion is satisfied if, for every variable defined, there is a path included in at least one of our test cases, that goes from the definition to a use. In a more informal way, each *def* reaches at least one *use*.

All-P-Uses Coverage

Here, we need to have, in at least one of our test cases, a path from every variable definition to the set of all its *p-uses*.

All-C-Uses/Some-P-Uses Coverage

To accomplish all-c-uses/some-p-uses coverage it is required that every *c-use* of all variables defined must be included in some path of our test cases. If there is no such *c-use*, then some *p-use* of the definition of the variable must be included. Thus to fulfill this criterion, every definition which is ever used must have some *use* included in the paths of the test cases, with the *c-uses* particularly emphasized.

All-P-Uses/Some-C-Uses Coverage

Similar to the previous criteria, only here we put the emphasis on the *p-uses*, and therefore, every *p-use* of all variables defined must be included in some path of our test cases. If there is no *p-use*, then some *c-use* must be included.

All-Uses Coverage

All-uses criterion is satisfied if, for every definition of a variable, there is a path in our test cases that includes a *def-clear* path from the definition to all its uses, both *c-uses* and

p-uses. In a more informal manner, this one requires that each *def* reaches all its possible *uses*.

All-du-Paths Coverage

This one requires that each *def* reaches all possible uses through all possible *du-paths*.

Figure 2.6 shows the subsumption relations, this time, between the criteria introduced by Rapps and Weyuker.

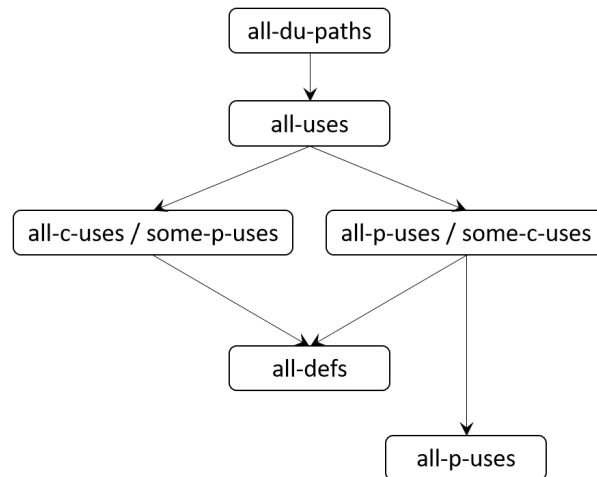


Figure 2.6: Subsumption relations among coverage criteria regarding uses and defs [85].

2.4.2.2 Ntafos' Required k-Tuples Criteria

Ntafos uses data flow information to overcome the shortcomings of using control flow information alone to select paths defining a class of path selection criteria, based on data flow analysis, the *Required k-Tuples*. These criteria require that a path set cover chains of alternating definitions and uses, called *k-dr* interactions. A *k-dr* interaction propagates information along a sub-path that is called an interaction sub-path for the *k-dr* interaction [3, 19].

2.4.2.3 Laski's and Korel's Criteria

Laski and Korel define criteria that emphasize the fact that a given node may contain uses of several different variables, and that each use may be reached by several definitions occurring at different nodes. These criteria are concerned with selecting sub-paths along which the various combinations of definitions reach the node and they are referred to as the Context Coverage and Ordered Context Coverage criterion [19].

2.4.3 Summary

To recapitulate, all the coverage criteria introduced in this section are named in Table 2.2.

Table 2.2: Overview of all coverage criteria defined in this chapter.

Control Flow	Data Flow		
	Rapps & Weyuker	Ntafos	Laski & Korel
Node Branch Condition Modified condition-decision Multiple condition Path Prime Path Simple Round Trip Complete Round Trip Specified Path	All-Defs All-P-Uses All-C-Uses/Some-P-Uses All-P-Uses/Some-C-Uses All-Uses All-du-Paths	Required k-tuples	Context Ordered Context

2.5 Testing over graphs

This section starts with an introduction to graph theory, followed by some graph traversal algorithms.

2.5.1 Introduction to graph theory

As it was already covered in [section 2.1](#), the OutSystems programming language is a visual language that represents the generated code visually through graphs, hence why we need a more in-depth understanding on how we can cover and traverse graphs.

A graph $G=(V, E)$ is defined by a set of vertices V , and contains a set of edges E . Several fundamental properties of graphs impact the choice of data structures used to represent them and the algorithms available to analyse them. The first step in any graph problem is to classify the graphs that are being dealt with according to a set of properties, including [98]:

- *Undirected vs Directed*: a graph is said to be undirected if edge $(x, y) \in E$ implies that $(y, x) \in E$, that is, there is no direction imposed over the flow in the edges;
- *Weighted vs Unweighted*: each edge (or vertex) in a weighted graph is assigned a numerical value, implying that said edge or vertex has a certain cost or gain associated with its traversal. In unweighted graphs there is no cost distinction between the various edges and vertices;
- *Cyclic vs Acyclic*: this property refers to the existence (or not) of cycles in the graph;
- *Implicit vs Explicit*: certain graphs are not explicitly constructed and then traversed, but build as they are used. Because there is no need to store the entire graph, it is often easier to work with an implicit graph than explicitly construct it prior to analysis;

- *Labeled vs Unlabeled*: each vertex is assigned a unique name or identifier in a labeled graph to distinguish it from all others. In unlabeled graphs, no such distinctions are made;
- *Connected vs Disconnected*: a graph is said to be connected if there is a path between every pair of nodes. In a connected graph there are no unreachable nodes.

2.5.2 Graph traversal

There are two primary graph traversal algorithms: [Breadth-first search \(BFS\)](#) and [Depth-first search \(DFS\)](#). The breadth-first algorithm (2.5.2.1), along with the depth-first (2.5.2.2), are probably the most simplistic uninformed search procedures. For certain problems, it makes no difference which one is used, but in others the distinction is crucial. The difference between these two algorithms lays in the order in which they explore vertices. This order depends completely upon the container data structure used to store the *discovered* vertices.

2.5.2.1 Breadth-First

Breadth-first's traversal proceeds uniformly outward from the start node, exploring the neighbour nodes layer-wise, after which moving towards the next-level neighbour nodes [98].

One interesting guarantee [BFS](#) provides, is that when a goal node is found, the path expanded is of minimal length to the goal. A disadvantage of this solution is that it requires the generation and storage of a tree whose size is exponential in the depth of the shallowest goal node [62].

A pseudo-code for this algorithm can be seen in [Algorithm 1](#).

Algorithm 1 Breadth-First algorithm [98]

```

1: procedure BFS( $G, s$ )
2:   for each vertex  $u \in V[G] - s$  do
3:      $state[u] = \text{"undiscovered"}$ 
4:      $p[u] = nil$ , i.e. no parent is in the BFS tree
5:    $state[s] = \text{"discovered"}$ 
6:    $p[s] = nil$ 
7:    $Q = s$ 
8:   while  $Q \neq \emptyset$  do
9:      $u = \text{dequeue}[Q]$ 
10:    process vertex  $u$  as desired
11:    for each  $v \in Adj[u]$  do
12:      process edge  $(u, v)$  as desired
13:      if  $state[v] = \text{"undiscovered"}$  then
14:         $state[v] = \text{"discovered"}$ 
15:         $p[v] = u$ 
16:        enqueue[ $Q, v$ ]
17:     $state[u] = \text{"processed"}$ 

```

Figure 2.7 illustrates a run of the BFS algorithm over a graph, with the insight of the associated data structures and its modification alongside the running of the algorithm.

An example applying the BFS algorithm over a directed graph is shown in Figure 2.7.

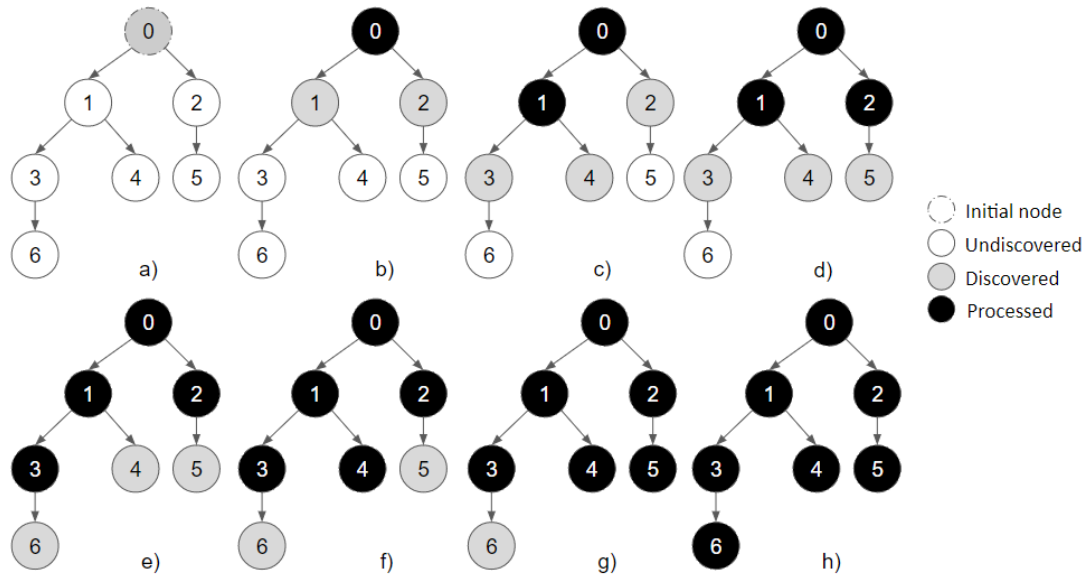


Figure 2.7: BFS execution example.

The *Uniform-cost search* [23], proposed by Dijkstra, is a variant of BFS in which all the branches in the graph have an associated cost and nodes are expanded outward from the starting node along the “contours” of equal cost rather than along contours of equal depth. If the cost of all arcs in the graph are identical, then uniform-cost is the same as breadth-first search [62].

2.5.2.2 Depth-First

DFS is one of the most versatile sequential algorithm techniques known for solving graph problems. Tarjan [100] and Hopcroft and Tarjan [43] first developed depth-first algorithms for connected and biconnected components of undirected graphs, and strong components of directed graphs.

DFS generates the successors of a node, just one at a time and, as soon as a successor is generated, one of its successors is generated and so on. To prevent the search process from running away toward nodes of unbounded depth from the start node, a depth bound can be set, making it so that no successor is generated whose depth is greater than the depth bound (it is presumed that not all nodes lie beyond the depth bound) [62].

This algorithm only requires part of the search tree consisting of the path currently being explored and traces at the yet fully expanded nodes along that path to be saved. The memory requirements are thus linear in depth bound. A disadvantage of DFS is that when a goal is found, we are not guaranteed to have a minimal length path. Another

problem is that we may have to explore a large part of the search space even to find a shallow goal if it is the only goal and a descendant of a shallow node expanded late [62].

A valid pseudo-code for this algorithm can be seen in Algorithm 2.

Algorithm 2 Depth-First algorithm [98]

```

1: procedure DFS( $G, u$ )
2:    $state[u] = \text{"discovered"}$ 
3:   process vertex  $u$  if desired
4:   for each  $v \in Adj[u]$  do
5:     process edge  $(u,v)$  if desired
6:     if  $state[v] = \text{"undiscovered"}$  then
7:        $p[v] = u$ 
8:       DFS( $G, v$ )
9:    $state[u] = \text{"processed"}$ 

```

Figure 2.8 illustrates a run of DFS over a directed graph.

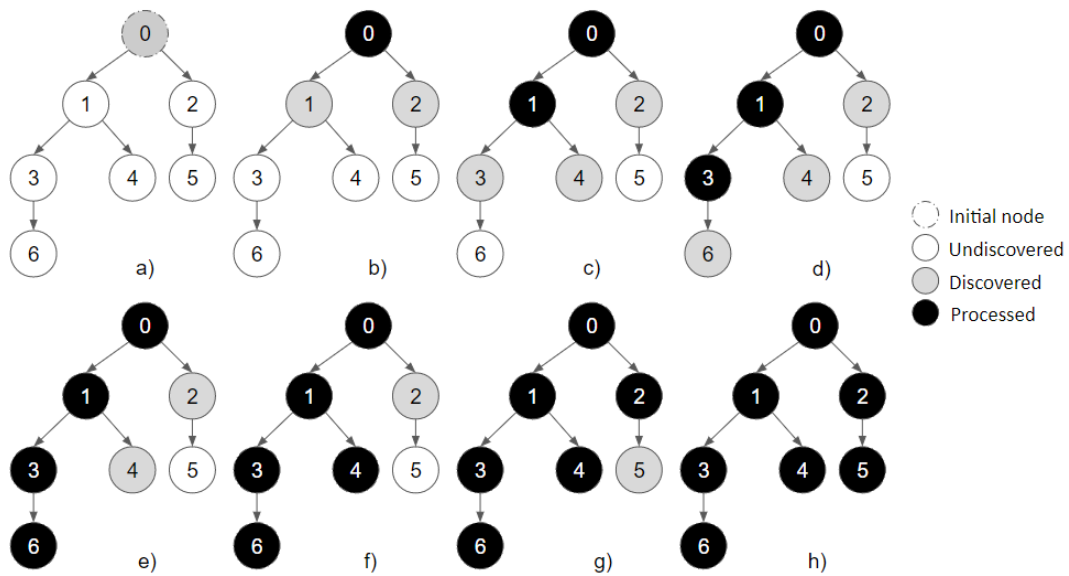


Figure 2.8: DFS execution example.

The computational complexity of DFS was investigated by John Reif [87] that considered the complexity of computing the depth-first search, given a graph and a source. A decision version of the problem (testing whether some vertex u occurs before some vertex v in this order) is P-complete, meaning that it is “a nightmare for parallel processing”.

RELATED WORK

In the context of this thesis, the research upon the related work already documented will be focused on the topic of automatic test generation techniques and tools. The prioritization of the test cases will also be a subject of interest in this chapter.

3.1 Tools and techniques

There are two main approaches used to generate test cases automatically: from source code, and from requirements and design specification. These two introduce the following topics of code-based testing (3.1.1) and model-based testing (3.1.2) [16].

3.1.1 Code-based testing

Code-based (or white-box) testing derives the test cases directly from the system source code. Some of its advantages include [25]:

1. It is able to reveal errors in hidden code;
2. Maximum coverage can be obtained while writing test scenarios.

There are also some disadvantages [21]:

1. Is not capable of executing the behavioural aspects of the system;
2. Is not suitable for component-based software development, because the source code may not be available to the developer.

Some tools and techniques developed for code-based testing will be introduced next.

3.1.1.1 TestEra (2001)

TestEra [54] automates the generation of test data and evaluates correctness criteria for Java programs using Alloy [4], a specification language to describe structures, to formulate the invariants of inputs and properties for a Java program. As a result, it produces Java inputs that infringe on any of the criteria evaluated.

This tool starts by specifying the inputs for the Java program through Alloy. Then, Alloy Analyser [46] generates all non-isomorphic¹ instances for that specification. These represent the test cases for the program. After their execution, the output registered is traced back to Alloy, and the Analyser checks the input and output against the given criteria.

TestEra focuses on exposing as many errors as possible, without producing false positives. For this, completeness may be compromised and errors missed, but the ones reported are clear counterexamples to violated properties.

3.1.1.2 DART (Directed Automated Random Testing) (2005)

DART [35] starts by automatically extracting the interface of a C program. From the interfaces, a test driver is generated and random testing is applied. Finally, dynamic analysis is performed in order to evaluate how the program behaves under random testing and new test inputs are generated directing the execution of the program over alternative paths.

This approach can be performed completely automated on any C program that compiles and it exposes errors such as program crashes, assertion violations, and non-termination. It also addresses the main limitation hampering unit testing, namely the need to write test driver and harness code to simulate the external environment of a software application [35].

3.1.1.3 Boshernitsan's et al. tool for testing based on software agitation (2006)

Boshernitsan et al developed and documented a tool based on *software agitation* [11]. This is a unit testing technique that generates test cases dynamically, integrates sets of input data against the test cases, thus “agitating” them, and then, analyses the results.

The authors implemented software agitation in a testing tool called Agitator [1] and the technique employed is: (1) write code → (2) agitate → (3) display observations → (4) review results → (5) define tests.

This tool is able to produce test cases that achieve up to 80.2% code-coverage percentage. The coverage metrics used represent statement and condition coverage [11].

¹A relation between objects is classified as isomorphic when they show equality in their composition. Non-isomorphism classifies objects that are structurally different [55].

3.1.1.4 Fraser's whole test suite generation using genetic algorithms (2013)

Fraser proposes a model where *whole test suites* are generated with the objective of reaching all defined goals while maintaining the size of these suites as small as possible [33]. This can be accomplished through the use of genetic algorithms in the task of searching. This approach advantages include the fact that the number of infeasible targets in the code will not affect its efficacy.

This methodology was then evaluated by the author against the traditional approach that targets each testing goal independently over open source libraries and the results showed that it was able to obtain up to 188 times the branch coverage, with up to 62% smaller test suites [33].

3.1.2 Model-based testing

Model-based (or black-box) testing derives test cases from an abstract model of software, including formal specifications and semi-formal design descriptions such as [Unified Modeling Language \(UML\)](#) diagrams. Automatically generating test cases directly from design models has several benefits which include [26, 94, 103]:

1. As requirements evolve, time is saved by only updating the model compared to updating the entire test suite, as they tend to be much larger;
2. The improper specification of requirements is a major source of system problems. Model-based testing helps exposing these requirement issues;
3. As the test cases are generated before the code, developers have the possibility to use them as they develop the code, thus reducing the iterations between development and testing, further saving resources.

Disadvantages [26, 103]:

1. Its effectiveness depends on the expertise and experience of those creating the model and selecting the test selection criteria;
2. If requirements change and the model is not updated, the tests will yield a significant amount of errors;
3. Models of any non-trivial software functionality can grow beyond manageable levels. Almost all other model-based tasks, such as model maintenance, non-random test generation and achieving coverage criteria, are affected in this scenario.

Model-based testing is gaining its popularity in both research and in industry. As systems are increasing in complexity, more systems perform mission-critical functions, and dependability requirements such as safety, reliability, availability, and security are vital to the users of these systems [94].

Therefore, researchers have used the analysis and design models such as [UML](#) for test case generation. These models are very popular because [UML](#) is a solution of standardization and utilization of design methodologies [21].

Tools and techniques for model-based testing developed within the past decade follow next.

3.1.2.1 QuickCheck (2002)

QuickCheck [17, 18] is a random testing tool² where specifications are used directly for both the test case generation and as oracle. Here, the tester provides a set of properties and then, a large number of test cases are randomly generated in an attempt to find some that contradict these properties. Recent versions of QuickCheck automatically reduce test cases that fail, seeing as having “noise” in the input tests makes it harder to understand the failure, reporting a set of “minimal” test inputs. The original implementation of QuickCheck was concerned with Haskell [39] programs; later re-implementations exist for a number of languages, including Java, C, C++, amongst others [18].

3.1.2.2 Korat (2002)

Korat [12, 59] automatically tests a program based on Java predicates [64]. For that, it generates all non-isomorphic³ inputs, up to a given small size, for which the predicate is valid, meticulously searching the bounded input space of the predicate but doing so efficiently, by monitoring the predicate’s executions, trimming large portions of the search space. For this constraint-based generation of structurally complex test inputs for Java programs, Korat takes: a predicate that specifies the desired structural integrity constraints and an initialization that bounds the desired test input size.

Korat then enforces the method on each test case and uses the method’s post-condition as a test oracle to validate each output.

3.1.2.3 Sarma’s et al. automatic test case generation from [UML](#) sequence diagram (2007)

Sarma et al. [93] proposed a method that generates test cases from [UML](#) [102] sequence diagrams. The approach consists in transforming a sequence diagram into a sequence diagram graph and supplying the graph’s nodes with the necessary information to compose test vectors. This graph is then traversed to generate test cases based on a coverage criteria and a fault model. These are suitable for system testing and to detect interaction and scenario faults. Scenario faults happen when, for a given operation scenario,

²The basis of random testing consists in identifying the input domain, selecting test points independently from this domain and then to execute the program on these inputs (that constitute a random test set). The results are then compared to the program specification [38].

³See footnote 1.

the sequence of messages does not pursue the desired path due to inaccurate condition evaluation, anomalous termination, etc.

3.1.2.4 Sawant and Shah's automatic generation of test cases from UML models (2011)

Sawant and Shah [94] presented a technique for generation of test cases based on UML [102] diagrams such as use case, class and sequence diagrams, and then transforming it into a **Sequence Diagram Graph (SDG)** [93], where each node stores the necessary information for the test case generation. A data dictionary is presented in the form of **Object Constrained Language (OCL)** [37]. The UML diagrams are created with the aid of tools such as MagicDraw [53] and Rational Rose [44] and then exported to **eXtensible Markup Language (XML)** [20] format. The XML file is then parsed to provide the extraction of different nodes of the graph and the generation of all sets of scenarios from start to end nodes. These sets of scenarios along with the use case template and OCL data dictionary are traversed using **BFS (2.5.2.1)** for the generation of test cases.

3.1.2.5 Dalai's et al. test case generation for concurrent object-oriented systems using combinational UML models (2012)

Dalai et al. [21] proposed an approach to generate test cases for concurrent object-oriented software systems using the combinational features of UML [102] sequence and activity diagrams in order to obtain higher coverage and to provide fault detection capability. The diagrams are first converted into graphs and then a sequence-activity graph is generated by combining the features of both these diagrams. The graph is then traversed with the following methodology: wherever a fork node is encountered, **BFS (2.5.2.1)** is applied, while **DFS (2.5.2.2)** is used for the remainder nodes. Activity path coverage criterion is used for the generation of test cases.

3.1.2.6 Chouhan's et al. test case generation based on activity diagram for mobile application (2012)

Chouhan et al. [16], developed a test case generation model based on UML activity diagrams for mobile applications, starting with the construction of an intermediate table called the activity dependency table that contains the following columns: name of the activity, dependency nodes, in degree value, dependent nodes and out degree values. The table automatically generates a directed graph called activity dependency graph that will then be examined with the **DFS (2.5.2.2)** algorithm in order to extract all the possible test cases. These generated test cases should go through all the branches in the activity diagram.

As for coverage criteria, this approach applies a hybrid coverage criterion as the combination of branch, full predicate and basic path coverage criteria along with cyclomatic complexity criterion. Cyclomatic complexity was introduced by McCabe in 1976 [56]

as a metric to measure program flow-of-control. The cyclomatic complexity is based on determining the number of linearly independent paths in a program module, suggesting that the complexity increases with this number and reliability decreases. This complexity can be computed using the number of edges (E) and the number of nodes (N) and the formula used to calculate it [51]:

$$CC = E - N + 2$$

3.2 Prioritization of test cases

Due to the magnitude that some test suites have, systems require a large amount of time and resources to execute all test cases. One of the main concerns in software testing lies on how to execute these tests in the most efficient way and that is the main objective of test case prioritization [60].

Test case prioritization means setting priority to every test case from a test case suit and executing the test cases in descending order. This technique helps in minimizing testing time and increases testing efficiency, which reduces cost and provides earlier identification of high-risk defects, seen as higher risk test cases are executed first, also providing better resource utilization [96].

Different techniques that prioritize test cases aim at maximizing some defined function or objective [89]. Some of those techniques are presented next.

Seth and Anand's prioritization of test case scenarios from UML sequence diagrams (2012)

Seth and Anand [96] proposed a technique for prioritization of test cases that will first generate test cases from UML sequence diagrams and prioritize them based on: (a) prioritization on depending upon depth: the more objects are covered, the higher the priority will be, (b) prioritization depending on the number of parameters: the more parameters involved, the higher the priority, (c) prioritization depending on code coverage and lastly, (d) the combination of all.

Then, priority is set according to any of the previous options and the test cases with higher priority are scheduled to be tested first.

Fernandez-Sanz and Misra's prioritization of test cases from UML activity diagrams (2012)

Fernandez-Sanz and Misra proposed an approach for the generation of a complete test case suite from UML activity diagrams [30]. The authors prioritized said test cases by using the risk each test case represents over the software as the main parameter. This risk value is defined through several measures, including the propability of a certain function being used and its importancy for the system. This information is then gathered and each test case is assigned a value by which they are prioritized and the test cases considered to present higher risk to the software are tested first [30].

Results obtained by the authors showed that over 70 software professionals recognized the advantages of this method [30].

3.3 Summary

All of the techniques previously introduced provide a good overview over the ideas that have been presented in the research field over the past couple decades.

Both code-based and model-based approaches have the ability to influence the work produced by this dissertation given that the OutSystems applicational model represents both a model and the actual source code for the applications. More specifically, the ability to reach coverage goals - for the specific context of this work, the ability to be able to reach all traversable graph with the generated test inputs -, as well as, from model-based approaches, the use of both **BFS** and **DFS** algorithms to traverse graphs. These ideas were indeed concretized in the implementation of both the **PoC** and the tool produced (see [chapter 4](#)).

For the tools presented, **Korat** in particular, has some similarities with the tool hereby developed, specifically in the idea of reducing the test case load by generating all non-isomorphic inputs. In a parallel manner, the tool developed is applying cause-effect graphing, where test inputs are only generated if they verify the context into which they came to be, i.e., if a test input is identified in a trace of code that it would never reach, it is discarded.

Furthermore, the ideas for prioritization of test cases inspire the work behind the combination of multiple factors such as in [96] with the intent of trying to display first the most pertinent test cases to the developer.

[Table 3.1](#) presents a quick summary of each tool and technique.

Table 3.1: Quick overview of the work presented in this chapter.

	Work	Year	Summary
Code-based	TestEra [54]	2001	Given a set of correctness criteria, it generates test inputs that violates them.
	DART [35]	2005	Identifies the external interfaces for the program and a random test driver is generated simulating the most general environment visible to the program.
	[11]	2006	Tool based on software agitation that combines the results of research in test-input generation and dynamic invariant detection.
	[33]	2013	Obtains more coverage with smaller test suites by applying generic algorithms.
Model-based	QuickCheck [17, 18]	2002	Randomly generates test cases that contradicts the properties that the functions should fulfill.
	Korat [12, 59]	2002	Automatically generates all non-isomorphic inputs for which the predicates return true.
	[93]	2007	Uses UML sequence diagrams from which a graph is generated that is then traversed to generate the test cases.
	[94]	2011	Uses UML diagrams to generate a graph that is then traversed using BFS to generate the test cases.
	[21]	2012	Uses UML sequence and activity diagrams to generate a graph that is traversed with a mix algorithm with both BFS and DFS in order to generate the test cases.
	[16]	2012	Uses UML activity diagrams to construct a table that generates a graph that is then traversed with DFS algorithm to extract all the test cases. Uses a hybrid coverage criteria of: branch, full predicate, basic path and cyclomatic complexity.
Prioritization	[96]	2012	Prioritizes the test cases depending on depth, the number of parameters, code coverage or a combination.
	[30]	2012	Prioritizes the test cases in order of software risk.

IMPLEMENTATION

This chapter describes the algorithm developed during this dissertation and presents both the PoC and the tool implemented. Its organization is as follows:

- **Section 4.1 - Algorithm:** explains all details associated with the development of the algorithm that was implemented in the following tools;
- **Section 4.2 - PoC with dummy model:** this PoC was implemented resorting to a simplified version of the OutSystems applicational model;
- **Section 4.3 - Tool applied to the OutSystems model:** shows the algorithm implemented directly over the OutSystems development environment, the *Service Studio*, this time using the OutSystems model.

4.1 Algorithm

Figure 4.1 represents a graph that will be traversed step-by-step over the course of this chapter in order to explain the strategies employed in the algorithm to generate all the different input combinations.

This is a simple graph comprising of the nodes this algorithm supports: *Assign*, *If* and *Switch*, some of the most important regarding the execution of logic flows for server and client actions in OutSystems applications.

The remainder of this section is organized as follows:

- **Subsection 4.1.1 - Architecture:** shows the architecture of the algorithm;
- **Subsection 4.1.2 - The test object:** description of this fundamental object that aids in the graph traversal;

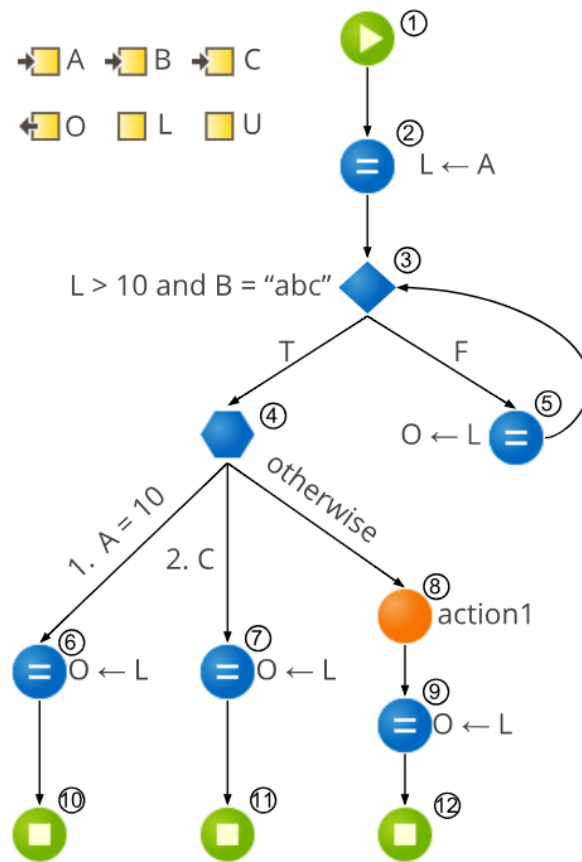


Figure 4.1: Example of a graph procedure in OutSystems to be traversed over the course of this chapter.

- [Subsection 4.1.3 - Data types and expressions](#): details how the values to be tested are selected according to the data types and expressions supported;
- [Subsection 4.1.4 - Graph traversal](#): methodology employed to traverse the graph;
- [Subsection 4.1.5 - Process nodes](#): explains how each type of node is processed;
- [Subsection 4.1.6 - Coverage evaluation](#): describes the strategies employed in order to evaluate the coverage criteria applied in this algorithm;
- [Subsection 4.1.7 - Expected output](#): the expected output values for a procedure are evaluated and presented to the developer as well;
- [Subsection 4.1.8 - Warnings evaluation](#): shows how warnings are computed;
- [Subsection 4.1.9 - Test case prioritization](#): the test cases are presented prioritized, i.e., the ones considered to bring more risk to the code are presented first;
- [Subsection 4.1.10 - Optimizations](#): introduces some optimizations that have been made to the algorithm in order to save resources.

4.1.1 Architecture

The algorithm starts by receiving the model as input to be evaluated and the graph is then traversed according to both the graph traversal algorithms of breadth (subsubsection 2.5.2.1) and depth-first (subsubsection 2.5.2.2) search. The different input combinations are generated through the mechanisms of boundary-value analysis (section 2.2.4.2) and cause-effect graphing (section 2.2.4.2). This last mechanism is applied in order to reduce the number of redundant test cases that, otherwise, would be generated. After the traversal is finished, the produced test coverage is calculated according to node, branch, condition, modified condition-decision and multiple condition coverage (subsection 2.4.1). Finally, all results are presented to the user in a graphical user interface.

Figure 4.2 shows a component diagram representing the architecture of the algorithm.

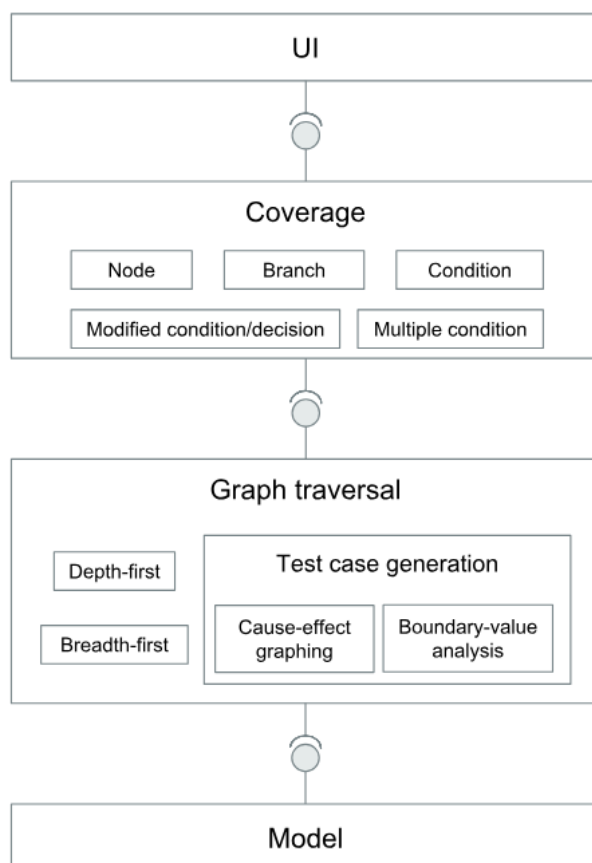


Figure 4.2: Structure of the developed algorithm.

4.1.2 The test object

A test object is a fundamental component in this algorithm. It will follow a specific path during graph traversal and record a set of interesting information, such as the different input combinations that would follow said path, the traversed decisions and whether they were met or not, and all the branches (and consequently, all the nodes) that path

covers. All of this up until an *End* node is reached, from which point the test object is then marked as final.

This is an essential object that not only aids in the graph traversal process but also contains the information that allows for the computation of warnings and coverage obtained during the traversal. This way, the graph only needs to be traversed one time and by the end of it, all information can be determined by looking at each generated test object.

The distinction between input variables/parameters and others (local and output variables/parameters) must be made. This results from the fact that only input variables can be instrumented from outside of the scope of a function, while the remainder results from either computations using other variables, or default values. Figure 4.3 shows the icons that the OutSystems language uses to distinguish these different types of variables.



Figure 4.3: OutSystems language nodes representing the different variable types.

In summary, the information stored in the test object is:

- (1) *Test cases*: set of all the input combinations generated that would follow the path traversed by this object;
- (2) *Dependencies*: a map connecting a variable to other variables it depends on. This is specially important for non-input variables in order to be able to test them by manipulating the input variables they depend upon;
- (3) *Branches*: a set that will track the branches which comprise the traversed path. From here, the nodes covered by this path are also available;
- (4) *Decisions passed*: contains the decisions that this test object covered during its path traversal and was met by its test cases;
- (5) *Decisions failed*: the decisions that this test object covered but were not met by its test cases.

Summarily, while traversing the graph, for each different path encountered, a new test object will be generated and once the traversal is finished, the total number of test objects will correspond to the number of independent paths found. Each test object will contain multiple input combinations and the set of all those combinations will define all the input combinations generated needed to cover all reachable code according to boundary-value analysis (section 2.2.4.2) and cause-effect graphing (section 2.2.4.2).

4.1.3 Data types and expressions

Three basic data types are currently taken into account: Integers, Booleans and Strings. Thus far, no other data types are being considered for a matter of complexity.

Algorithm 3 represents the pseudocode which defines the values to be tested for each data type, according to the following definitions:

- For Integers, the values to be evaluated follow the methodology of boundary-value analysis (see section 2.2.4.2). Therefore, the values to be tested are the ones just inside/outside boundaries and the boundary itself. Example: For the condition: $var1 > 5$, the generated values for $var1$ would be: 4, 5 and 6. These values are not influenced by the operator, only by the value, which in this example was 5;
- For Booleans, both the True and False values will be evaluated;
- For Strings, the values to be tested will consist of the String itself, a different String (randomly generated and different from the original) and the empty String.

Algorithm 3 Process data types

```

1: function PROCESSINTEGER(int)
2:   prev ← {int - 1}
3:   next ← {int + 1}
4:   return {prev, int, next}
5:
6: function PROCESSBOOLEAN( )
7:   return {True, False}
8:
9: function PROCESSSTRING(string)
10:  random ← generateDiffRandomString(string)
11:  return {string, random, ""}

```

In this work, complex expressions that involve comparing variables to other variables are not yet being considered. This comes from the fact that, by following the current generation mechanism, for example, if we have condition $I > I2$ (both integer variables), the values $\{I2 - 1, I2, I2 + 1\}$ would be generated. To be able to express these values into actual input combinations for variable I , it would be required that $I2$ would have values defined by the end of this algorithm's execution. And that might not happen.

A way to overcome this issue could be by also employing the error guessing mechanism (see section 2.2.4.2) where the key values for each data type that normally bring in problems would also be tested.

In short, this algorithm currently supports three of the most simple data types (Integer, Boolean and String) as well as simple expressions, such as the following:

Attributions:

1. $I = I2$
2. $S = \text{"abc"}$

Decisions:

1. $I > 24$ or $J = 3$
2. $S = \text{"abc"}$ and $T = \text{"hij"}$

4.1.4 Graph traversal

The traversal begins at the *Start* node, progressively following outgoing flows until either an *End* node is reached or a cycle is detected.

During traversal, whenever a new node is found, after being processed according to its own node type (see subsection 4.1.5), its outgoing flows are added to a shared queue. This shared queue is composed by blocks containing both a test object (subsection 4.1.2) and the node that represents the outgoing flow. From this queue, a defined set of threads can then pull blocks and continue the traversal. This way, the work is parallelised in an effort to reduce processing time.

With this approach, both the breadth-first (subsubsection 2.5.2.1) and depth-first (subsubsection 2.5.2.2) algorithms are employed. From a single thread standpoint, the work is performed in a depth-first manner. From the graph's general point of view, all threads traverse the branches simultaneously, mimicking a breath-first approach. Figure 4.4 and Figure 4.5 represent the technique described.

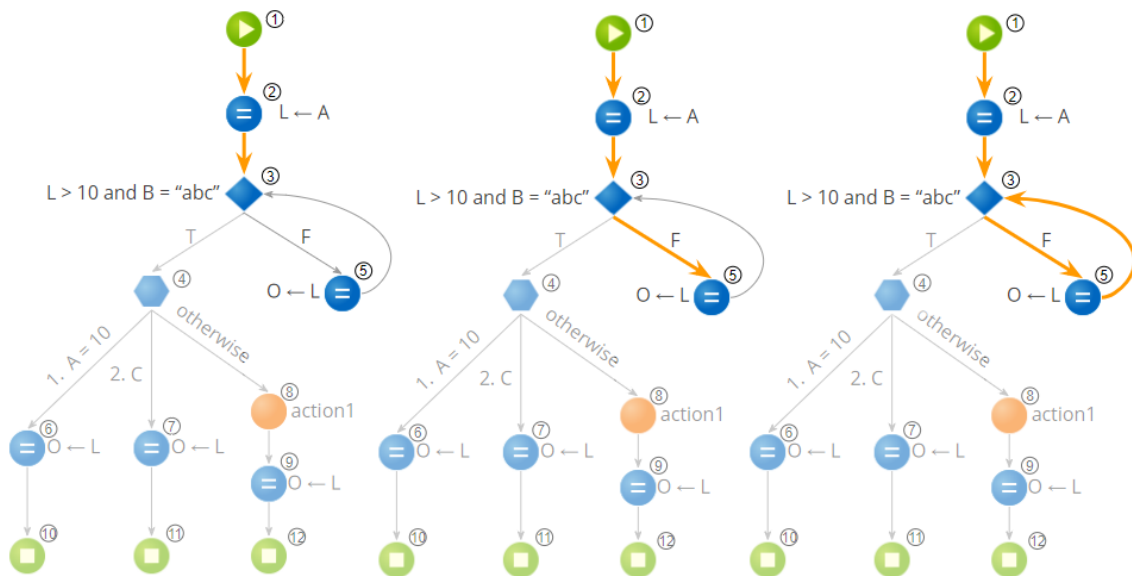


Figure 4.4: Graph traversal shows a depth-first behaviour from the standpoint of each singular thread.

The traversal is marked as finalized when after all threads finish their current execution and the shared queue is empty, thus not providing any more work for any thread. After this, the results are compiled and sent back to the developer.

Algorithm 4 shows the pseudocode for the graph traversal and it is divided into two distinct processes:

1. The graph traversal itself that consists in launching a set of threads and initializing the shared queue from where each thread will pull work and start/continue traversal;

2. The code executed by each thread to process the different node types.

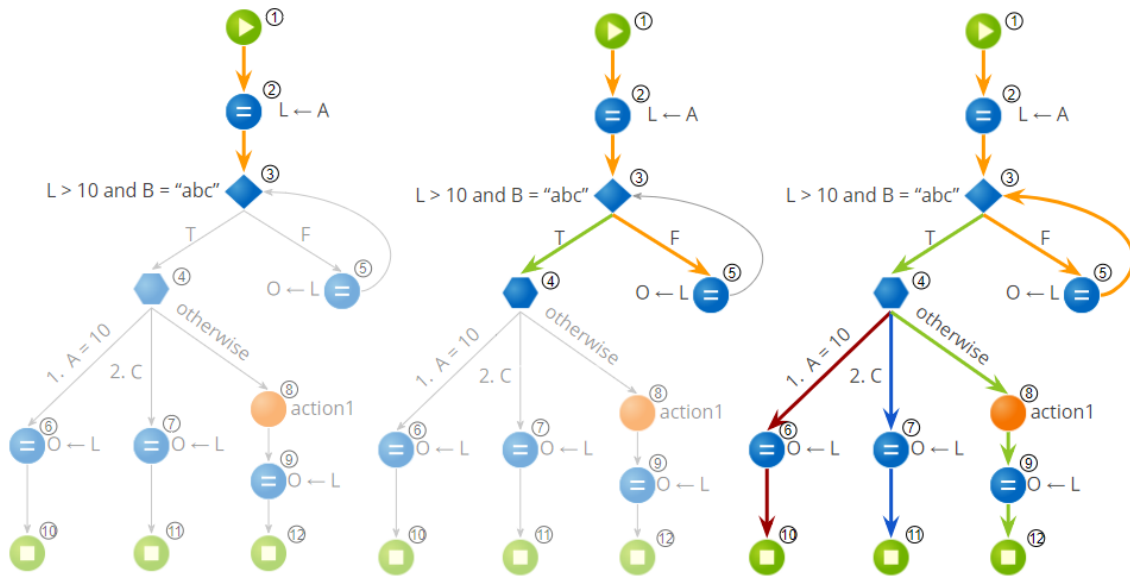


Figure 4.5: Graph traversal shows a breadth-first behaviour from the standpoint of the entire algorithm where the different colours represent multiple threads.

4.1.5 Process nodes

As it has been previously discussed (see subsection 2.1.2), the OutSystems applicational model is rather extensive and it would not be possible to analyse and evaluate every aspect of it. Therefore, this thesis focus is on the logic side of the applications, i.e., the client-server actions. This way, the main nodes subject to analysis in the prototype are the *Assign*, *If* and *Switch* nodes, as these are very important to exercise logic flows for OutSystems applications.

4.1.5.1 Start

Every procedure in OutSystems launches with a *Start* node, which is the beginning of the flow for this procedure.

When a *Start* node is encountered, a new test object is created containing simply the outgoing link from this *Start*. Then, a new block is added to the shared queue, comprising this new test object and the identifier of the outgoing node of this *Start*.

Algorithm 5 shows the pseudocode to process a *Start* node.

Example:

This traversal starts with an empty *sharedQueue* and receives the variables and parameters the graph defines as well as its *Start* node.

The algorithm will then take as input this *Start* node, get the link connecting it to the next node in the flow and creates a new block to add to the shared queue. This block will

Algorithm 4 Graph traversal

```

1: sharedQueue = { }
2: checkedOutTestObjects = { }
3:
4: function TRAVERSAL(threads, startNode, allVariables)
5:   call processStart(startNode, sharedQueue)
6:   for each t in threads do
7:     t.run()
8:   if traversalFinished then
9:     calcAndShowResults()
10:
11: function RUN( ) ▷ Code to be executed by each thread
12:   while traversalNotFinished do
13:     if notEmpty(sharedQueue) then
14:       {testObj, nextNode} ← removeFirst(sharedQueue)
15:       switch nextNode.type() do
16:         case IF: call ProcessIf(testObj, nextNode, sharedQueue)
17:         case SWITCH: call ProcessSwitch(testObj, nextNode, sharedQueue)
18:         case ASSIGN: call ProcessAssign(testObj, nextNode, sharedQueue)
19:         case END: call ProcessIf(testObj, checkedOutTestObjects, sharedQueue)
20:         case default: call ProcessUnsupportedNode(testObj, nextNode, sharedQueue)
  
```

Algorithm 5 Process Start node

```

1: function PROCESSSTART(startNode, sharedQueue)
2:   branch ← getLink(startNode)
3:   nextNode ← getOutgoing(branch)
4:   newTestObject ← {branch}
5:   sharedQueue ← sharedQueue ∪ {newTestObject, nextNode}
  
```

comprise a test object whose only content so far will be the branch that has been traversed (1 → 2).

This first stage of traversal ends by adding this block to the *sharedQueue*, after which a set of threads can start to try to pull work from. Figure 4.6 represents the state of the traversal after this initial step alongside the internal representation of this first block added to the shared queue.

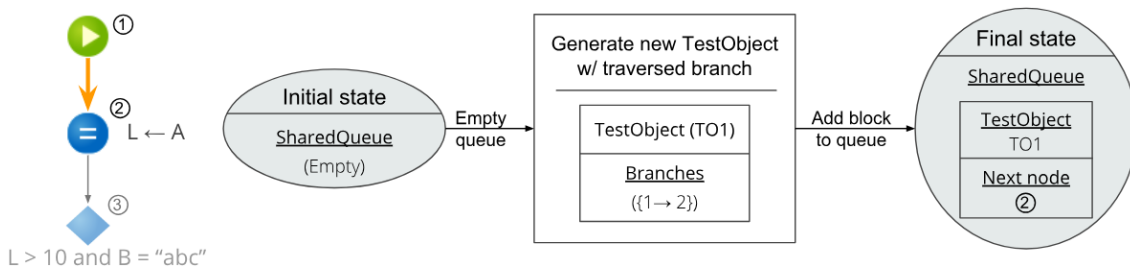


Figure 4.6: Traversal starts processing the *Start* node.

4.1.5.2 Assign

An *Assign* node is used for the purpose of assigning some value to a variable, and this can be of the form of another variable or a simple value. Thus, the following attributions are valid according to the variable types:

Integer :	String :	Boolean :
1. I = I2;	1. S = S2;	1. B = B2;
2. I = 5;	2. S = "word";	2. B = False;

As each *Assign* node can comprise multiple assignments, that must also be taken into account.

During graph traversal, whenever an *Assign* node is reached, for each assignment it contains, it simply records this assignment as a dependency in the test object that is following traversal.

Algorithm 6 Process Assign

```
1: function PROCESSASSIGN(testObject, assignNode, sharedQueue)
2:   branch ← getLink(assignNode)
3:   nextNode ← getOutgoing(branch)
4:
5:   for each attribution in getAttributions(assignNode) do
6:     variable ← getLeftSideAttribution(attribution)
7:     value ← getRightSideAttribution(attribution)
8:     dependency ← {variable, value}
9:     testObject ← testObject ∪ {dependency}
10:
11:   call addSharedQueue(testObject, branch, nextNode)
```

Example:

In the last section, the *Start* node was processed and as a result, the *sharedQueue* was left off with one block available to be pulled by any thread.

After pulling this block, the next node field points to the *Assign* of identifier 2. It contains a single assignment, $L \leftarrow A$, indicating that the local variable L shall take in the value of the input variable A . As explained, the only step involved in an *Assign* is to add its assignments as dependencies to the current test object.

Figure 4.7 shows the state of the algorithm before, during and after this *Assign* node is processed.

4.1.5.3 If

An *If* node consists of a typical if-then-else block in most programming languages. It contains a decision and, in OutSystems, always has two outgoing flows specifically defined

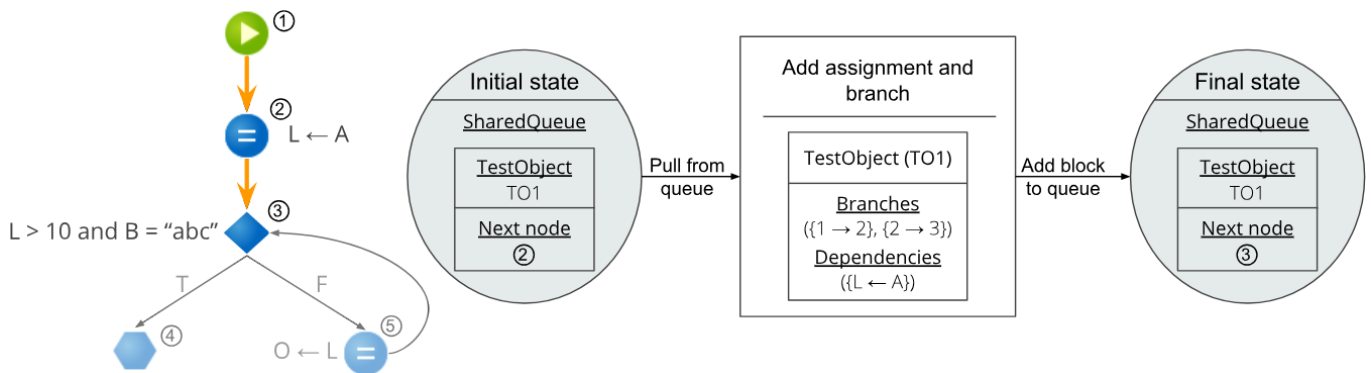


Figure 4.7: Processing an Assign node.

for the two possible outcomes: True and False.

When an *If* node is reached, the following steps are executed:

1. Add to a map all the values needed to be tested for each variable:
 - a) Add the values already in the current test object test cases;
 - b) Add all the values required to evaluate the current decision, dividing it into its set of simple conditions and retrieving the values according to the methodologies explained in [Data types and expressions](#);
2. To the map defined in 1, check if there are values needed to be tested for non-input variables and if so, verify if it depends upon an input variable, affecting the values to that variable instead;
3. At this point, there is a map containing all the candidate values to be tested for the decision under evaluation. The next step is to take these values and compute all its possible combinations thus generating all necessary test cases;
4. Now, with all the possible test cases defined in 3, each one will be tested in order to check if it meets the past decisions of the current test object. If it fails one decision, the test case is immediately discarded. This step is essential in employing a cause-effect behaviour in the traversal (see [section 2.2.4.2](#));
5. The test cases that survive the step 4 are then tested in order to check if they meet the decision currently under test. The test cases will then be grouped into the ones that meet this decision and the ones that do not, as they will follow different paths from now on.
6. Now, for each of the groups defined:
 - a) A new test object is created by copying the contents of the current test object (except the test cases it contained), and adding the set of test cases newly generated along with the next branch to be traversed. After this, before adding

this test object to the queue, it is verified if the next node in this particular path has already been processed by this test object. If it has, the test object is immediately marked as final and will not be added to the queue. This happens because a cycle is detected and seeing as complex expressions are not yet supported (see [subsection 4.1.3](#)), it would not be possible to break this cycle during traversal. This way, a warning is set to the developer warning him that a cycle is induced and so he should pay attention. If no cycle is detected, the test object is added to the shared queue.

[Algorithm 7](#) also shows in greater detail the steps previously introduced.

Algorithm 7 Process If

```

1: function PROCESSIF(testObject, ifNode, sharedQueue)
2:   branchFalse  $\leftarrow$  getLinkFalse(ifNode)
3:   branchTrue  $\leftarrow$  getLinkTrue(ifNode)
4:   decision  $\leftarrow$  getDecision(ifNode)
5:   varValues  $\leftarrow$  {} ▷ associates a variable with the values to be tested for it.
6:   varValues  $\leftarrow$  varValues  $\cup$  addValuesFromTestCases(testObject)
7:   varValues  $\leftarrow$  varValues  $\cup$  getValuesToEvaluate(decision)
8:   ▷ end of step 1
9:   for each var in varValues do
10:    if notInputVariable(var) then
11:      values  $\leftarrow$  varValues[var]
12:      if dependsOnInput(var) then
13:        varValues  $\leftarrow$  varValues  $\setminus$  {var, values}
14:        inputVar  $\leftarrow$  getDependency(var)
15:        tuple  $\leftarrow$  {inputVar, values}
16:        varValues  $\leftarrow$  varValues  $\cup$  tuple
17:      ▷ end of step 2
18:    testCases  $\leftarrow$  combineAllVarValues(varValues)
19:    ▷ end of step 3
20:    prevDecisions  $\leftarrow$  getPreviousDecisions(testObject)
21:    testCases  $\leftarrow$  call passesPrevDecisions(prevDecisions, testCases)
22:    ▷ end of step 4
23:    trueTestCases  $\leftarrow$  {}   falseTestCases  $\leftarrow$  {}
24:    for each test in testCases do
25:      if notPasses(test, decision) then
26:        falseTestCases  $\leftarrow$  falseTestCases  $\cup$  test
27:      else
28:        trueTestCases  $\leftarrow$  trueTestCases  $\cup$  test
29:      ▷ end of step 5
30:    call createTOandAddSharedQueue(testObject, trueTestCases, branchTrue,
31:    decision, true)
32:    call createTOandAddSharedQueue(testObject, falseTestCases, branchFalse,
33:    decision, false)
34:    ▷ end of step 6

```

Algorithm 8 shows some auxiliary functions to be used when most nodes are processed and are thus here gathered.

Algorithm 8 Auxiliary functions

```

1: function CREATEOANDADDSHAREDQUEUE(testObject, testCases, branch,
   decision, passed)
2:   newTestObject  $\leftarrow$  copyTestObject(testObject)
3:   newTestObject  $\leftarrow$  newTestObject  $\cup$  testCases
4:   newTestObject  $\leftarrow$  newTestObject  $\cup$  {decision, passed}
5:   nextNode  $\leftarrow$  {getOutgoing(branch)}
6:   call addSharedQueue(newTestObject, branch, nextNode)
7: function ADDSHAREDQUEUE(testObject, branch, nextNode)
8:   if containsNode(testObject, nextNode) then
9:     testObject  $\leftarrow$  testObject  $\cup$  branch
10:    call yieldCycleWarning(testObject, branch)
11:   else
12:     testObject  $\leftarrow$  testObject  $\cup$  branch
13:     sharedQueue  $\leftarrow$  sharedQueue  $\cup$  {testObject, nextNode}
14: function PASSESPREVDECISIONS(prevDecisions, testCases)
15:   for each test in testCases do
16:     for each prevD in prevDecisions do
17:       if notPasses(test, prevD) then
18:         testCases  $\leftarrow$  testCases  $\setminus$  {test}
19: function YIELDCYCLEWARNING(testObject, branch)
20:   yieldWarning('A cycle was detected regarding the link:' + branch)
21:   setFinishedTO(testObject)

```

Example:

After processing the first *Assign*, this example continues by pulling the only block in the shared queue that points to an *If* node as the next one to be processed.

For step 1, as the current test object does not have any test cases defined, the values to evaluate will result from the ones retrieved from the decision $L > 10$ and $B = "abc"$. This decision's simple conditions are $L > 10$; $B = "abc"$ and thus the values to evaluate for this decision are the following:

$$\begin{aligned}
 \text{valuesToTest} &\leftarrow \{ L = \{9, 10, 11\}, \\
 &\quad B = \{ "abc", "rand1", "" \} \}
 \end{aligned}$$

Step 2 identifies that there is a non-input variable with values to evaluate, L , and verifies that it does depend on the input variable A , thus transferring these values to A , and now we have:

$$\begin{aligned}
 \text{valuesToTest} &\leftarrow \{ A = \{9, 10, 11\}, \\
 &\quad B = \{ "abc", "rand1", "" \} \}
 \end{aligned}$$

Step 3 now computes all combinations between variables and values to test, generating the following test cases:

$TC1 = [A = 9, B = "abc"]$ $TC2 = [A = 9, B = "rand1"]$ $TC3 = [A = 9, B = ""]$
 $TC4 = [A = 10, B = "abc"]$ $TC5 = [A = 10, B = "rand1"]$ $TC6 = [A = 10, B = ""]$
 $TC7 = [A = 11, B = "abc"]$ $TC8 = [A = 11, B = "rand1"]$ $TC9 = [A = 11, B = ""]$

Step 4 does not remove any of the test cases previously defined as there are still no decisions saved onto the test object used in this traversal.

Step 5 will now group the combinations generated by if they meet the decision $L > 10$ and $B = "abc"$ or not. Thus, in this case only $TC7$ meets the decision and will continue on traversal following the true branch ($3 \rightarrow 4$), while the remainder do not meet the decision and shall continue through the false branch ($3 \rightarrow 5$).

Finally, step 6 creates a test object for each group, adding to both the contents of the current test object, supplementing it with the appropriate set of test cases and the branch that those test objects will next traverse. These test objects will be encapsulated in a block each to be added to the shared queue, pointing to the correct node to be next processed, thus finishing the process of this *If* node.

Figure 4.8 represents the state of the algorithm during the traversal of this node and Figure 4.9 shows the final state of the graph.

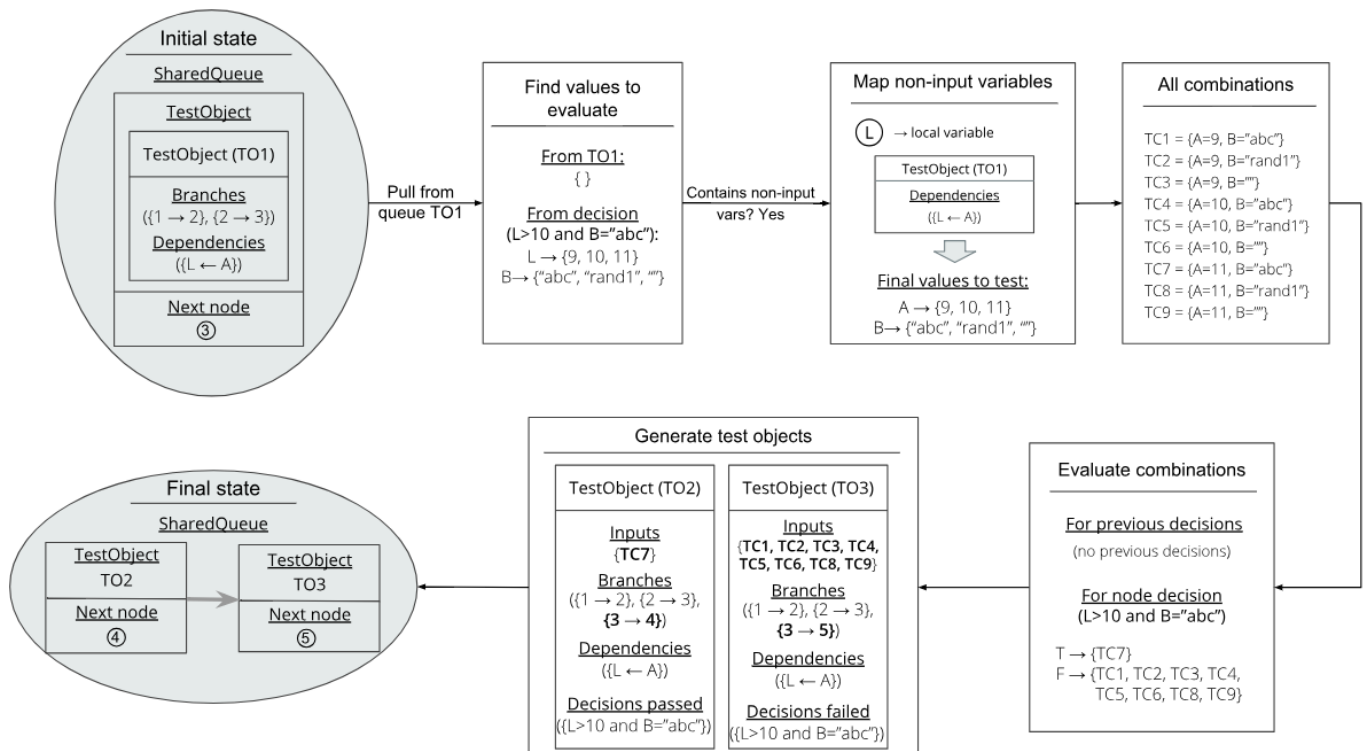


Figure 4.8: Processing an *If* node.

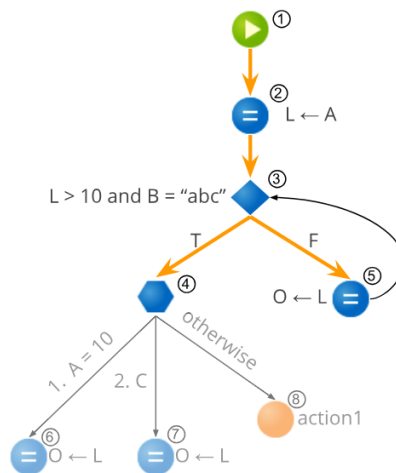


Figure 4.9: Graph after processing the *If* node.

4.1.5.4 *Switch*

The *Switch* node receives multiple decisions that the developer wants to verify and the flow of information will then take the path whose decision it meets. If none does, in the OutSystems language, the value *otherwise* is also explicitly defined.

Thus, a *Switch* node can be perceived as a sequence of if-then-else blocks and is processed in a similar way to the one already described for an *If*.

Just like for *If* nodes, the values to evaluate are found by first getting all the ones already in the test object and then, in the *If* node the values for the decision would be identified but seeing as a switch node can have multiple decisions defined, the values for each of those decisions are also added.

After this step, the execution continues by following the same steps already defined for *If* nodes (see 2, 3, 4). After step 4, each one of these test cases is evaluated against each one of the decisions. If it meets a decision, it shall follow its path. If it does not meet any decision, then it shall follow the *otherwise* path. The test cases are then grouped according to the decision they verify and a test object per group is created, always maintaining the information already retrieved over the course of the traversal of the current test object. These test objects are finally added to the shared queue, if no cycles are detected.

[Algorithm 9](#) also shows in greater detail the mechanism described.

Example:

After processing the *If* node, a thread now pulls the first element from the queue, that contains the test object *TO2* and points towards the node of identifier 4, a *Switch* node that will now be processed.

The first step is to identify the values to test, by first getting the values already in the test cases present in *TO2* and then identifying all the values to test from the decisions defined in the *Switch* ($A = 10$ and C), thus we have:

Algorithm 9 Process Switch

```
1: function PROCESSSWITCH(testObject, switchNode, sharedQueue)
2:   decisions  $\leftarrow$  getDecisions(switchNode)
3:   varValues  $\leftarrow$  {}
4:   for each decision in decisions do
5:     vals  $\leftarrow$  getValuesToEvaluate(decision)
6:     varValues  $\leftarrow$  varValues  $\cup$  vals
7:     testCases  $\leftarrow$  combineAllVarValues(varValues)
8:     testCases  $\leftarrow$  call passesPrevDecisions(prevDecisions, testCases)
9:     groups  $\leftarrow$  {}  $\triangleright$  maps the decisions to the set of test cases that shall follow that
      decisions's path
10:    otherwiseTcs  $\leftarrow$  {}
11:    for each tc in testCases do
12:      if matchesDecision(tc, decisions) then
13:        d  $\leftarrow$  decisionMatched(tc, decisions)
14:        groups[d]  $\leftarrow$  groups[d]  $\cup$  tc
15:      else
16:        otherwiseTcs  $\leftarrow$  otherwiseTcs  $\cup$  tc
17:      for each {decision, tCs} in groups do
18:        branch  $\leftarrow$  getLink(decision, switchNode)
19:        call createTOandAddSharedQueue(testObject, tCs, branch, decision, true)
20:        branchOtherwise  $\leftarrow$  getOtherwiseLink(switchNode)
21:        call createTOandAddSharedQueue(testObject, otherwiseTcs, branchOtherwise,
22:        decision, false)
```

1. From *TO2*: $A = \{11\}$, $B = \{\text{"abc"}\}$

2. From $\{A = 10\}$: $A = \{9, 10, 11\}$

3. From $\{C\}$: $C = \{\text{True}, \text{False}\}$

Final values to evaluate: $A = \{9, 10, 11\}$ $B = \{\text{"abc"}\}$ $C = \{\text{True}, \text{False}\}$

Having all the variables and values to evaluate, just like for the *If*, it is verified if any of those variables are non-input. Contrarily to the last example, here we have all input variables and thus no mapping needs to take place.

Now, in Step 3 the combination of all test cases is computed:

$TC1 = [A = 9, B = \text{"abc"}, C = \text{True}]$ $TC2 = [A = 9, B = \text{"abc"}, C = \text{False}]$

$TC3 = [A = 10, B = \text{"abc"}, C = \text{True}]$ $TC4 = [A = 10, B = \text{"abc"}, C = \text{False}]$

$TC5 = [A = 11, B = \text{"abc"}, C = \text{True}]$ $TC6 = [A = 11, B = \text{"abc"}, C = \text{False}]$

Having all combinations, we will then check if they are valid in this point of the path, by testing them against previous passed/failed decisions that have been encountered over the current path. *TO2* has recorded that the decision $\{L > 10 \text{ and } B = \text{"abc"}\}$ was previously met therefore, when evaluating each of the test cases *TC1*, *TC2*, *TC3* and *TC4* do not meet that decision, meaning that they are combinations that would never reach

this point of traversal, so they are removed from the set of test cases to evaluate, where *TC5* and *TC6* remain for this *Switch*.

The next step is to then group the remaining two test cases according to the path they will follow, whether that is any of the *Switch* decisions, or the defined *otherwise* branch.

As both concern the variable *A* with the value 11, the branch that verifies the decision $A = 10$ will actually have no test cases to follow that path. If we look closely at the composition of this graph, we can see that the previous *If* node traversed required that for its true branch, the condition $L > 10$ had to be true, and we also saw that the local variable *L* receives the value of the input variable *A*, meaning that actually, the condition $A > 10$ needs to be true in order to reach the *Switch* node, thus it will never be possible to have any combination of input traversing the *Switch* branch that requests for $A = 10$! We now already know that that branch and following path will become unreachable and will yield dead code. Let's see the confirmation from the algorithm's result closer to the end of this chapter.

So, checking each *TC5* and *TC6*, we already know that none will verify the condition $A = 10$, but for the condition *C*, we have *TC5* that contains the $C = True$ and will, therefore, follow that branch. *TC6*, on the other hand, claims $C = False$ and will then not verify any *Switch* decision, following the *otherwise* branch.

Finally, just as before, for each group of generated test cases, a test object will be created and added to the queue.

Figure 4.10 and Figure 4.11 show the state of the algorithm and graph during the traversal of this *Switch* node.

Continuing this example, the next element to be pulled by the shared queue points towards the node of identifier 5, an *Assign*. As assigns have been previously exemplified, we'll go over this one briefly.

For the *Assign* node, the only step that takes place is to add the dependencies it encapsulates back into the current test object (*TO3*) and then tries to push it back to the end of the queue. After adding the dependency and before adding the block back into the queue, it is verified if a cycle is encountered, by checking if the next node to be followed by *TO3* would be a node already traversed in its path. That node has the identifier 3 and it is confirmed that it has indeed been processed by this test object before. Thus, the traversal of this test object is stopped and it is marked as finished.

Figure 4.12 now shows the state of the traversal after processing the node with identifier 5.

4.1.5.5 *Unsupported Node*

The reasoning behind the choice of the initial set of nodes currently supported by this algorithm was previously depicted. As procedures developed in OutSystems normally have much more complexity in terms of the variety of the nodes used, and in order not to block the evaluation of more complex procedures, this algorithm "bypasses" not

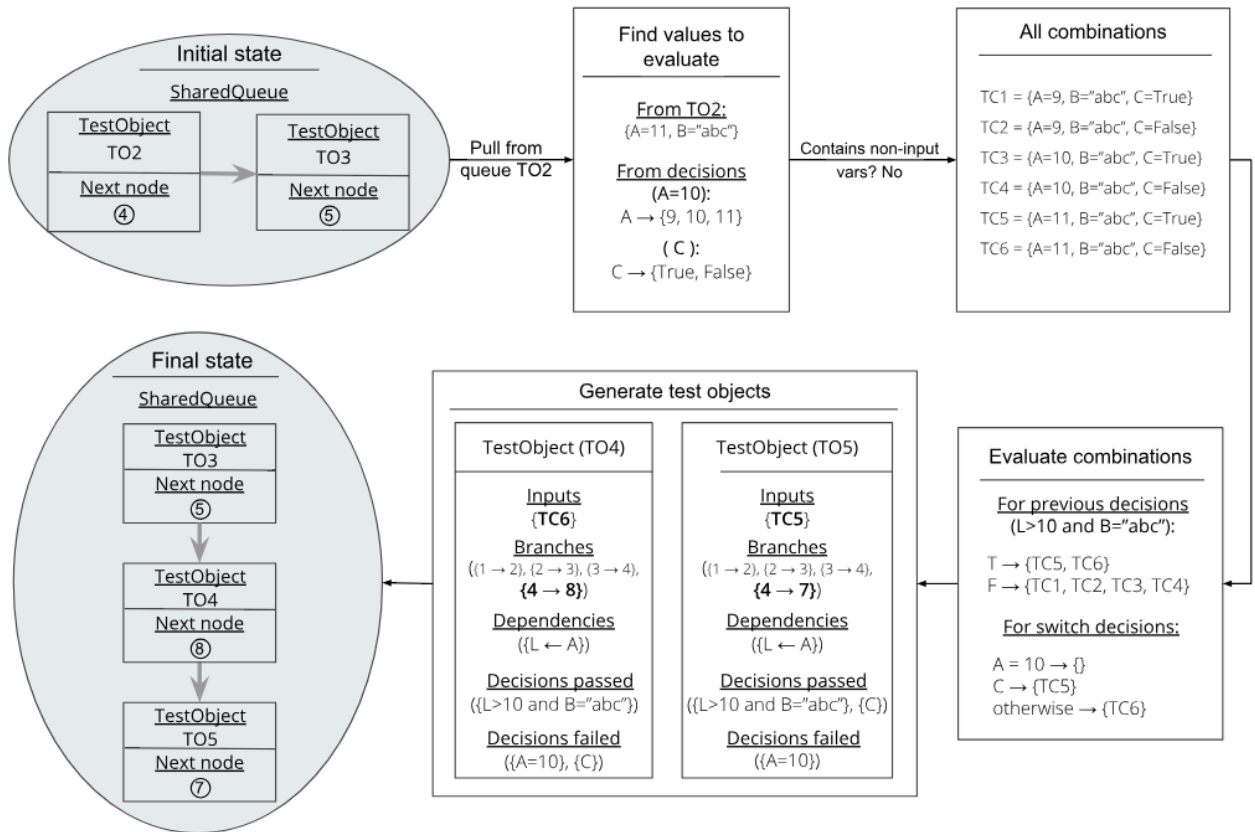


Figure 4.10: Processing a *Switch* node.

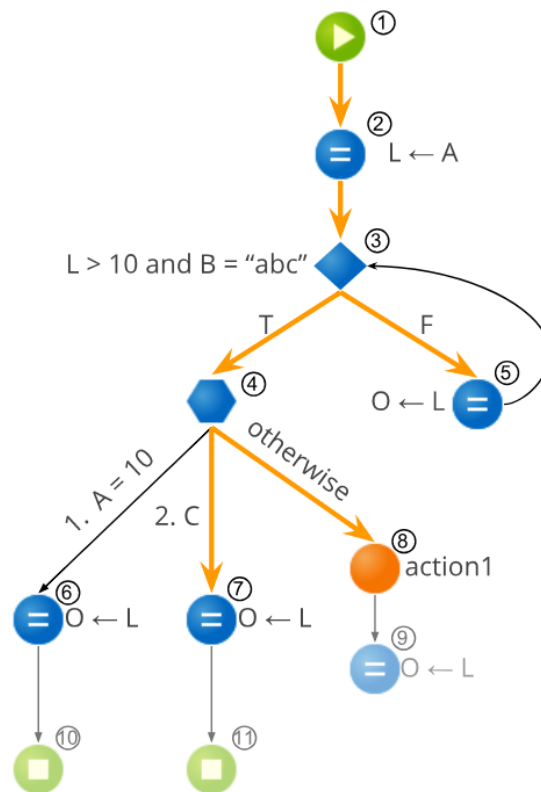


Figure 4.11: Graph after processing the *Switch* node.

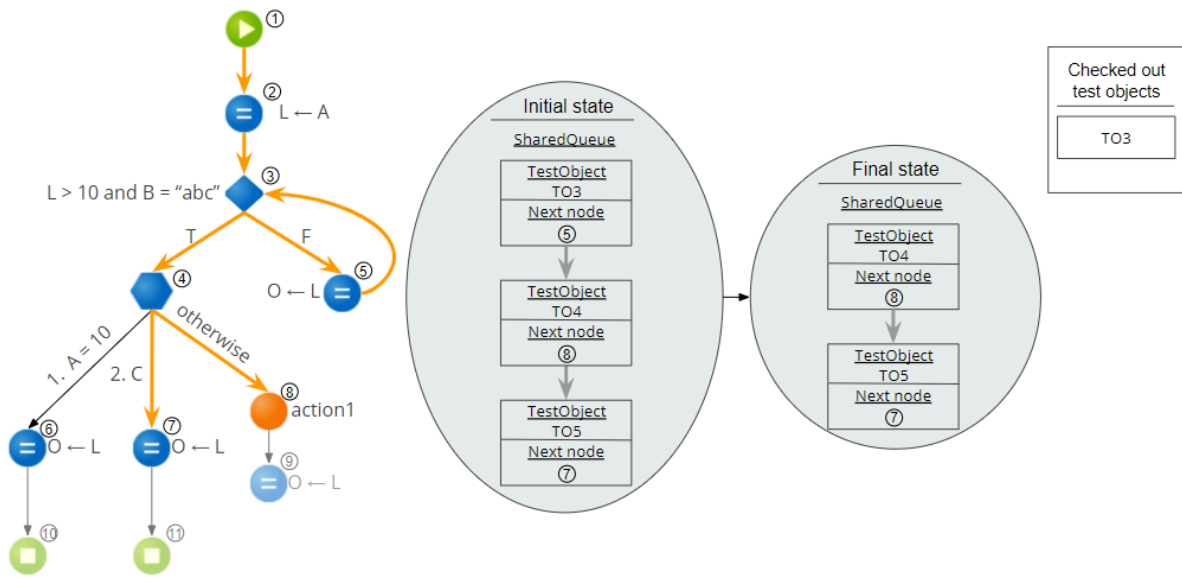


Figure 4.12: Graph after processing the node of identifier 5.

supported nodes by simply jumping over to their outgoing flows. If the node contains multiple outgoing flows, the test object following the *Unsupported* node will be multiplied in order to have one test object per each outgoing flow (i.e., one test object per independent path).

Algorithm 10 shows the pseudocode to process an *Unsupported* node.

Algorithm 10 Process *Unsupported*

```

1: function PROCESSUNSUPPORTEDNODE(testObject, unsupportedNode, sharedQueue)
2:   outgoings ← getLinks(unsupportedNode)
3:   for each out in outgoings do
4:     newTestObject ← copyTestObject(testObject)
5:     if containsBranch(testObject, branch) then
6:       call yieldCycleWarning(testObject, branch)
7:     else
8:       newTestObject ← newTestObject ∪ out
9:       nextNode ← {getOutgoing(out)}
10:    sharedQueue ← sharedQueue ∪ {newTestObject, nextNode}

```

Example:

Pulling the first element of the queue, we now have *TO4* and the pointer to the node of identifier 8, a node that is not of the type *Assign*, *If* nor *Switch* and is thus a not supported node type.

Therefore, the only step to take here is to get the outgoing link for this node (it only contains one {8 → 9}), add it to *TO4* and add this test object back to the end of the queue, encapsulated in a block along with a pointer to the node of identifier 9.

Figure 4.13 shows the state of traversal after processing this *Unsupported* node.

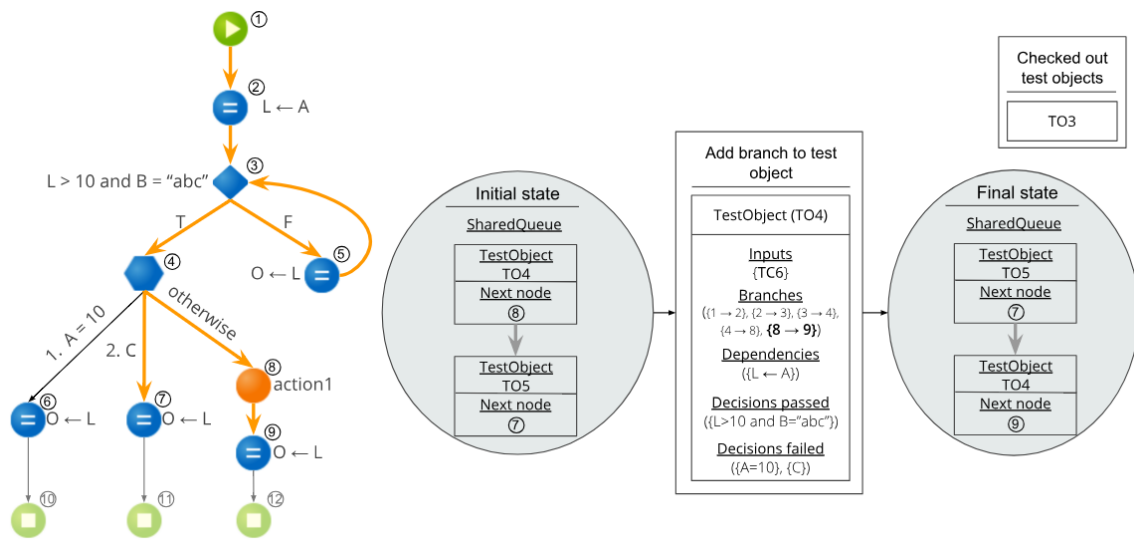


Figure 4.13: Processing a *Unsupported* node.

Continuing the example, the next block to pull from the queue contains the test object *TO5*, and points to the node of identifier 7, an *Assign* node. Again, its assignment is saved onto the test object as a dependency, its outgoing link added as a branch traversed ($7 \rightarrow 11$) and the block comprising of *TO5* and the identifier 11 is again added to the end of the queue.

Figure 4.14 shows both the graph and the algorithm's state after processing this node.

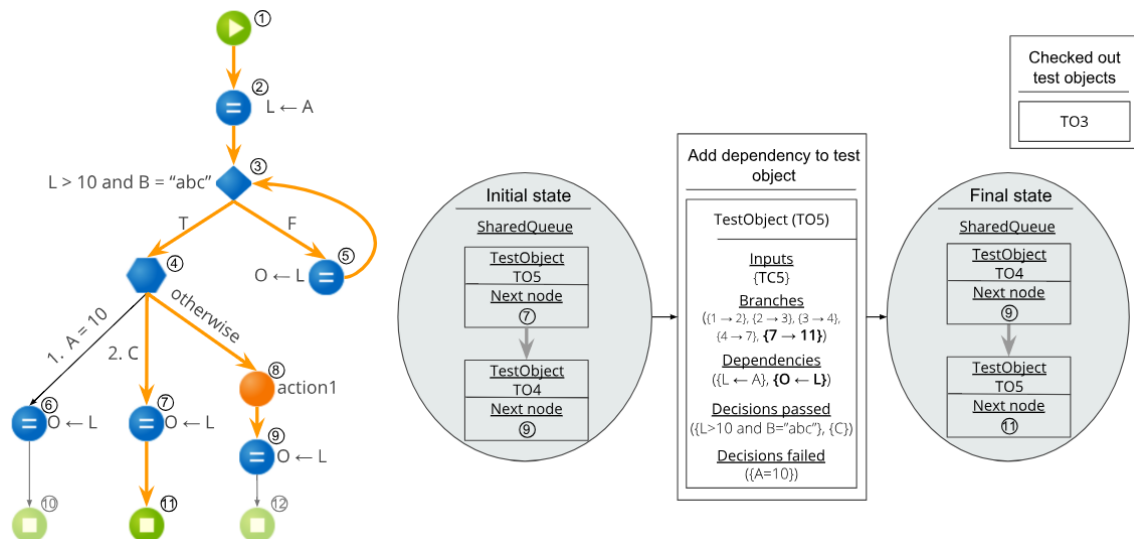


Figure 4.14: The graph and algorithm's state after traversal of node with identifier 7.

Up next, we have in the queue the block comprising of *TO4* and the node 9, another *Assign* node. Again, the same step of adding both the assignment and the outgoing link to the test object and adding it back to the end of the queue takes place.

Figure 4.15 shows both the graph and the algorithm's state after processing this node.

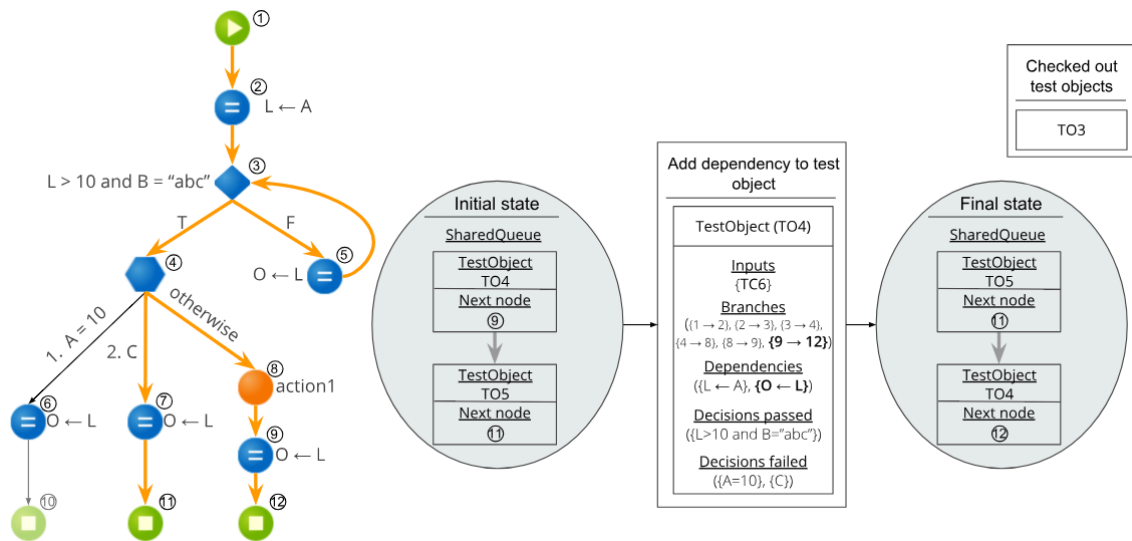


Figure 4.15: The graph and algorithm’s state after traversal of node with identifier 9.

4.1.5.6 End

The *End* node marks the end of execution for a particular path in the procedure. In order to have a correct flow of information, all paths must begin with a *Start* node and finish with an *End*.

For *End* nodes, the only step executed is to add the test object onto a set of test objects marked as finalized. These finished test objects represent paths that have reached an end after the algorithm’s traversal of the graph. In a correct procedure, the number of finished test objects would coincide with the number of independent paths in the graph. Situations such as traces of unreachable branches will reduce the overall number of test objects.

Algorithm 11 shows the pseudocode to process an *End* node.

Algorithm 11 Process End

- 1: **function** PROCESSEND(*testObject*, *checkedOutTestObjects*)
 - 2: *checkedOutTestObjects* ← *checkedOutTestObjects* ∪ *testObject*
-

Example:

The next block in the queue contains the test object *TO5* and points to an *End* node, thus the test object is marked as final. The same happens for the last block in the queue, containing *TO4* and pointing to the node 12 (also an *End* node).

Here we arrive at the end of traversal for this example. The way coverage criteria and warnings are computed is explained in the next sections, as well as the final results presented to the user. Figure 4.16 shows these last two blocks being processed and the final state of the test objects checked out.

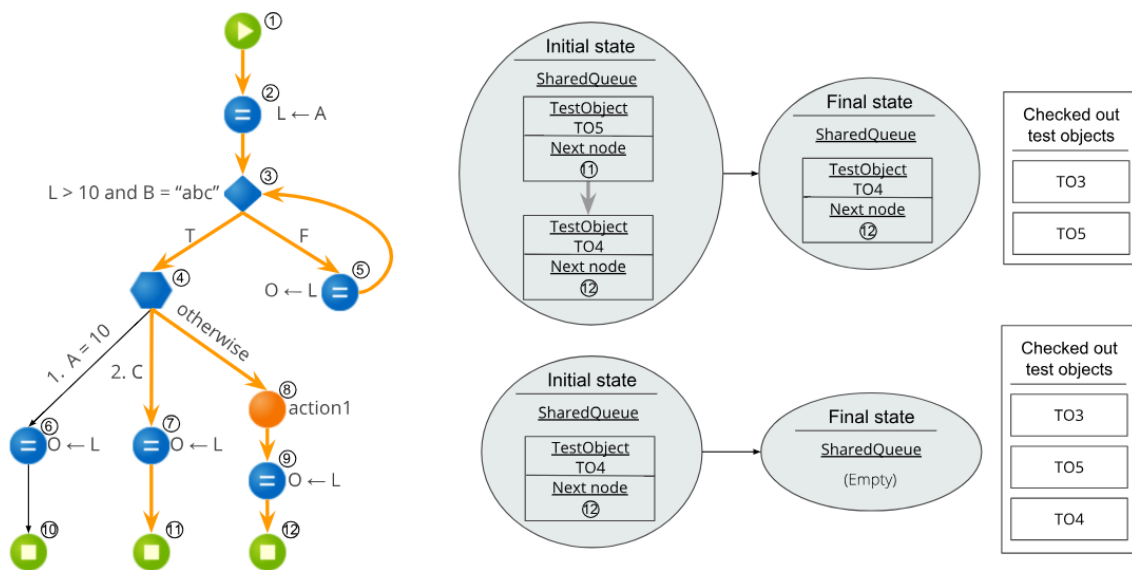


Figure 4.16: The graph and algorithm’s state while processing the last couple nodes and after the end of traversal of this graph.

4.1.6 Coverage evaluation

To evaluate the coverage obtained by the entire set of test cases defined, and also to be able to pinpoint the exact percentage of coverage provided by a selected few, some important information is stored in the test objects (subsection 4.1.2). This way, by analysing the final results, these percentages can be easily computed without the need to traverse the graph once more.

Example:

Figure 4.17 shows the final state of the test objects that reached the end of traversal for the graph that has been used as example throughout this chapter. These test objects will now help calculate the following coverage criteria and in identifying the final warnings to the developer.

Figure 4.18 presents the characteristics of the graph that are also used to compute the coverage criteria presented next.

4.1.6.1 Node

To compute the percentage of node coverage (see section 2.4.1) obtained by the test cases selected (or by all of them), the algorithm will pick from each test object the nodes it reaches by checking the branches covered and compares these with the entirety of nodes the graph contains.

Example:

The following examples will show how the different coverage criteria are calculated for the full set of test objects generated. The same methodology is applied when the coverage

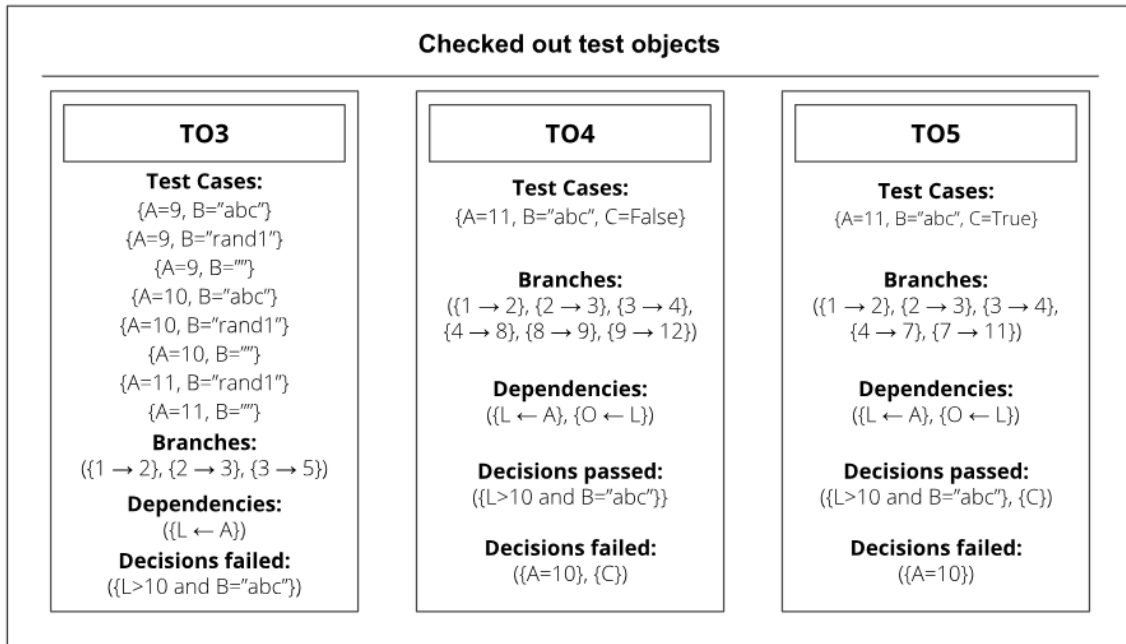


Figure 4.17: Final test objects after graph traversal.

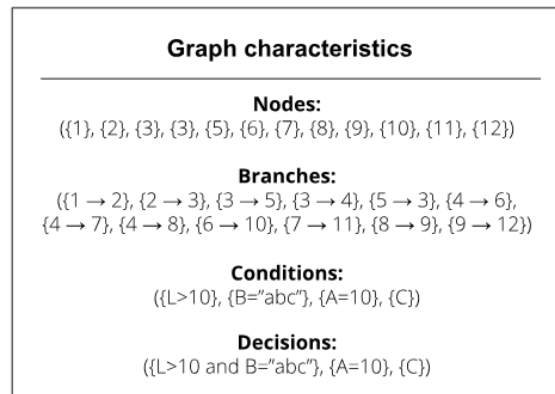


Figure 4.18: Graph characteristics: set of all nodes, branches, simple conditions and decisions found in the graph.

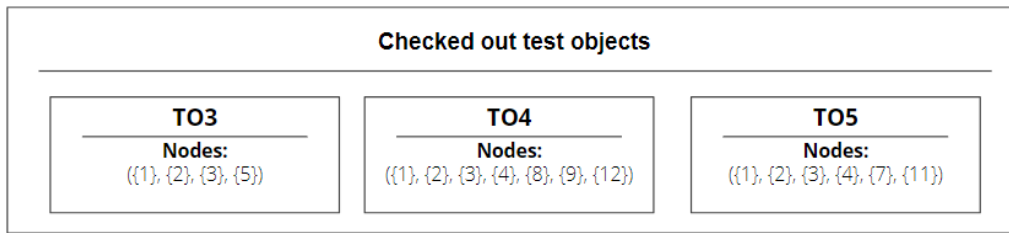
is calculated for only a sub-set of the test cases identified (each test case is mapped back to its parent test object and the set of test objects selected are then analysed).

As explained, to calculate node coverage, the nodes the set of test objects cover are identified and checked against the total set of nodes the graph contains.

Thus, *TO3* covers the following nodes: 1, 2, 3, 5; *TO4* covers: 1, 2, 3, 4, 8, 9, 12 and *TO5* covers: 1, 2, 3, 4, 7, 11. In total, the following nodes are covered by the full set of test objects generated during graph traversal: 1, 2, 3, 4, 5, 7, 8, 9, 11, 12 (ten nodes).

Comparing this with the number of nodes the graph contains, we see the graph has twelve nodes, meaning that we will not have 100% node coverage.

$$\text{Node coverage} : \frac{\text{nodes covered in test objects}}{\text{total nodes}} \Rightarrow \frac{10}{12} = 0.83(3) \approx 83\%$$



Nodes covered by TO3, TO4 and TO5:
 ({1}, {2}, {3}, {4}, {5}, {7}, {8}, {9}, {11}, {12}) → 10 nodes

Figure 4.19: Nodes covered by each of the test objects generated by this algorithm.

The total node coverage obtained in this example is 83%. Because it is less than 100%, we already know that something is probably not right with this graph. In the warnings section we will revise this topic.

Figure 4.19 represents graphically the nodes covered by the set of test objects generated.

4.1.6.2 Branch

In a very similar manner to node coverage, branch coverage (see section 2.4.1) is also computed directly through the information already stored in the test objects, comparing it to the full set of branches in the graph in order to calculate a percentage.

Example:

Similarly to node coverage, here the branches covered by each of the test objects are identified and thus we have that *TO3*, *TO4* and *TO5* cover the following branches: ({1 → 2}, {2 → 3}, {3 → 5}, {3 → 4}, {5 → 3}, {4 → 7}, {4 → 8}, {7 → 11}, {8 → 9}, {9 → 12}) (ten branches). Figure 4.20 represents this information graphically.

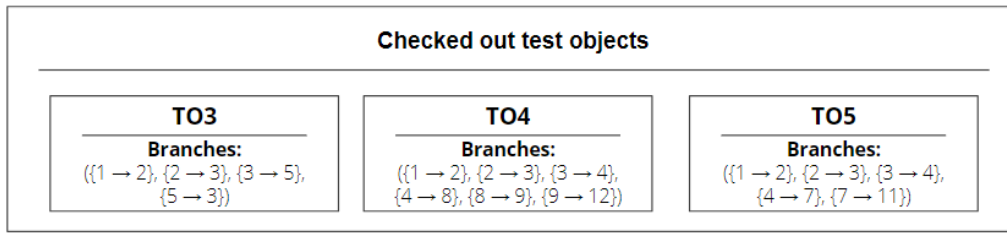
$$\text{Branch coverage: } \frac{\text{branches covered in test objects}}{\text{total branches}} \Rightarrow \frac{10}{12} = 0.83(3) \approx 83\%$$

Just as before, here the test objects cover ten out of twelve branches so we will have 83% for branch coverage.

4.1.6.3 Condition

Condition coverage (see section 2.4.1) verifies if all individual conditions present on the graph are evaluated for both its True and False values.

To evaluate the percentage of condition coverage, the set of all conditions in the graph is analysed for each test case that reaches said condition in the path it covers, and the amount of conditions that are both evaluated for the values True and False are marked as fulfilling condition coverage. The ratio of conditions evaluated and the conditions that are not evaluated will represent the final condition coverage percentage.



Branches covered by TO3, TO4 and TO5:
 {(1 → 2), (2 → 3), (3 → 4), (3 → 5), (4 → 8), (4 → 7), (5 → 3), (7 → 11), (8 → 9), (9 → 12)} → 10 branches

Figure 4.20: Branches covered by each of the test objects generated by this algorithm.

Example:

Figure 4.18 showed the full set of conditions for the graph used as example: $\{L > 10\}$, $\{B = "abc"\}$, $\{A = 10\}$ and $\{C\}$. We also know that the local variable L , after the first node, depends on the input variable A , thus we can say instead that we have the condition: $\{A > 10\}$.

The input combinations generated are now divided according to the conditions they cover, and tested whether they meet or not the condition (we only need to check if there are, for each condition, two test cases where one meets the condition and the other does not) so we have:

- For the condition $A > 10$: $[A = 9, B = "abc"] \rightarrow F$
 $[A = 11, B = "abc"] \rightarrow T$
- For the condition $B = "abc"$: $[A = 11, B = "rand1"] \rightarrow F$
 $[A = 11, B = "abc"] \rightarrow T$
- For the condition C : $[A = 11, B = "abc", C = False] \rightarrow F$
 $[A = 11, B = "abc", C = True] \rightarrow T$
- For the condition $A = 10$: $[A = 11, B = "abc", C = False] \rightarrow F$
 $[A = 11, B = "abc", C = True] \rightarrow F$

Here, there is no test case that meets this condition, so condition coverage is not verified for $A = 10$.

$$\text{Condition coverage: } \frac{\text{conditions covered}}{\text{total conditions}} \Rightarrow \frac{3}{4} = 0.75 = 75\%$$

In total, there are four conditions in the graph and only three verify the criterion, thus leaving us with 75% condition coverage.

Figure 4.21 shows the test cases per condition and the conditions that verify condition coverage.

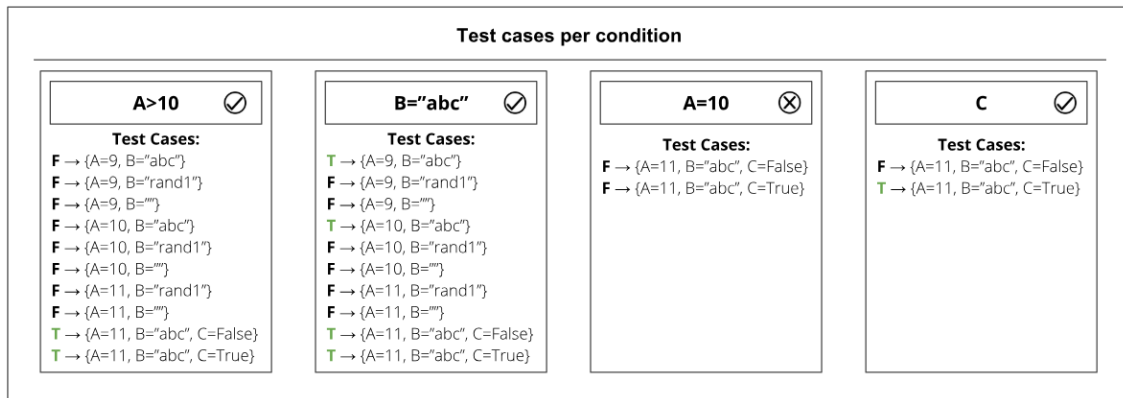


Figure 4.21: Test cases per condition.

4.1.6.4 Modified condition-decision

To calculate modified condition-decision coverage (see section 2.4.1), this algorithm starts by identifying the test cases that cover each decision (i.e., that pass through said decision over the path they traverse). Then, for each decision, each test case will represent an entry for that decision's truth table and for the entries generated, it will be verified if this criterion is held. For that, it is required for each individual variable to affect the result of the decision, i.e., checking if by fixing the values of all other variables and switching only the value of one variable at a time, the result of the decision changes.

The percentage is calculated by checking the number of decisions whose set of test cases verify modified condition-decision coverage against the full set of decisions.

Example:

In a similar manner to the previous criterion, here we also start by grouping the test cases we have by the decisions they cover (before they were grouped by the conditions). Then, for each group, we test if they verify this criterion is met.

For that, a truth table is drawn for each decision, based on the results provided by each of the test cases available. Figure 4.22 shows the entries, for each decision, that are able to be drawn based on the test cases generated. It can be seen that for the decisions {L > 10 and B = "abc"}; and {C}, the entire truth tables are generated, meaning that this criterion will be covered for both of those decisions. For A = 10, we have only one entry, for when this decision turns false. Because we are not able to check the case for when the condition A = 10 changes the value of the overall decision (as this is a decision with only one condition, the full table needs to be drawn in order to have modified condition-decision coverage).

Figure 4.23 shows the test cases per decision and the decisions that verify modified condition-decision coverage.

$$\text{Modified condition - decision coverage} = \frac{\text{decisions covered}}{\text{total decisions}} \Rightarrow \frac{2}{3} = 0.66(6) \approx 66\%$$

A>10 and B="abc"		Test case
T	T	{A=11, B="abc", C=False}
T	F	{A=11, B=""}
F	T	{A=9, B="abc"}
F	F	{A=9, B="rand1"}

A=10	Test case
T	(None) ⊗
F	{A=11, B="abc", C=True}

C	Test case
T	{A=11, B="abc", C=False}
F	{A=11, B="abc", C=True}

Figure 4.22: Truth tables for each decision, showing if there is a test object that is able to vouch for said entry.

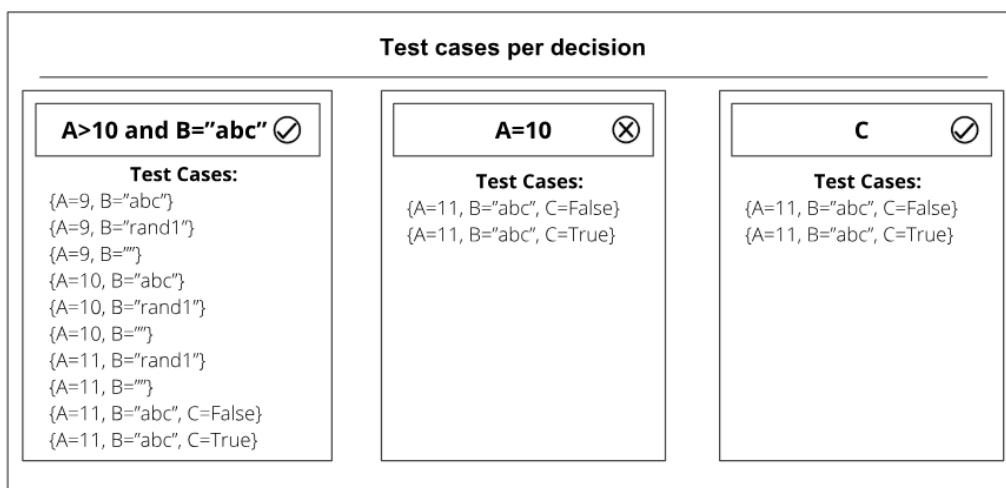


Figure 4.23: Test cases per decision.

Again, the decision $A = 10$ does not verify this criterion, so we have two out of three decisions which results in around 66% for modified condition-decision coverage.

4.1.6.5 Multiple condition

For multiple condition coverage (see [section 2.4.1](#)), all possible combinations of conditions in each decision are analysed, as if a truth table would be drawn for each decision.

So, in order to evaluate the percentage of condition coverage, for each decision, a truth table is generated through the test cases that cover said decision (similarly to modified condition-decision) and it is compared to the total number of decisions that have a full truth table generated against the number of total decisions in the graph.

Example:

Already having the test cases grouped by decision from the previous criterion (Figure 4.23), each of these test cases will correspond to an entry in the truth table relative to the decision. If we have all the different entries required covered by, at least, one test case, then said decision is marked as fulfilling multiple decision coverage.

Figure 4.22 shows the truth tables for each decision, associating for each entry a test case, if it verifies said entry. Just like for the previous couple criteria, the decision $A = 10$ will not verify this criterion, while the remainder decisions do.

$$\text{Multiple condition coverage: } \frac{\text{decisions covered}}{\text{total decisions}} \Rightarrow \frac{2}{3} = 0.66(6) \approx 66\%$$

Again we have two out of three decisions verifying this criterion thus resulting in 66% for multiple condition coverage.

4.1.7 Expected output

This algorithm also produces the expected output for the procedure. In OutSystems, output variables are defined and used in order to return values. There are no explicit statements for this (such as, for some programming languages, the *return* statement). Instead, output variables are defined, receive values through *Assign* nodes and then always return the state they hold when the procedure terminates.

To identify these output values, after traversal is finished, for each test case, the dependencies for the output variables are searched and the expected value is computed accordingly.

Example:

This example contains one output variable, O . After traversal, for each test object, its dependencies are identified and the expected values for O are calculated.

Starting with $TO3$, it contains the dependency $\{O \leftarrow L\}$ and since we also know that $\{L \leftarrow A\}$, each test case for $TO3$ will return for O the value it has stored for A . If no value for A is found (or if the output variable did not have any dependency for this path), the expected value presented for O would be *N.D.* (not defined).

Since all other test objects have the same dependencies, for every test case generated, the output variable will simply take the value received by A .

Table 4.1: Expected values for the output variable.

Test case	Expected output
{A=9, B="abc"}	{O=9}
{A=9, B="rand1"}	{O=9}
{A=9, B=""}	{O=9}
{A=10, B="abc"}	{O=10}
{A=10, B="rand1"}	{O=10}
{A=10, B=""}	{O=10}
{A=11, B="rand1"}	{O=11}
{A=11, B=""}	{O=11}
{A=11, B="abc", C=False}	{O=11}
{A=11, B="abc", C=True}	{O=11}

4.1.8 Warnings evaluation

Along with the definition of the full set of test cases and expected output values, this algorithm can also identify some warnings that the user should take into account. These include the identification of variables that are defined but never used, traces of code that are never reached due to the combination of different decisions that made them unreachable, and cycles that might never break.

4.1.8.1 Unused variables

To check if a variable is unused, the set of variables that all test objects found over its traversal is checked against the set of all the variables defined within the graph. The result is computed by a simple difference between sets.

Example:

Recalling the example (Figure 4.1), we see that it has defined variables of all types: input (A , B , C), local (L , U) and output (O). By checking the variables involved in both attributions and while evaluating decisions, we see that all but the local variable U are used at some point in that graph. Thus U is never used and a warning regarding that variable is presented to back the developer.

4.1.8.2 Dead code

As the algorithm consists in the full traversal of the graph, the existence of dead code is evaluated by checking when the full suite of test objects cannot reach certain paths.

This way, knowing all the branches defined in the graph and all the ones reached by the test objects, the branches that are not reachable are easily identified and pointed out to the developer.

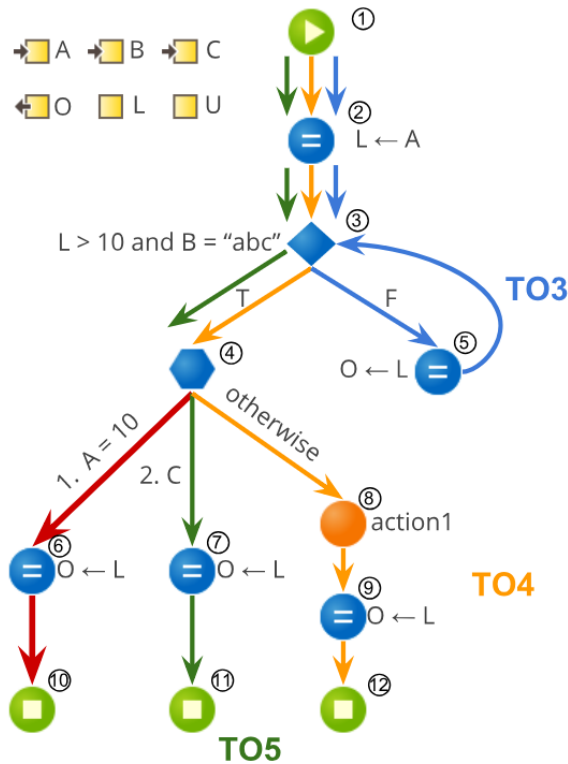


Figure 4.24: Dead code and paths covered by each test object.

Example:

While calculating branch coverage (subsubsection 4.1.6.2), it was noticed that only ten out of the twelve branches in the graph are traversed. We are now going to identify those that are never reached, by comparing both the sets of the entire branches covered in the paths traversed by the test objects, and the ones defined by the graph.

The three test objects checked out by the end of traversal are able to cover the following branches: ($\{1 \rightarrow 2\}$, $\{2 \rightarrow 3\}$, $\{3 \rightarrow 4\}$, $\{3 \rightarrow 5\}$, $\{4 \rightarrow 8\}$, $\{4 \rightarrow 7\}$, $\{5 \rightarrow 3\}$, $\{7 \rightarrow 11\}$, $\{8 \rightarrow 9\}$, $\{9 \rightarrow 12\}$). Comparing that to the set of all branches defined, we can see that $\{4 \rightarrow 6\}$ and $\{6 \rightarrow 10\}$ are never reached. Looking closely at the procedure (Figure 4.1), we can see that those branches represent the outgoing path from the decision $A = 10$, that we had identified before would never be true and would thus never have a test object continuing traversal through that path.

Here we have confirmed that those branches are indeed dead code and can never be reached for any input combination. Figure 4.24 shows in the graph these branches alongside the path each test object does cover.

4.1.8.3 Cycles

As explained, since the algorithm cannot, currently, identify if a cycle is ever stopped or not, when one is found, a warning is produced so the developer knows and can analyse its conditions more carefully.

Cycles are detected by verifying, after processing a node, if its outgoing link points towards a node already processed by the current test object's path. If this is verified, that test object is marked as final and a warning is raised.

Example:

We've already identified a cycle in this example when the *Assign* node with identifier 5 had the outgoing link (5 → 3) which points to the *If* node that had already been processed by the test object that was currently traversing that path. Thus it was marked as a warning that from the False branch of the *If* node, a cycle exists.

4.1.9 Test case prioritization

The order by which the test cases are presented to the developer is of the utmost importance (see section 3.2). Thus, the final test suite generated is prioritized according to two criteria:

1. They are first organized in terms of the combined coverage they provide for both branches and nodes;
2. If the previous results in ties when multiple items have the same priority, the second criteria takes into account the number of decisions the path traversed by this test case encounters.

This prioritization is also complementary, meaning that when the first “best” test case is found, the second test case to be displayed is the one that, together with the first one, helps to cover more nodes and branches. The same goes for the third pick and so on. This means that the first x test cases presented are the ones that will cover the most nodes and branches and no other combination of x test cases will be able to cover more code.

Example:

In this example, as we had seen before, after graph traversal we were left with three test objects. For each of those, we'll start by calculating both the node and branch coverage they provide to the code.

We also know that the entire graph contains 12 nodes and 12 branches, so we will use that to calculate the following coverages. We have also seen the nodes and branches each test object checked out covers, in both Figure 4.19 and Figure 4.20, respectively.

Table 4.2 shows, for each test object, the percentage of nodes and branches they cover. These values are summed up in order to find out the test object that covers more nodes and branches. We can see that TO4 is the winner for this first step of the prioritization, and as there is no other test object with the same accumulated value, there is no need to also verify the number of conditions each path covers. If ties were detected, the tie breaker would be the number of decisions covered in the path traversed, if that would

also result in a tie, then it would be considered that the test objects involved in that tie offer the same risk to the code and the choice would be randomly applied.

Table 4.2: Values used to identify the test object who covers more code and will be the first presented to the developer.

Test object	Node coverage	Branch coverage	Total accum.
TO3	$\frac{4}{12} \approx 33\%$	$\frac{4}{12} \approx 33\%$	66
TO4	$\frac{7}{12} \approx 58\%$	$\frac{6}{12} = 50\%$	108
TO5	$\frac{6}{12} = 50\%$	$\frac{5}{12} \approx 41\%$	91

So, *TO4* is marked as the first test object. This means that the following path of the code is already covered: ($\{1 \rightarrow 2\}$, $\{2 \rightarrow 3\}$, $\{3 \rightarrow 4\}$, $\{4 \rightarrow 8\}$, $\{8 \rightarrow 9\}$, $\{9 \rightarrow 12\}$).

The next step is to now calculate the coverage that *TO3* and *TO5* offer to the code not yet covered by *TO4*. Table 4.3 shows the results for this step where it is visible that *TO5* covers most of the remaining code and is thus marked as the second most important test object. As we only have *TO3* left, that will be the third in the line.

Table 4.3: Results from the next step of prioritization.

Test object	Node coverage	Branch coverage	Total accum.
TO3	$\frac{1}{12} \approx 8\%$	$\frac{2}{12} \approx 16\%$	24
TO5	$\frac{2}{12} = 16\%$	$\frac{2}{12} \approx 16\%$	32

Finally, to actually present these results to the user, one randomly picked test case from the test object *TO4* will appear first, followed by one from *TO5* and then *TO3*. The remainder test cases from each test object will be randomly ordered. Figure 4.25 shows the order by which the code is covered.

4.1.10 Optimizations

Some optimizations have taken place in order to allow a better management of both memory and time resources. For this, there are two main mechanisms being employed:

- Parallelism: the work is being parallelised in an effort to reduce the time the algorithm runs;
- Cause-effect: with this methodology the number of test cases produced is reduced significantly, thus saving memory resources.

4.2 PoC with dummy model

A PoC was first developed in order to verify if the algorithm shown could be implemented over a model like OutSystems. For that, a simplified version of the model was created,

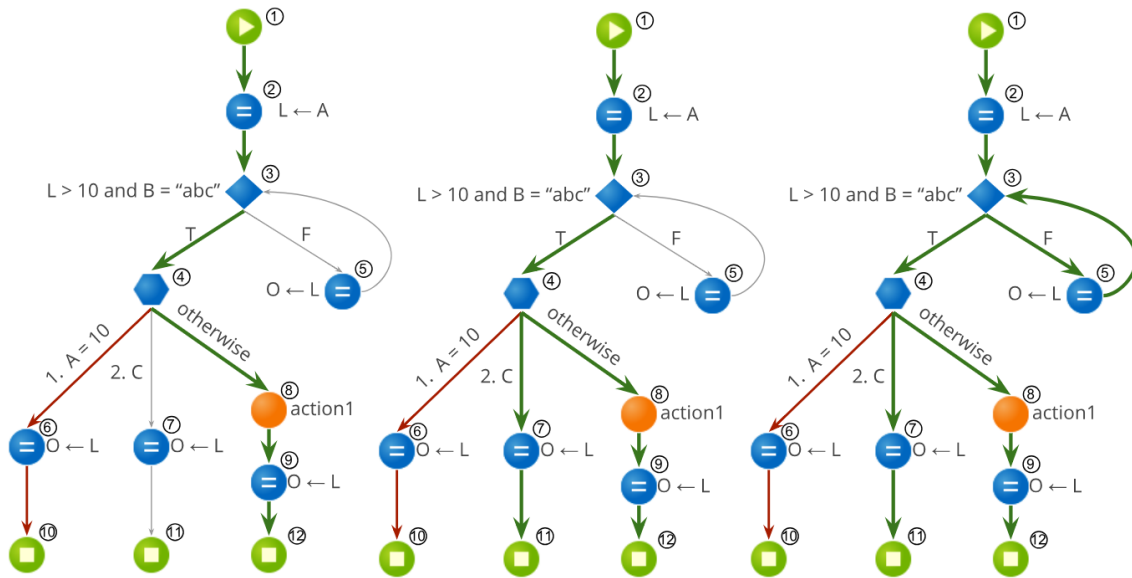


Figure 4.25: The order by which the code is covered if test cases from each path are executed sequentially from left to right.

```

<node type="START" key="1" label="Start" source="images/Start.ico" loc="0 0"/>
<node type="ASSIGN" key="2" label="L=A" assignments="L<-A" source="images/Assign.ico" loc="0 100"/>
<node type="IF" key="3" label="L > 10 and B = abc" condition="L > 10 and B = abc" outTrue="4" outFalse="5"
source="images/If.ico" loc="-42 200"/>
<node type="SWITCH" key="4" label="4" decisions="1->A=10 2->C" outDefault="8" outgoings="1->6 2->7"
source="images/Switch.ico" loc="-100 300"/>
<node type="ASSIGN" key="5" label="O=L" assignments="O<-L" source="images/Assign.ico" loc="100 300"/>
<node type="DEFAULT" key="8" label="action1" source="images/Default.ico" loc="0 420"/>
<node type="END" key="10" label="End" source="images/End.ico" loc="-200 560"/>
<node type="INPUT_VAR" key="A" var_type="int" label="A" source="images/InputParameter.ico" loc="-250 0"/>
<node type="LOCAL_VAR" key="L" var_type="int" label="L" source="images/LocalVariable.ico" loc="-200 50"/>
<node type="OUT_VAR" key="O" var_type="int" label="O" source="images/InputParameter.ico" loc="-250 50"/>
<link from="1" to="2" key="1->2"/>
<link from="3" to="4" key="3->4" label="True"/>
<link from="3" to="5" key="3->5" label="False"/>
<link from="4" to="6" key="4->6" label="A = 10"/>
<link from="4" to="7" key="4->7" label="2. C"/>
<link from="4" to="8" key="4->8" label="otherwise"/>

```

Figure 4.26: XML file snippet from the graph being used as example.

only containing the supported nodes.

This simple model consists of two objects: nodes and links. Each link connects two nodes and each node contains a set of information according to its type. Figure 4.26 shows a snippet of the XML file that models the graph used as example.

Figure 4.27 represents the structure for this PoC. It receives the XML representation of the model and converts it to Java objects with the Java Architecture for XML Binding (JAXB) [68] API. Then, having these objects, the algorithm executes. After traversing the graph and calculating the coverage criteria, Java Servlets [65] request the results from the algorithm's API and report them to the JavaServer Pages (JSP) [66] that instantiate the interface that developers can then interact with.

Figure 4.28 shows the user interface for this PoC, using the graph that has been used throughout this chapter, where:

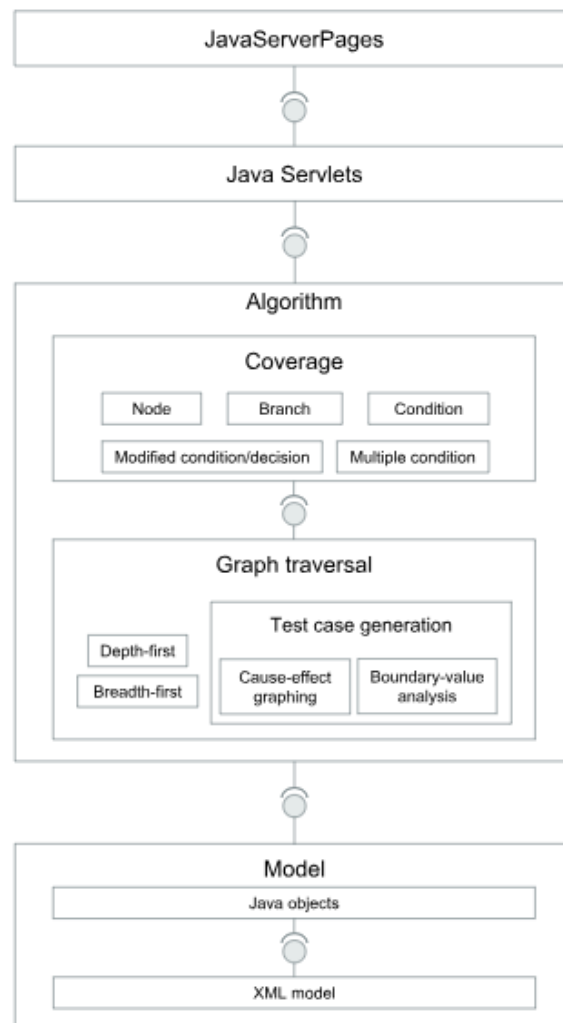


Figure 4.27: Structure of this PoC.

- (A) shows the procedure with unreachable branches highlighted in red. In green are highlighted the branches covered by the selected test cases (in this case they are all selected and thus all branches but the unreachable are marked as green);
- (B) allows the developer to choose the procedure it wishes to evaluate;
- (C) contains a selectable list of the input combinations generated (all selected at first);
- (D) presents the meaningful warnings;
- (E) complements the visual representation of the unreachable branches, as they can also be seen in a textual form;
- (F) depicts the results obtained for the multiple coverage criteria evaluated, according to the test selected cases;
- (G) the most complex criteria can be hidden from view.

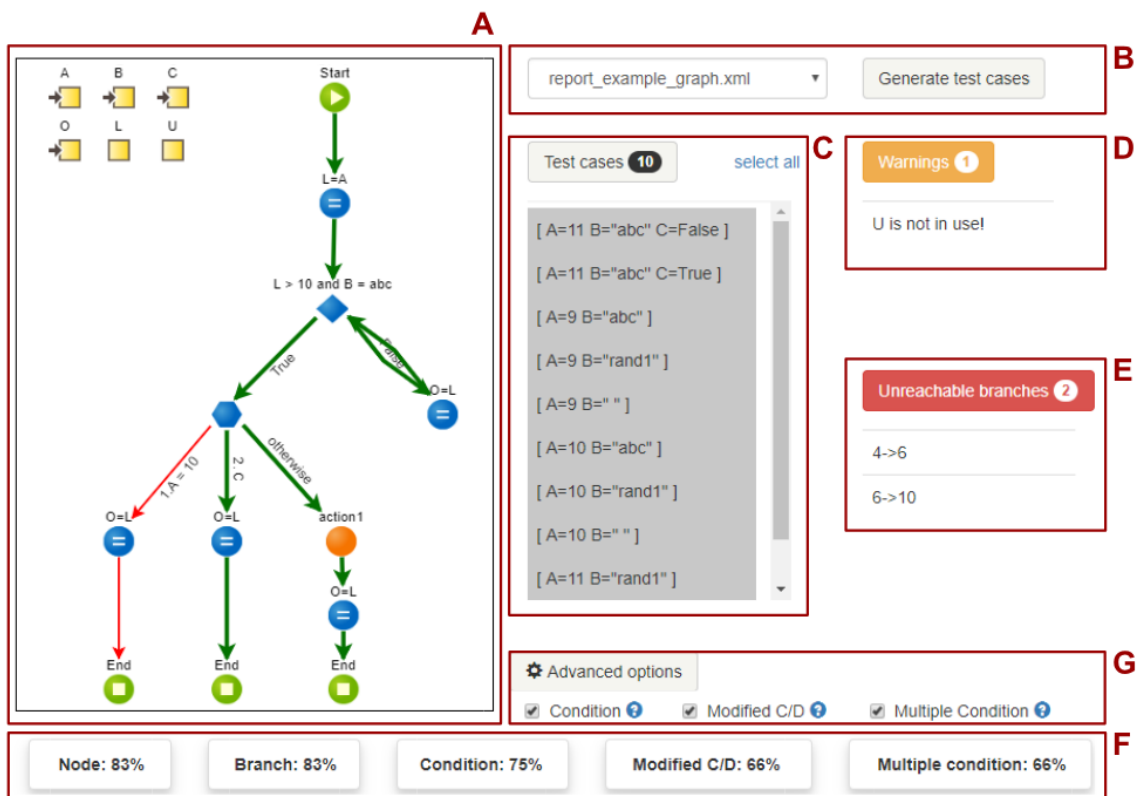


Figure 4.28: PoC interface screenshot.

Figure 4.28 also shows the results obtained from this example. This prototype did not include the expected output results feature, as it was only implemented later on, as part of the tool described in the following section.

4.3 Tool applied to the OutSystems model

This tool was implemented directly over the OutSystems development environment, *Service Studio*. Figure 4.29 shows the structure of this tool. The model exists in *Service Studio* intrinsically, and it is used as input by the tool. The algorithm, implemented in C#, begins its execution and the final results are presented in a dialog, which is a view built upon web technologies, in particular, HTML, Typescript (incl. React) and CSS.

A paper [15] further describing this tool can be seen in Appendix B and a video (available [here](#)¹) presents a screen-cast of a execution scenario.

Figure 4.31 a) shows the user interface for this tool, presenting the results for the example used through this chapter, where the following fields correspond to:

- (A) generated input combinations. These initial combinations represent a minimum set of test cases necessary to reach maximum node and branch coverage, meaning that

¹<https://youtu.be/8GsY8NTNXdk>

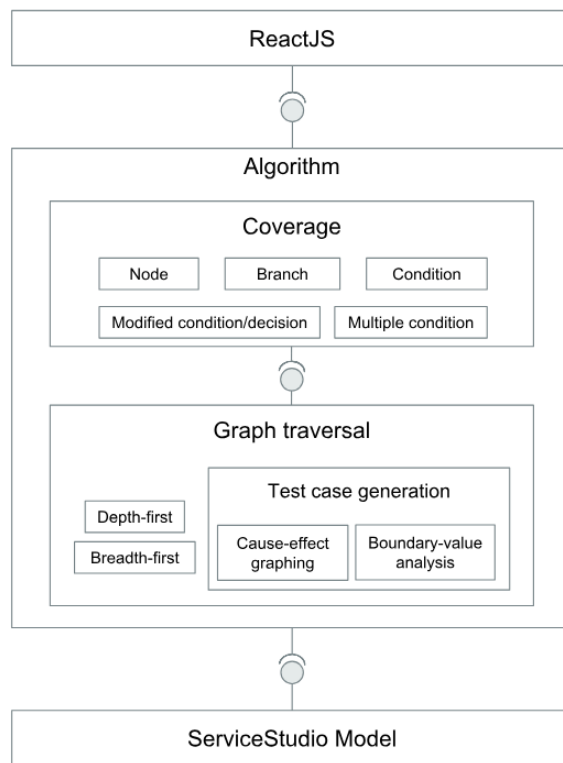


Figure 4.29: Structure of the tool implemented for ServiceStudio.

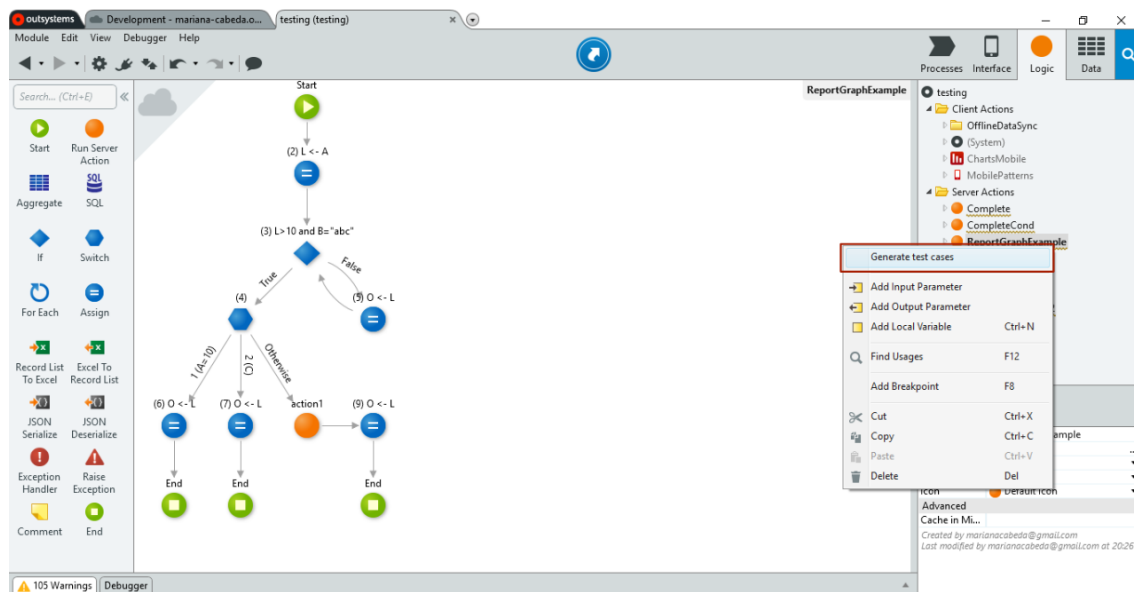


Figure 4.30: Screenshot of ServiceStudio where the command to call the tool can be seen.

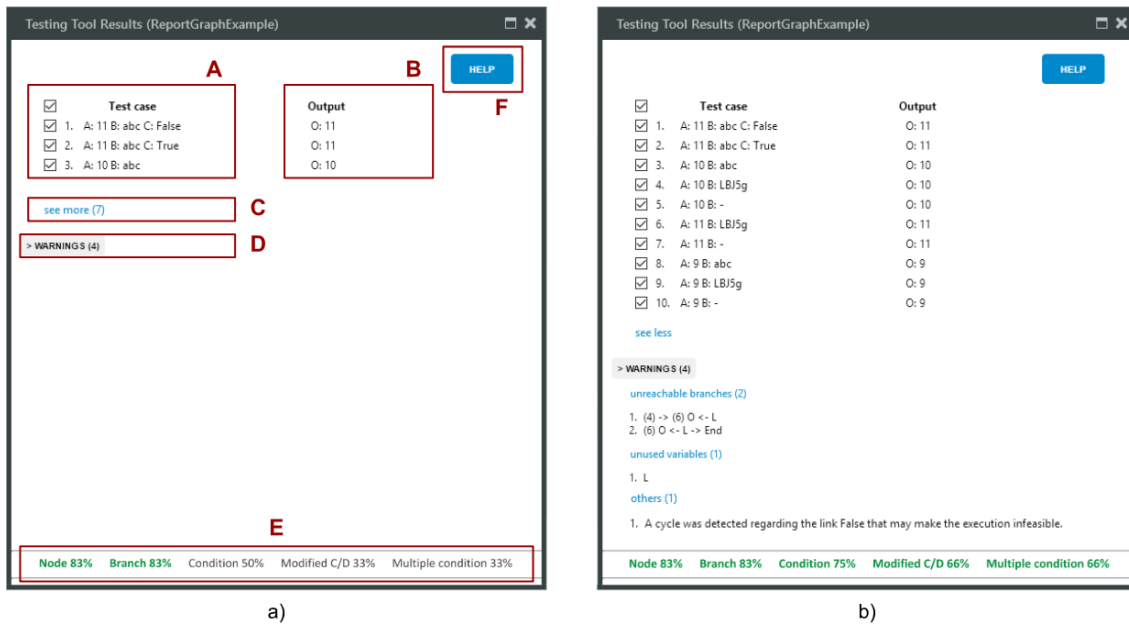


Figure 4.31: Tool window with results from this example. a) shows the initial state of the window and b) after expanding all fields.

they represent one test case from each test object and are prioritized between each other according to the methodology explained in subsection 4.1.9;

- (B) the expected output variables for each input combination;
- (C) the remaining generated input combinations can be expanded by clicking over this field;
- (D) warnings found for the evaluated procedure;
- (E) the coverage obtained for the different criteria according to selected items. When a percentage is highlighted in green, it means that it is the highest value recorded for that criterion and selecting further items will never increase it;
- (F) a help button which opens a menu listing the [Frequently Asked Questions \(FAQ\)](#). These questions explain what each field of the window represents. It also clarifies some questions that were considered to be problematic while this tool was under usability tests (see section 5.2).

EVALUATION

In order to test this tool, two main tests were performed. First, the algorithm was executed over multiple graphs from where values such as the number of top test cases found, the total number of test cases generated and what would the total number of test cases be if cause-effect graphing was not implemented. Then, usability tests were also carried out by users with varying degrees of expertise in testing.

Both tests were applied over the tool developed for *Service Studio* (see [section 4.3](#)) and obtained results and feedback are now discriminated.

5.1 Algorithm execution

The algorithm was executed over 24 graphs and the following data was retained:

- Total number of test cases generated by the algorithm;
- Number of “top test cases”, i.e., the set of minimum test cases necessary to reach all independent paths in the graph;
- Number of test cases that would be generated if not for the implementation of cause-effect graphing.

Each of the graphs is identified by its cyclomatic complexity, a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths¹ through a program’s source code and relates to the control graph of the program, indicating the risk level for each procedure [41, 56]. It can be computed using the control flow graph of the program with $CC = E - N + 2$, where CC is

¹Paths are linearly independent if every path contains an edge not included in any other path. Thus, linearly independent paths are a sub-set of all independent paths in a graph [31].

Cyclomatic Complexity	Risk Evaluation
1-10	Simple code Without much risk
11-20	More complex code Moderate risk
21-50	Complex code High risk
>50	Untestable Very high risk

Table 5.1: Risk evaluation according to cyclomatic complexity [41].

the cyclomatic complexity, E refers to the edges and N to the nodes [56]. Table 5.1 shows the risk levels associated with the cyclomatic complexity of a graph.

Figure 5.1 and Figure 5.2 show the comparison, for each graph, of the amount of generated test cases against the top test cases that are first presented to the user where, with a smaller amount of the test case suite, all independent paths in the graph can be tested. These figures show the incredible difference between the size of these sets and how helpful this feature can be for developers that, simply put, may not be able to test every single scenario for their applications. This smaller set of test cases can provide a very important assurance to the software, helping developers prioritizing the tests they execute.

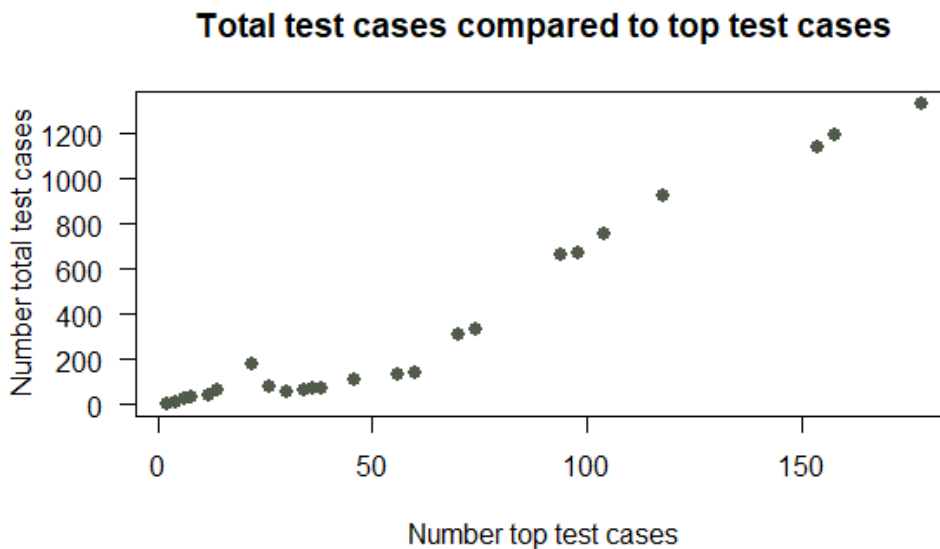


Figure 5.1: Scatter plot comparing the amount of top test cases presented against the total test cases generated by this tool.

Figure 5.3 presents the improvement in the number of test cases generated by applying cause-effect graphing. This mechanism allows us to minimize the number of test cases

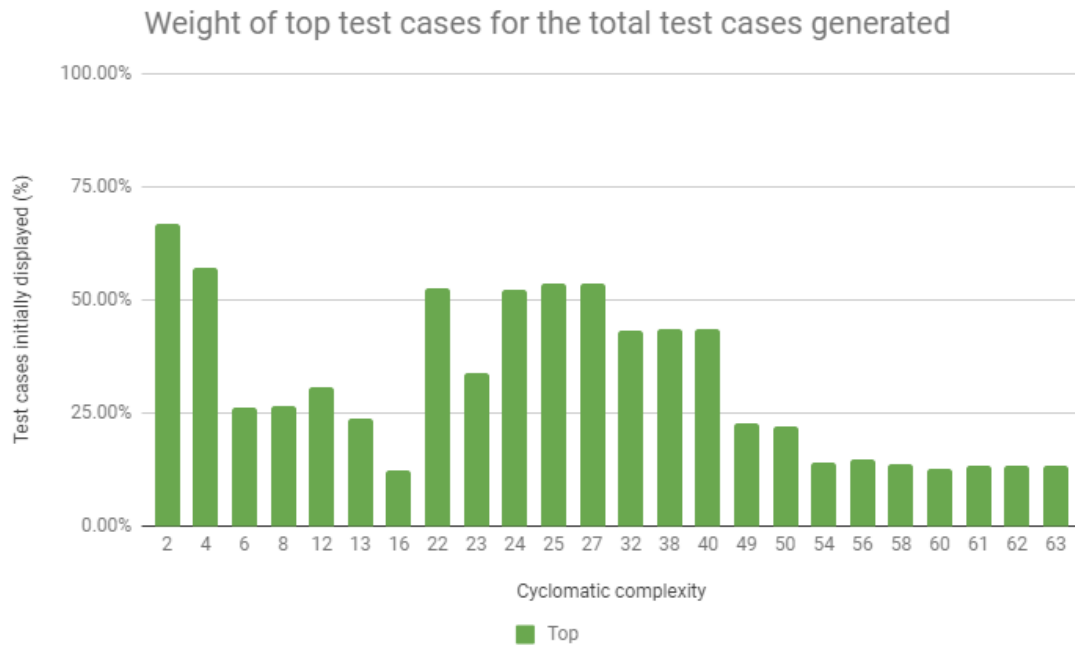


Figure 5.2: Percentage represented by the top test cases over the total amount of test cases generated.

generated, knowing that the remainder correspond to redundant situations that would not provide new information to the developer seeing as they would simply traverse paths with input combinations already contemplated in the smaller set of presented test cases.

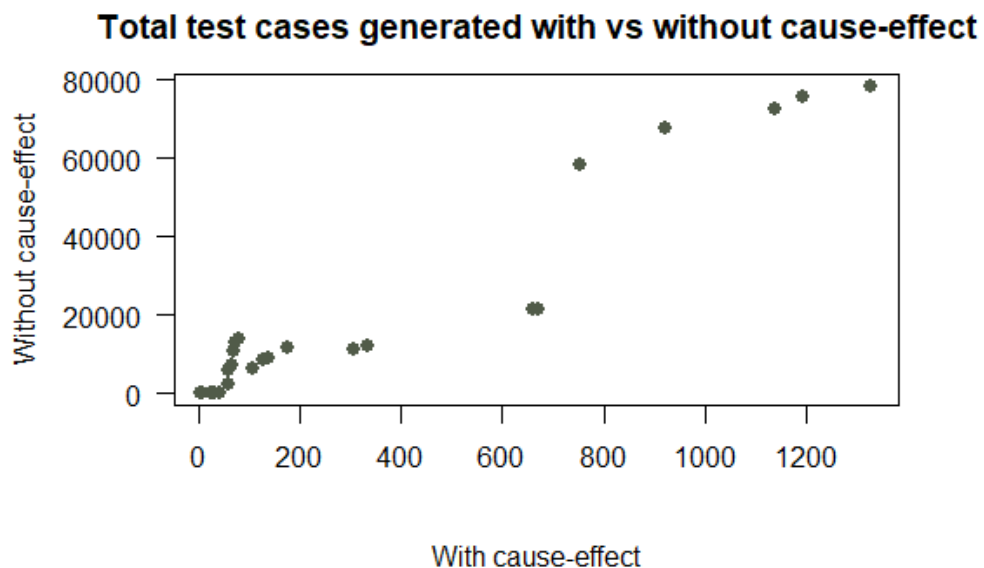


Figure 5.3: Scatter plot comparing the amount of total test cases generated for this tool against what would be generated if not for cause-effect graphing.

Without the application of cause-effect graphing, using the example presented in

Implementation (see Figure 4.1), after processing the *Switch* node with identifier 4, the following input combinations were generated:

$TC1 = [A = 9, B = "abc", C = True]$ $TC2 = [A = 9, B = "abc", C = False]$
 $TC3 = [A = 10, B = "abc", C = True]$ $TC4 = [A = 10, B = "abc", C = False]$
 $TC5 = [A = 11, B = "abc", C = True]$ $TC6 = [A = 11, B = "abc", C = False]$

$TC1, TC2, TC3$ and $TC4$ were eliminated due to cause-effect (to reach the *Switch* node, the decision $L > 10$ and $B = "abc"$ had to be met). Would those test cases not have been eliminated, we would have had generated four test cases that do not even reach the node that produced those combinations, as they would follow an alternative path that already had test cases generated for its boundary values (in this example, they would follow the *False* branch of the *If* node). In this example, we saved just four test cases, but as we can see in Figure 5.3, for larger examples, this can escalate pretty quickly.

More detailed results of these tests can be found in Appendix A.

5.2 Usability experiment

The usability of a product represents the interaction of the user with the system and can be accurately measured by assessing user performance, satisfaction and acceptability. For a software product, usability is the user's view of software quality [10].

To evaluate how the user interface performed when tested by users, usability tests took place for 40 different participants, 65% male and 35% female, ages between 21 and 30, with varying expertise in testing and each test took between 10 and 15 minutes.

The tests consisted in a two-part exercise where the user would receive a simple action in OutSystems and then attempt to answer some questions revolving two main scenarios: (1) manually calculating the results and (2) having the results provided by the tool available.

The questions included whether the user was able to identify a minimum set of input combinations that would cover each independent path in the graph, if he could order them by the amount of code covered, and others. Appendix A details the graphs used, the questions, expected answers and more detailed results for this test.

Table 5.2 lists the questions performed associated with the correct answer rate for both the original method (manual) and this tool. Figure 5.4 also compares the percentage of correct answers for both methods.

5.2.1 SUS

Complementing the comprehension questions, at the end of each usability test, the **System Usability Scale (SUS)** test was presented to the participants regarding the developed tool.

SUS [14] is a simple, ten item scale that offers a global view of subjective assessments of usability indicating the degree of agreement or disagreement with the statement on a five point scale (1-Strongly disagree, 5-Strongly agree).

Question	Correct rate (%)	
	Original	Tool
1. Identify a set of minimum test cases to cover all independent paths of this action.	2.5	90
2. What is the percentage of branches and nodes the provided input combinations offer to the code.	2.5	97.5
3. Can you order the set of minimum test cases defined by the code covered.	2.5	75
4. What is the expected value for the output variable according to the provided input combination?	100	100
5. Are there any possible errors in this trace of code that jump to sight?	30	97.5
6. What is the maximum node and branch coverage possible to obtain for this action?	32.5	97.5
7. Are there any paths unreachable? If so, which ones?	42.5	97.5

Table 5.2: For each question on the usability tests, the correct answer rate for both the original method (manual analysis) and for the tool.

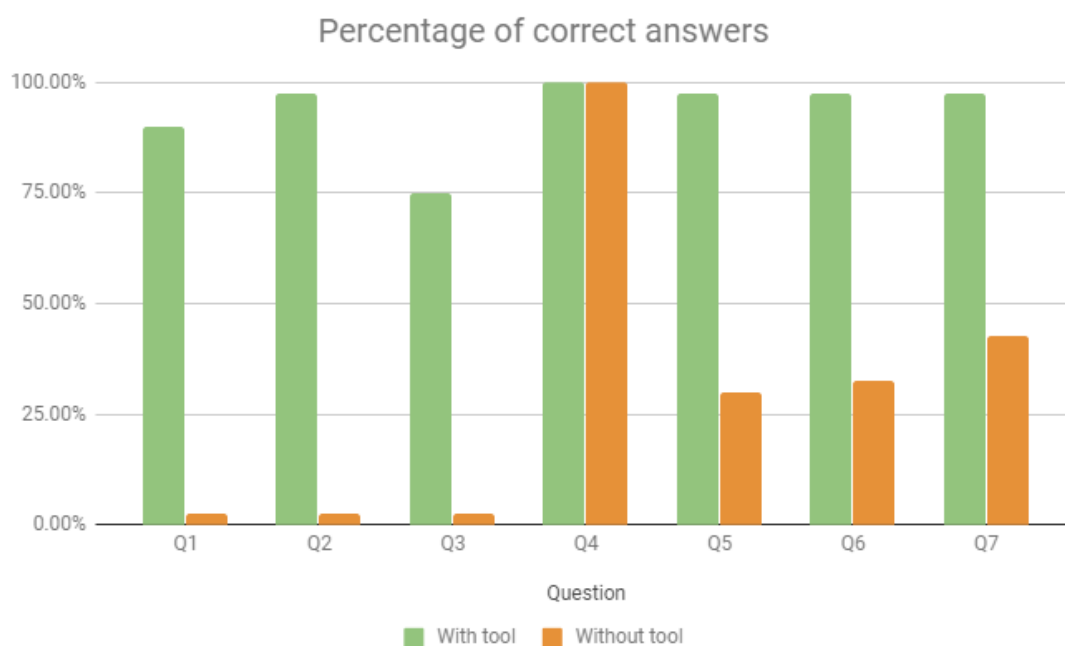


Figure 5.4: Percentage of correct answers for both methods.

Table 5.3 shows the meaning for SUS scores, that range between 0 and 100.

Score	Rating
<25	Worst imaginable Not acceptable
26-39	Poor Not acceptable
50-52	Ok Not acceptable
53-73	Good Marginal
74-85	Excelent Acceptable
86-100	Best imaginable Acceptable

Table 5.3: SUS meaning [7, 13].

Question	Mean answer
1. I think that I would like to use this system frequently.	4.6
2. I found the system unnecessarily complex.	1.175
3. I thought the system was easy to use.	4.45
4. I think that I would need the support of a technical person to be able to use this system.	1.05
5. I found the various functions in this system were well integrated.	4.45
6. I thought there was too much inconsistency in this system.	1.35
7. I would imagine that most people would learn to use this system very quickly.	4.35
8. I found the system very cumbersome to use.	1
9. I felt very confident using the system.	4.2
10. I needed to learn a lot of things before I could get going with this system.	1.625

Table 5.4: Mean SUS answer for each question.

Table 5.4 shows the mean answer for each question, Figure 5.5 the distribution of the results and Table 5.5 the descriptive statistics recorded, from which the mean SUS obtained can be analysed. Through Table 5.3, we can interpret that 90.875 is a very promising result.

N	Mean	Std. Dev.	Skew.	Kurt.	Shapiro-Wilk	
					W	p-value
40	90.875	6.138	-0.498	-0.227	0.947	0.060

Table 5.5: SUS descriptive statistics.

5.2.2 Results analysis

Throughout the usability tests, for the current method:

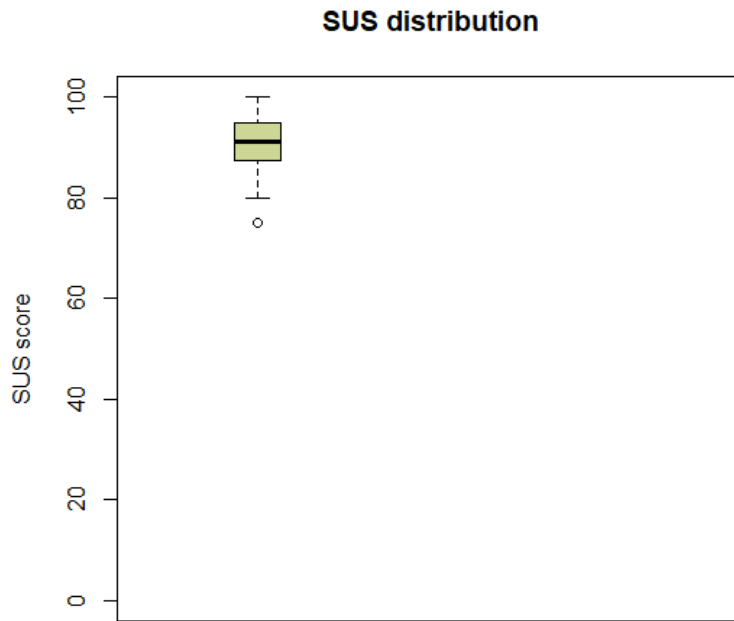


Figure 5.5: SUS distribution.

- The participants demonstrated enormous difficulties defining a minimum set of input combinations to cover all independent paths. In most cases they would define more tests than required (namely not thinking about cause-effect, they would simply try out all possible combinations) and a few defined an insufficient set;
- Also when confronted with the prioritization question, the first instinct was to, individually, see the nodes and branches each test case was covering, not taking into account the fact that some tests would cover partially the same path;
- The participants also had trouble identifying that unreachable branches existed in one of the graphs, and the unbreakable cycle in the other graph was also rarely identified;
- Overall, for the current method, the biggest complaint from the participants was on how time consuming and cumbersome it was to, manually, compute all those results.

For the tool developed:

- The main problems identified were the bar where the coverage percentages were presented, not standing out as much as it should, as some participants had difficulties finding it;
- Same happened for the button that displayed the warnings;
- Some users also did not immediately interpret that the presented test cases were the minimum set to reach all independent paths nor that they were already ordered;

- Overall, for most of the questions, the participants found the answer very quickly, and the immediate feedback from the tool was on how much time and how much effort this tool would minimize.

The final conclusions drawn from these tests tell us that the way the coverage is displayed needs to be improved, as well as the idea that the initial test cases displayed are a minimum set to cover all independent paths.

For the [SUS](#), the result obtained is extremely satisfying, reassuring that we are on the right path, not only for the interface design, but also for the importance and relevance of this work.

CONCLUSIONS

Software testing is extremely labor-intensive and expensive, accounting for 50% of the cost of software development [2, 49, 61] and one of the most difficult problems in testing is finding a test data selection strategy that is both valid and reliable [36].

In industry, the selection of test data is generally a manual process and is usually carried out by the tester. However, this practice is extremely costly, difficult and laborious. Automation in this area has been limited [57].

OutSystems provides a low-code application delivery platform that simplifies every stage of the app development and delivery process. OutSystems enables rapid, agile and continuous development, delivery and management of web and mobile applications [70].

The automation of the test case generation activity matches perfectly with OutSystems objectives, helping to further agilize the process of software development, bringing incredible value for OutSystems developers.

This work had the following objectives: ideation and development of an algorithm that generates the necessary test cases to reach traversable code in OutSystems applications' logic, as well as provide a coverage evaluation over these test cases and, finally, implement a PoC that would prove the viability of this solution.

All of the above were not only met but surpassed, seen as there is not only a PoC but also a running tool that already yields extremely interesting results. This tool also evaluates the generated test cases over a set of five coverage criterion, allowing their prioritization that determines the way they are displayed to the end user.

The graphs used throughout this thesis produced very interesting results regarding the optimization of the number of test cases that are generated as a whole. Not only that but it actually went further by presenting to the developer a set of minimum test cases that provide maximum node and branch coverage over the code, which can be of great advantage in terms of managing time and effort in test implementation.

The user feedback gathered during the usability experiment was also very encouraging. It was noticeable, that the tool had a very positive impact in the mood of the developers and in the way they would be more excited to approach the challenges knowing that they would already have a “helping hand”. Even though a couple of usability issues were raised, they can be easily fixed and re-integrated back into the tool. The bulk of the feedback was extremely positive.

6.1 Contributions

This dissertation complied the following contributions:

1. PoC with dummy model (section 4.2): provided the confirmation that the concept implemented over a generic model and language was viable to be ported to the real system and produce optimistic results;
2. Tool applied to the OutSystems model (section 4.3): even though its still a naive approach, seen as only a small sub-set of the OutSystems model is covered, this tool is already a very important stepping stone into bringing automated testing and coverage analysis to OutSystems developers (for both clients and in-house users);
3. Paper for the VL/HCC of 2018 [45] with title “Automated Test Generation Based on a Visual Language Applicational Model”: a paper [15] was submitted and accepted as a showpiece for the VL/HCC conference of 2018, offering an opportunity to showcase this tool to the VL/HCC community, helping us bring more exposure to the tool and the work hereby presented. This paper can be consulted in Appendix B.

6.2 Future work

Even though the expectations originally put on this work were already surpassed, due to the complexity and magnitude of the OutSystems model, as well as the testing activity itself being quite cumbersome, important aspects were yet not covered within the time allocated for this work and are now depicted in greater detail.

1. The “boundary dependency problem”: so far, the work presented relies on the boundary-value analysis to identify the values we want to generate. Subsection 4.1.3 described some of the issues that cannot be surpassed based solemnly on this methodology. For example, let’s say we have two input variables of the same type, A and B and the following decision is reached: $A = B$. For that, we would have to go to the input variable B and get its values to evaluate them for A . But what if B does not yet have any values? That can very easily happen and the solution in this work presented does not yet cover these situations. A possible solution would be to also incorporate error guessing (see section 2.2.4.2), where error-prone situations for a determinate

data type are evaluated. This way, for the example where B does not yet have any values, we can start by testing the most problematic ones;

2. Complex expressions: this work currently supports relatively simple expressions and more complex ones should also be taken into account such as algebraic expressions or external calls to procedures, amongst others. Seen as the main issue arising from these expressions is the “*boundary dependency problem*”, the same idea of adding more methodologies can also be taken into account;
3. Data types: more complex data types such as lists or objects would also be a very interesting addition to this tool;
4. Expanding the model: OutSystems model is vast and even though we started by tackling the logic behind user and server actions, much more is available through this model, such as designing [User Interface \(UI\)](#), business processes, data models, amongst others [72]. Supporting additional elements would bring incredible value to the developer;
5. Cycles: The current approach simply identifies that a cycle is present and signals it to the developer so that he can pay closer attention to it. A better approach would be to actually verify the boundaries of that cycle, what is the minimum times it can run, the maximum and if it is expected to break (recall in the example presented in [subsubsection 4.1.5.3](#), the cycle found would actually never break and thus the warning presented to the developer should reflect that);
6. Test case execution: the automated execution of the test cases themselves, by comparing the result obtained with the expected output would also be a fabulous addition to this tool;
7. Integration testing: this work is focused on unit testing, but for the future, integration testing could be a very interesting complement for this work, ensuring a more in-depth analysis for the developer.

BIBLIOGRAPHY

- [1] AgitarOne. *Agitator*. URL: http://www.agitar.com/solutions/products/software_agitation.html (visited on 02/13/2018).
- [2] D. S. Alberts. “The economics of software quality assurance.” In: *Proceedings of the June 7-10, 1976, national computer conference and exposition*. ACM. 1976, pp. 433–442.
- [3] J. G. W. Allen Kent. *Encyclopedia of Computer Science and Technology: Volume 32 - Supplement 17: Compiler Construction to Visualization and Quantification of Vortex-Dominated Flows*. 1nd. Marcel Dekker, 1995, pp. 284–287. ISBN: 9780824722852.
- [4] Alloy. *Alloy: a language and tool for relational models*. URL: <http://alloy.lcs.mit.edu/alloy/> (visited on 09/05/2018).
- [5] P. Ammann and J. Offutt. *Introduction to Software Testing*. 1st ed. New York, NY, USA: Cambridge University Press, 2008, pp. 27–51. ISBN: 978-0-511-39330-3.
- [6] T. Ball. “The Concept of Dynamic Analysis.” In: *SIGSOFT Softw. Eng. Notes* 24.6 (Oct. 1999), pp. 216–234. ISSN: 0163-5948. DOI: [10.1145/318774.318944](https://doi.org/10.1145/318774.318944). URL: <http://doi.acm.org/10.1145/318774.318944>.
- [7] A. Bangor, P. Kortum, and J. Miller. “Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale.” In: 4 (Apr. 2009), pp. 114–123.
- [8] L. Baresi and M. Pezzè. “An Introduction to Software Testing.” In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006). Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), pp. 89–111. ISSN: 1571-0661. DOI: <https://doi.org/10.1016/j.entcs.2005.12.014>. URL: <http://www.sciencedirect.com/science/article/pii/S1571066106000442>.
- [9] A. Bertolino and E Marchelli. “A Brief Essay on Software Testing.” In: *Software Engineering* 1 (2005).
- [10] N. Bevana, J. Kirakowskib, and J. Maissela. “What is Usability?” In: *4th International Conference on Human-Computer Interaction*. Stuttgart, Germany, 1991, pp. 30–39. DOI: [10.1109/QUATIC.2007.8](https://doi.org/10.1109/QUATIC.2007.8).

- [11] M. Boshernitsan, R. Doong, and A. Savoia. “From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing.” In: *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM. 2006, pp. 169–180.
- [12] C. Boyapati, S. Khurshid, and D. Marinov. “Korat: Automated Testing Based on Java Predicates.” In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. Roma, Italy: ACM, 2002, pp. 123–133. ISBN: 1-58113-562-9. DOI: [10.1145/566172.566191](https://doi.org/10.1145/566172.566191). URL: <http://doi.acm.org/10.1145/566172.566191>.
- [13] J. Brooke. “SUS: A Retrospective.” In: *Journal of Usability Studies*. Vol. 8. 2. 2013, pp. 29–40.
- [14] J. Brooke et al. “SUS-A quick and dirty usability scale.” In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.
- [15] M. Cabeda and P. Santos. “Automated Test Generation Based on a Visual Language Applicational Model.” In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 289–290. DOI: [10.1109/VLHCC.2018.8506582](https://doi.org/10.1109/VLHCC.2018.8506582).
- [16] C. Chouhan, V. Shrivastava, and P. S. Sodhi. “Test case generation based on activity diagram for mobile application.” In: *International Journal of Computer Applications* 57.23 (2012).
- [17] K. Claessen and J. Hughes. “Testing Monadic Code with QuickCheck.” In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 47–59. ISSN: 0362-1340. DOI: [10.1145/636517.636527](https://doi.org/10.1145/636517.636527). URL: <http://doi.acm.org/10.1145/636517.636527>.
- [18] K. Claessen and J. Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.” In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64. ISSN: 0362-1340. DOI: [10.1145/1988042.1988046](https://doi.org/10.1145/1988042.1988046). URL: <http://doi.acm.org/10.1145/1988042.1988046>.
- [19] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. *A Formal Evaluation of Data Flow Path Selection Criteria*. Tech. rep. Amherst, MA, USA, 1988, pp. 1318–1332. URL: http://www.ncstr1.org:8900/ncstr1/servlet/search?formname=detail&id=oai%3Ancstr1h%3Aumass_cs%3Ancstr1.umassa_cs%2F%2FUM-CS-1988-073.
- [20] W. W. W. Consortium. *eXtensible Markup Language (XML)*. URL: <https://www.w3.org/XML/> (visited on 02/02/2018).
- [21] S. Dalai, A. A. Acharya, and D. P. Mohapatra. “Test case generation for concurrent object-oriented systems using combinational UML models.” In: *Editorial Preface* 3.5 (2012). DOI: [10.14569/IJACSA.2012.030515](https://doi.org/10.14569/IJACSA.2012.030515).

- [22] S. Desai and S. Abhishek. *Software Testing : A Practical Approach*. second. PHI Learning Private Limited, Delhi, 2016, pp. 145–148. ISBN: 978-81-203-5226-1.
- [23] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs.” In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: [10.1007/BF01386390](https://doi.org/10.1007/BF01386390). URL: <http://dx.doi.org/10.1007/BF01386390>.
- [24] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-43287-0.
- [25] M. Ehmer and F. Khan. “A Comparative Study of White Box, Black Box and Grey Box Testing Techniques.” In: 3 (June 2012), pp. 12–15.
- [26] I. K. El-Far and J. A. Whittaker. “Model-based software testing.” In: *Encyclopedia of Software Engineering* (2002).
- [27] E. Çelik, S. Eren, E. Çini, and. Keleş. “Software test automation and a sample practice for an enterprise business software.” In: *2017 International Conference on Computer Science and Engineering (UBMK)*. 2017, pp. 141–144. DOI: [10.1109/UBMK.2017.8093583](https://doi.org/10.1109/UBMK.2017.8093583).
- [28] M. E. Fagan. “Advances in software inspections.” In: *IEEE Transactions on Software Engineering* SE-12.7 (1986), pp. 744–751. ISSN: 0098-5589. DOI: [10.1109/TSE.1986.6312976](https://doi.org/10.1109/TSE.1986.6312976).
- [29] M. E. Fagan. “Design and Code Inspections to Reduce Errors in Program Development.” In: *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*. Ed. by M. Broy and E. Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 301–334. ISBN: 978-3-642-48354-7. DOI: [10.1007/978-3-642-48354-7_13](https://doi.org/10.1007/978-3-642-48354-7_13). URL: https://doi.org/10.1007/978-3-642-48354-7_13.
- [30] L. Fernandez-Sanz and S. Misra. “Practical application of UML activity diagrams for the generation of test cases.” In: *Proceedings of the Romanian academy, Series A* 13.3 (2012), pp. 251–260.
- [31] D. Fotakis. “Congestion Games with Linearly Independent Paths: Convergence Time and Price of Anarchy.” In: *Theory of Computing Systems* 47.1 (2010), pp. 113–136. ISSN: 1433-0490. DOI: [10.1007/s00224-009-9205-7](https://doi.org/10.1007/s00224-009-9205-7). URL: <https://doi.org/10.1007/s00224-009-9205-7>.
- [32] M. Fowler and M. Foemmel. “Continuous integration.” In: *Thought-Works*) [http://www.thoughtworks.com/Continuous Integration.pdf](http://www.thoughtworks.com/Continuous%20Integration.pdf) 122 (2006), p. 14.
- [33] G. Fraser and A. Arcuri. “Whole test suite generation.” In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 276–291.

- [34] V. Garousi and M. V. Mäntylä. “When and What to Automate in Software Testing? A Multi-vocal Literature Review.” In: *Inf. Softw. Technol.* 76.C (Aug. 2016), pp. 92–117. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.04.015. URL: <http://dx.doi.org/10.1016/j.infsof.2016.04.015>.
- [35] P. Godefroid, N. Klarlund, and K. Sen. “DART: directed automated random testing.” In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.
- [36] J. B. Goodenough and S. L. Gerhart. “Toward a theory of test data selection.” In: *IEEE Transactions on Software Engineering SE-1.2* (1975), pp. 156–173. ISSN: 0098-5589. DOI: 10.1109/TSE.1975.6312836.
- [37] O. M. Group. *Object Constraint Language is available from Object Management Group’s web site*. URL: <http://www.omg.org> (visited on 02/02/2018).
- [38] R. Hamlet. “Random testing.” In: *Encyclopedia of software Engineering* (2002).
- [39] *haskell*. URL: <https://www.haskell.org/> (visited on 05/21/2018).
- [40] R. Hat. *JBoss*. URL: <http://www.jboss.org/> (visited on 01/15/2018).
- [41] I. Heitlager, T. Kuipers, and J. Visser. “A Practical Model for Measuring Maintainability.” In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8.
- [42] D. Hoffman. “Cost Benefits Analysis of Test Automation.” In: (1999).
- [43] J. Hopcroft and R. Tarjan. “Algorithm 447: Efficient Algorithms for Graph Manipulation.” In: *Commun. ACM* 16.6 (June 1973), pp. 372–378. ISSN: 0001-0782. DOI: 10.1145/362248.362272. URL: <http://doi.acm.org/10.1145/362248.362272>.
- [44] IBM. *Rational Rose*. URL: <https://www-03.ibm.com/software/products/pt/rosemod> (visited on 02/02/2018).
- [45] *IEEE Symposium on Visual Languages and Human-Centric Computing*. URL: <https://vlhcc18.github.io/> (visited on 08/28/2018).
- [46] D. Jackson, I. Schechter, and H. Shlyachter. “Alcoa: the alloy constraint analyzer.” In: *Proceedings of the 22nd international conference on Software engineering*. ACM. 2000, pp. 730–733.
- [47] D. Janzen and H. Saiedian. “Test-driven development concepts, taxonomy, and future direction.” In: *Computer* 38.9 (2005), pp. 43–50.
- [48] B. Jost, M. Ketterl, R. Budde, and T. Leimbach. “Graphical Programming Environments for Educational Robots: Open Roberta - Yet Another One?” In: *2014 IEEE International Symposium on Multimedia*. 2014, pp. 381–386. DOI: 10.1109/ISM.2014.24.

- [49] B. Korel. "Automated software test data generation." In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 870–879. ISSN: 0098-5589. DOI: 10.1109/32.57624.
- [50] P. T. Kshirasagar Naik. *Software testing and quality assurance: theory and practice*. first. Wiley, 2008, pp. 7–17, 96–101. ISBN: 978-0-471-78911-6.
- [51] P. A. Laplante. *What every engineer should know about software engineering*. CRC Press, 2007, p. 176. ISBN: 978-0-8493-7228-5.
- [52] A. Lima. *OutSystems Platform - Architecture and Infrastructure Overview*. Tech. rep. 2015. URL: <https://www.outsystems.com/home/document-download/178/8/0/0>.
- [53] N. Magic. *MagicDraw*. URL: <https://www.nomagic.com/products/magicdraw> (visited on 02/02/2018).
- [54] D. Marinov and S. Khurshid. "TestEra: A novel framework for automated testing of Java programs." In: *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE. 2001, pp. 22–31.
- [55] E. of mathematics. *Isomorphism*. URL: <http://www.encyclopediaofmath.org/index.php?title=Isomorphism&oldid=21572> (visited on 09/05/2018).
- [56] T. J. McCabe. "A complexity measure." In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320. DOI: 10.1109/tse.1976.233837.
- [57] P. McMinn. "Search-based Software Test Data Generation: A Survey." In: *Softw. Test. Verif. Reliab.* 14.2 (June 2004), pp. 105–156. ISSN: 0960-0833. DOI: 10.1002/stvr.v14:2. URL: <http://dx.doi.org/10.1002/stvr.v14:2>.
- [58] Microsoft. *ASP.NET*. URL: <https://www.asp.net> (visited on 01/15/2018).
- [59] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. "Korat: A Tool for Generating Structurally Complex Test Inputs." In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 771–774. DOI: 10.1109/ICSE.2007.48.
- [60] S. Mohanty, A. A. Acharya, and D. P. Mohapatra. "A survey on model based test case prioritization." In: *International Journal of Computer Science and Information Technologies* 2.3 (2011), pp. 1042–1047.
- [61] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN: 0471469122.
- [62] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 131–135. ISBN: 1-55860-467-7.
- [63] D. North. *Introducing BDD*. 2006. URL: <https://dannorth.net/introducing-bdd/> (visited on 02/15/2018).
- [64] Oracle. *Java Predicate*. Oracle. URL: <https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html> (visited on 03/29/2018).

- [65] Oracle. *Java Servlet Technology*. URL: <https://www.oracle.com/technetwork/java/index-jsp-135475.html> (visited on 08/28/2018).
- [66] Oracle. *JavaServer Pages Technology*. URL: <https://www.oracle.com/technetwork/java/javaee/jsp/index.html> (visited on 08/28/2018).
- [67] Oracle. *Oracle WebLogic*. URL: <https://www.oracle.com/middleware/weblogic/index.html> (visited on 01/15/2018).
- [68] E. Ort and B. Mehta. *Java Architecture for XML Binding (JAXB)*. 2003. URL: <http://www.oracle.com/technetwork/articles/javase/index-140168.html> (visited on 01/05/2018).
- [69] OutSystems. *Data*. URL: https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data (visited on 12/10/2018).
- [70] OutSystems. *Executive Overview of OutSystems*. URL: https://files.mtstatic.com/site_6602/3363/0?Expires=1536585167&Signature=X2CUZ731Ru1z6heI0WF~3MnmbfqHQjr654ZkKF~Ub9qtgCY--zL5vSjtYb-eXjh6yniUQVpgGIbJ4dz3M7s2v0ZKt3fEoXzXa4nG82FuXYJYJUPvYq&Key-Pair-Id=APKAJ5Y6AV4GI7A555NA (visited on 09/05/2018).
- [71] OutSystems. *How does OutSystems support testing and quality assurance?* URL: https://success.outsystems.com/Evaluation/Lifecycle_Management/9_How_does_OutSystems_support_testing_and_quality_assurance?origin=d (visited on 02/15/2018).
- [72] OutSystems. *How OutSystems solves the problem*. URL: https://success.outsystems.com/Evaluation/Why_OutSystems/02_How_does_OutSystems_solve_the_problems_of_app_delivery (visited on 09/05/2018).
- [73] OutSystems. *Logic*. URL: https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Logic (visited on 12/10/2018).
- [74] OutSystems. *Mobile Interfaces*. URL: https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Mobile_Interfaces?origin=d (visited on 12/10/2018).
- [75] OutSystems. *OutSystems Language*. URL: https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language (visited on 12/10/2018).
- [76] OutSystems. *OutSystems tools and components*. URL: https://success.outsystems.com/Evaluation/Architecture/1_OutSystems_Platform_tools_and_components (visited on 01/15/2018).
- [77] OutSystems. *Processes*. URL: https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Processes/Processes?origin=d (visited on 12/10/2018).
- [78] OutSystems. *TrueChange™*. URL: https://www.outsystems.com/help/ServiceStudio/9.0/app_life_cycle/TrueChange.htm (visited on 02/15/2018).

- [79] OutSystems. *Unit and regression testing with OutSystems*. URL: https://success.outsystems.com/Evaluation/Lifecycle_Management/9_How_does_OutSystems_support_testing_and_quality_assurance/Unit_and_regression_testing_with_OutSystems#Functional.2C_User_Interface.2C_Regression (visited on 02/15/2018).
- [80] OutSystems. *Web Interfaces*. URL: https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Web_Interfaces?origin=d (visited on 12/10/2018).
- [81] OutSystems. *OutSystems - Agile Methodology DataSheet*. Tech. rep. 2010. URL: <https://www.outsystems.com/home/downloadsdetail/25/75/>.
- [82] OutSystems. *OutByNumbers - Benchmark Overview Report*. Tech. rep. 2013. URL: <http://www.outsystems.com/res/OutbyNumbers-DataSheet>.
- [83] J. Proença. *BDDFramework*. OutSystems. URL: <https://www.outsystems.com/forge/component/1201/BDDFramework/> (visited on 02/15/2018).
- [84] P. Ramos. *Unit Test Framework*. OutSystems. 2018. URL: <https://www.outsystems.com/forge/component-details/387/Unit+Testing+Framework/> (visited on 02/15/2018).
- [85] S. Rapps and E. J. Weyuker. “Data Flow Analysis Techniques for Test Data Selection.” In: *Proceedings of the 6th International Conference on Software Engineering*. ICSE '82. Tokyo, Japan: IEEE Computer Society Press, 1982, pp. 272–278. URL: <http://dl.acm.org/citation.cfm?id=800254.807769>.
- [86] S. Rapps and E. J. Weyuker. “Selecting Software Test Data Using Data Flow Information.” In: *IEEE Trans. Softw. Eng.* 11.4 (Apr. 1985), pp. 367–375. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232226. URL: <http://dx.doi.org/10.1109/TSE.1985.232226>.
- [87] J. H. Reif. “Depth-first search is inherently sequential.” In: *Information Processing Letters* 20.5 (1985), pp. 229–234.
- [88] M. Revell. *What Is Visual Programming?* 2017. URL: <https://www.outsystems.com/blog/what-is-visual-programming.html> (visited on 01/14/2018).
- [89] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. “Test case prioritization: An empirical study.” In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.
- [90] R. Santos, C. C. de Magalhaes, J. Correia-Neto, F. Silva, and L. Capretz. “Would You Like to Motivate Software Testers? Ask Them How.” In: *Electrical and Computer Engineering Publications*. Vol. 114. Nov. 2017, pp. 95–104. DOI: 10.1109/ESEM.2017.16.

- [91] SAP. *BAPI Programming Guide*. URL: https://help.sap.com/saphelp_nw70/helpdata/en/e0/9eb2370f9cbe68e10000009b38f8cf/frameset.htm (visited on 01/15/2018).
- [92] K. Saravanan and E. P. C. Prasad. "Open Source Software Test Automation Tools: A Competitive Necessity." In: *Scholedge International Journal of Management and Development* 3.6 (2016), pp. 103–110. ISSN: 2394-3378. URL: <http://thescholedge.org/index.php/sijmd/article/view/320>.
- [93] M. Sarma, D. Kundu, and R. Mall. "Automatic test case generation from UML sequence diagram." In: *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on Advanced Computing and Communications*. IEEE, 2007, pp. 60–67. DOI: 10.1109/ADCOM.2007.68.
- [94] V. Sawant and K. Shah. "Automatic Generation of Test Cases from UML Models." In: *IJCA Proceedings on International Conference on Technology Systems and Management (ICTSM) 2* (2011), pp. 7–10.
- [95] SeleniumHQ. *Selenium*. URL: <http://www.seleniumhq.org/> (visited on 02/15/2018).
- [96] R. Seth and S. Anand. "Prioritization of Test Cases scenarios derived from UML Diagrams." In: *International Journal of Computer Applications (0975-8887)* 46.12 (2012).
- [97] T. Simões. *Visual Programming Is Unbelievable... Here's Why We Don't Believe In It*. 2015. URL: <https://www.outsystems.com/blog/visual-programming-is-unbelievable.html> (visited on 01/21/2018).
- [98] S. S. Skiena. *The Algorithm Design Manual*. 2nd. Springer Publishing Company, Incorporated, 2008, pp. 147–177. ISBN: 1848000693, 9781848000698.
- [99] C. Solis and X. Wang. "A study of the characteristics of behaviour driven development." In: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, 2011, pp. 383–387.
- [100] R. Tarjan. "Depth-first search and linear graph algorithms." In: *SIAM Journal on Computing* 1.2 (1972).
- [101] T. A. Team. *Test Automator*. OutSystems. 2015. URL: <https://www.outsystems.com/forge/component-details/82/Test+Automator/> (visited on 02/15/2018).
- [102] UML. *Unified Modeling Language (UML)*. URL: <http://www.uml.org/> (visited on 02/06/2018).
- [103] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.
- [104] C. Wenjing and X. Shenghong. "A Software Function Testing Method Based on Data Flow Graph." In: *2008 International Symposium on Information Science and Engineering*. Vol. 2. 2008, pp. 28–31. DOI: 10.1109/ISISE.2008.23.

- [105] E. J. Weyuker, T. J. Ostrand, J. Brophy, and R. Prasad. “Clearing a Career Path for Software Testers.” In: *IEEE Softw.* 17.2 (Mar. 2000), pp. 76–82. ISSN: 0740-7459. DOI: 10.1109/52.841696. URL: <http://dx.doi.org/10.1109/52.841696>.

D E T A I L E D T E S T R E S U L T S

This appendix details the results obtained for both the execution over multiple graphs and the performed usability tests.

A.1 Algorithm execution

Table A.1 shows the number of test cases identified for each tested graph, as well as the number of top test cases and what would have been the total test cases generated if not for the implementation of cause-effect graphing. Figure A.1, Figure A.2 and Figure A.3 show the same results in bar charts.

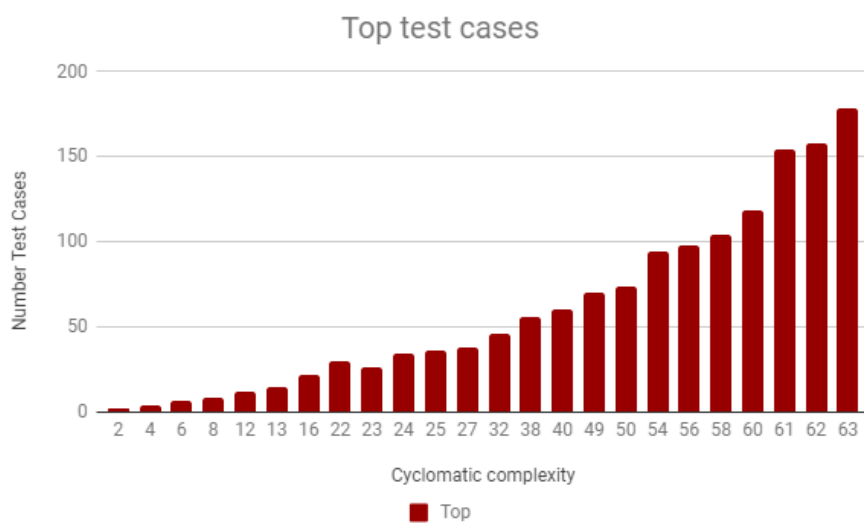


Figure A.1: Top test cases results.

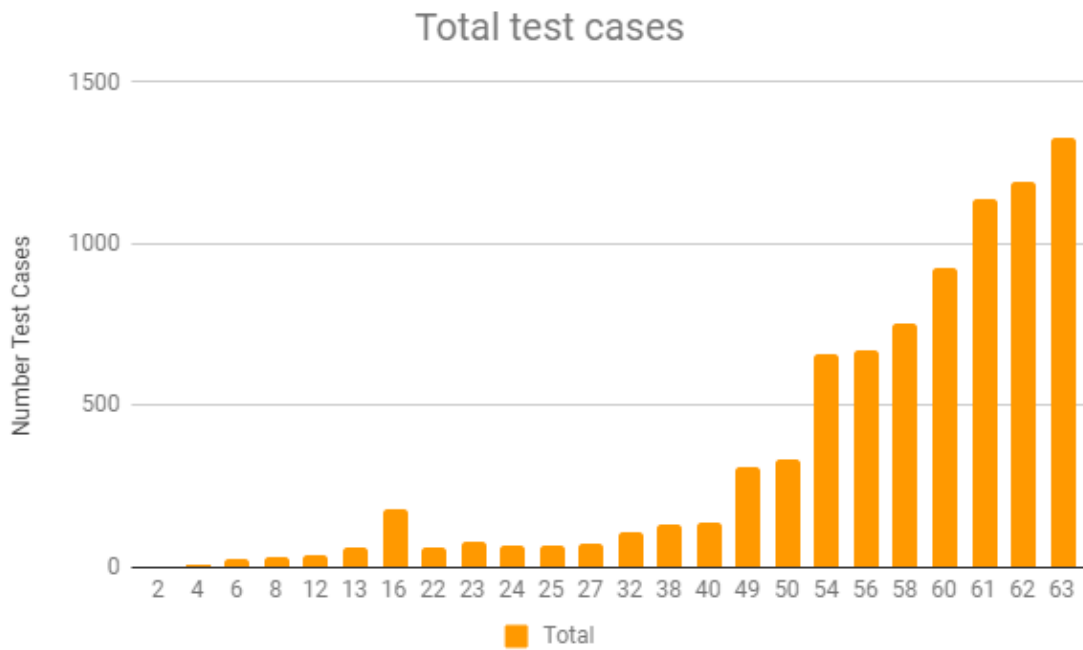


Figure A.2: Total test cases generated.

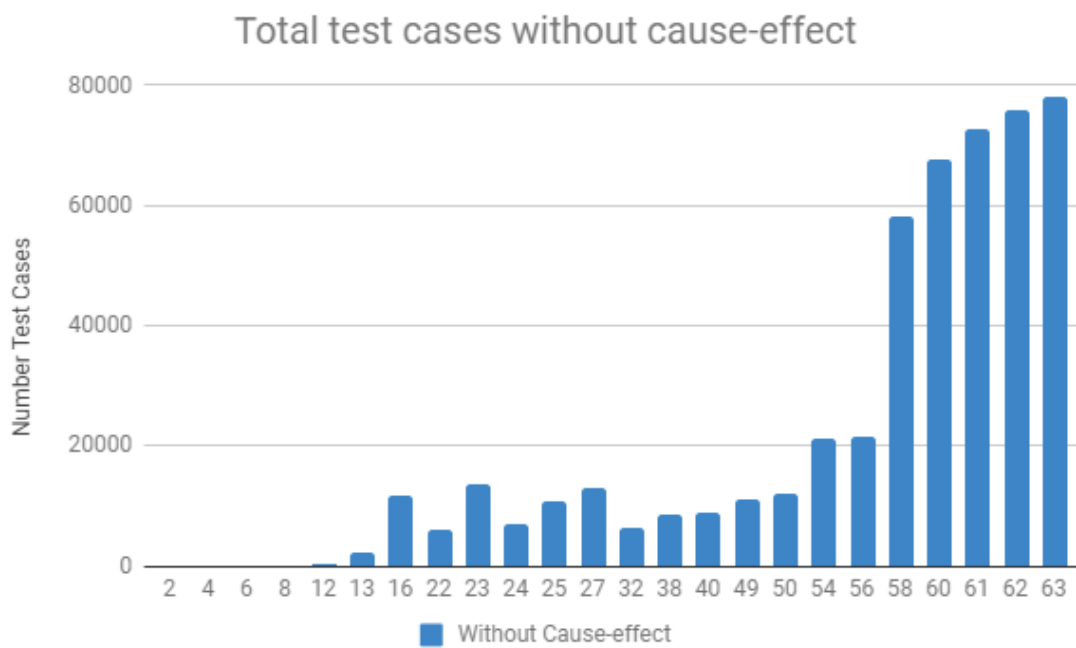


Figure A.3: Total test cases generated without cause-effect graphing.

CC	Top test cases	Total test cases	Without cause-effect	displayed
2	2	3	3	66%
4	4	7	7	57%
6	6	23	180	26%
8	8	30	183	26%
12	12	39	252	30%
13	14	59	2184	23%
16	22	176	11736	12%
22	30	57	5877	52%
23	26	77	13692	33%
24	34	65	7011	52%
25	36	67	10662	53%
27	37	71	13086	53%
32	46	106	6474	43%
38	56	128	8424	43%
40	60	138	8892	43%
49	70	306	11232	22%
50	74	334	12012	22%
54	94	660	21327	14%
56	98	670	21363	14%
58	104	754	58227	13%
60	118	922	67704	12%
61	154	1138	72618	13%
62	158	1194	75738	13%
63	178	1330	78234	13%

Table A.1: Detailed results obtained for each tested graph.

A.2 Usability experiment

This usability test consisted in two phases, where two different graphs would be used and the users would be confronted with seven questions regarding the data they had available. For one graph the user could use the tool, for the other, he could not and had to manually produce the results.

A.3 Graph and questions

A.3.1 Graph A

Figure A.4 shows Graph A and next follow the questions and respective expected answers for this graph.

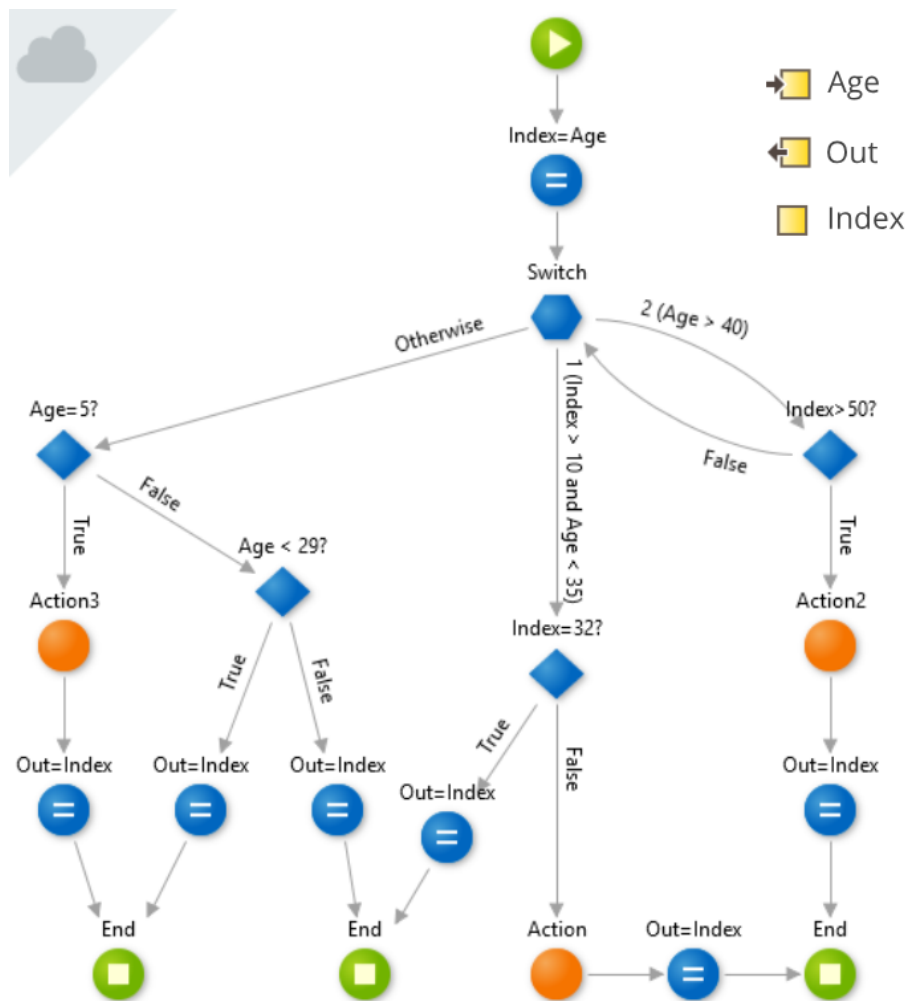


Figure A.4: Graph A.

1. Identify a set of minimum test cases to cover all independent paths of this action.

Answer: Age=10; Age=11; Age=51; Age=5; Age=35; Age=32; Age=41, 7 independent paths for this graph (this is one possible answer, others might be considered).

2. What is the percentage of branches the following test cases provide to the code:

- a) Age = 5
- b) Age = 11
- c) Age = 32

Answer: 45%.

3. Can you order the set of minimum test cases defined by the code covered?

Answer: Same order as in 1.

4. What is the expected value for the output variable "Out" for the following input combination?

a) Age = 51

Answer: Out = 51.

5. Are there any possible errors in this trace of code that jump to sight?

Answer: Yes, there is a cycle regarding the link 2(Age>40) that may make the execution infeasible.

6. What is the maximum node and branch coverage possible to obtain for this action?

Answer: Node = 100%, Branch = 100%.

7. Are there any paths unreachable? If so, which ones?

Answer: There are no paths unreachable in this graph.

A.3.2 Graph B

Figure A.5 shows Graph B and next follow the questions and respective expected answers for this graph.

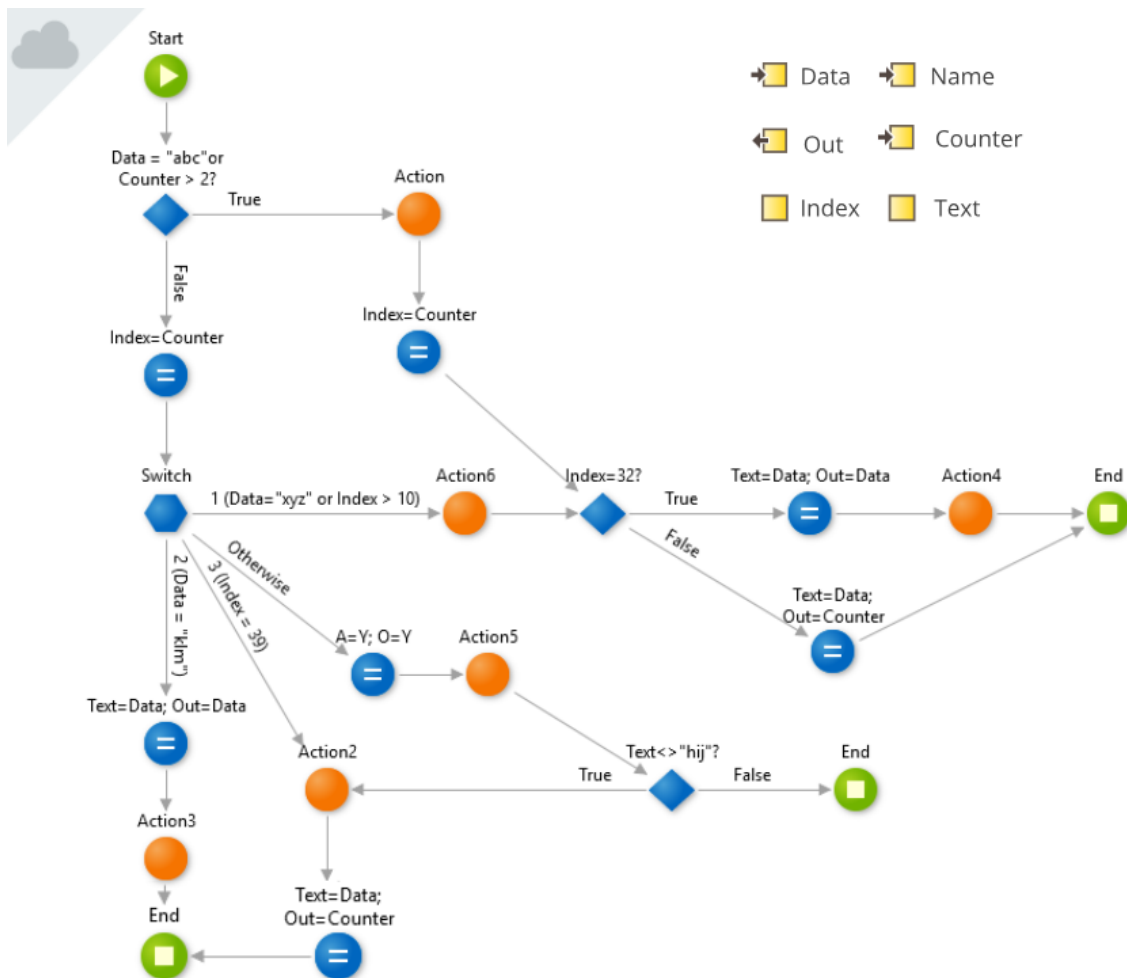


Figure A.5: Graph B.

1. Identify a set of minimum test cases to cover all independent paths of this action.

Answer: Data=rand1, Counter=2; Data=abc, Counter=32; Data=xyz, Counter=2; Data=klm, Counter=2; Data=hij, Counter=2; Data=abc, Counter=2, 6 independent paths for this graph (this is one possible answer, others might be considered).

2. What is the percentage of node the following test cases provide to the code:

a) Data = "abc", Counter = 3

b) Data = "klm", Counter = 2

Answer: 57%.

3. Can you order the set of minimum test cases defined by the code covered?

Answer: Same order as in 1.

4. What is the expected value for the output variable "Out" for the following input combination?

a) Data = "abc", Counter = 32

Answer: Out = 32.

5. Are there any possible errors in this trace of code that jump to sight?

Answer: Yes, there is an unreachable branch from the Switch node to Action2 ({Switch → Action2}).

6. What is the maximum node and branch coverage possible to obtain for this action?

Answer: Node = 100%, Branch = 95%.

7. Are there any paths unreachable? If so, which ones?

Answer: Yes, the branch {Switch → Action2} is unreachable.

DOCUMENTS REFERENCED

Table B.1 contains all the documents utilized in the making of this report, indicating its title, authors, year of publication, the main topics it refers and the chapters such document is cited on. The topics are represented by labels: (T) Testing, (A) Automatic Testing, (C) Coverage Criteria, (G) Graphs, (O) OutSystems, (Te) Technologies, (R) Related Work.

Table B.1: List of documents utilized in the making of this report.

	Title	Authors	Year	Topics						Chapters								
				T	A	C	G	O	Te	R	1	2	3	4	5	6		
[92]	Open Source Software Test Automation Tools: A Competitive Necessity	K. Saravanan, E. P. C. Prasa	2016															
[104]	A Software Function Testing Method Based on Data Flow Graph	C. Wenjing, X. Shenghong	2008															
[105]	Clearing a Career Path for Software Testers	Weyuker, Ostrand, Brophy, Prasad	2000															
[23]	A Note on Two Problems in Connexion with Graphs	E. W. Dijkstra	1959															

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work

Book
Paper
Online doc

	Title	Authors	Year	Topics							Chapters								
				T	A	C	G	O	Te	R	1	2	3	4	5	6			
[48]	Graphical Programming Environments for Educational Robots	B. Jost, Ketterl, Budde, Leimbac	2014																
[88]	What Is Visual Programming?	M. Revell	2017																
[81]	OutSystems - Agile Methodology DataSheet	OutSystems	2010																
[24]	Automated Software Testing: Introduction, Management, and Performance	E. Dustin, J. Rashka, J. Paul	1999																
[15]	Automated Test Generation Based on a Visual Language Applicational Model	M. Cabeda, P. Santos	2018																
[61]	The Art of Software Testing	G. J. Myers, C. Sandler	2004																
[34]	When and What to Automate in Software Testing? A Multi-vocal Literature Review	V. Garousi, M. V. Mäntylä	2016																
[27]	Software test automation and a sample practice for an enterprise business software	E. Çelik, S. Eren, E. Çini, Ö. Keleş	2017																
[8]	An Introduction to Software Testing	L. Baresi, M. Pezz	2006																
[50]	Software testing and quality assurance: theory and practice	P. T. Kshirasagar Naik	2008																
[5]	Introduction to Software Testing	P. Ammann, J. Offutt	2008																
[9]	Brief Essay on Software Testing	A. Bertolino E Marchelli	2005																
[22]	Software Testing: A Practical Approach	S. Desai S. Abhishek	2016																
[85]	Data Flow Analysis Techniques for Test Data Selection	S. Rapps E. J. Weyuker	1982																

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work
Book Paper Online doc

	Title	Authors	Year	Topics						Chapters								
				T	A	C	G	O	Te	R	1	2	3	4	5	6		
[86]	Selecting Software Test Data Using Data Flow Information	S. Rapps E. J. Weyuker	1985															
[19]	A Formal Evaluation of Data Flow Path Selection Criteria	Clarke, Podgurski, Richardson, Zeil	1988															
[98]	The Algorithm Design Manual	S. S. Skiena	2008															
[6]	The Concept of Dynamic Analysis	T. Ball	1999															
[3]	Encyclopedia of Computer Science and Technology	J. G. W. Allen Kent	1995															
[62]	Artificial Intelligence: A New Synthesis	N. J. Nilsson	1998															
[82]	OutByNumbers - Benchmark Overview Report	OutSystems	2013															
[52]	OutSystems Platform - Architecture and Infrastructure Overview	A. Lima	2015															
[40]	JBoss	R. Hat	-															
[67]	Oracle WebLogic	Oracle	-															
[76]	OutSystems tools and components	OutSystems	-															
[58]	ASP.NET	Microsoft	-															
[91]	BAPI Programming Guide	SAP	-															
[25]	A Comparative Study of White Box, Black Box and Grey Box Testing Techniques	M. Ehmer, F. Khan	2012															
[42]	Cost Benefits Analysis of Test Automation	D. Hoffman	1999															
[28]	Advances in software inspections	M. E. Fagan	1986															

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work

Book Paper Online doc

	Title	Authors	Year	Topics						Chapters							
				T	A	C	G	O	Te	R	1	2	3	4	5	6	
[29]	Design and Code Inspections to Reduce Errors in Program Development	M. E. Fagan	2001	■								■					
[97]	Visual Programming Is Unbelievable... Here's Why We Don't Believe In It	T. Simões	2015					■	■				■				
[43]	Efficient Algorithms for Graph Manipulation	J. Hopcroft, R. Tarja	1973				■					■					
[100]	Depth-first search and linear graph algorithms	R. Tarjan	1972				■					■					
[87]	Depth-first search is inherently sequential	J. Reif	1985				■					■					
[83]	BDD Framework	OutSystems	-					■	■			■					
[78]	TrueChange	OutSystems	-					■	■			■					
[84]	Unit Test Framework	P. Ramos	2018					■	■			■					
[79]	Unit and regression testing with OutSystems	OutSystems	-					■	■			■					
[63]	Introducing BDD	D. North	2006						■			■					
[47]	Test-driven development concepts, taxonomy, and future direction	D. Janzen H. Saiedian	2005						■			■					
[99]	A study of the characteristics of behaviour driven development	C. Solis X. Wang	2011						■			■					
[95]	Selenium	SeleniumHQ	-						■			■					
[101]	Test Automator	T. A. Team	2015						■			■					
[71]	How does OutSystems support testing and quality assurance?	OutSystems	-					■				■					

(T) Testing

(A) Automatic Testing

(C) Coverage Criteria

(G) Graphs

(O) OutSystems

(Te) Technologies

(R) Related Work

Book

Paper

Online doc

	Title	Authors	Year	Topics							Chapters					
				T	A	C	G	O	Te	R	1	2	3	4	5	6
[94]	Automatic Generation of Test Cases from UML Models	V. Sawant, K. Shah	2011													
[37]	Object Constraint Language	O. M. Group	-													
[53]	MagicDraw	N. Magic	-													
[44]	Rational Rose	IBM	-													
[20]	eXtensible Markup Language (XML)	W. W. W. Consortium	-													
[21]	Test case generation for concurrent object-oriented systems using combinational UML models	S. Dalai, A. A. Acharya, D. P. Mohapatra	2012													
[93]	Automatic test case generation from UML sequence diagram	M. Sarma, D. Kundu, R. Mall	2007													
[16]	Test case generation based on activity diagram for mobile application	Chouhan, Shrivastava, Sodh	2012													
[56]	A complexity measure	T. J. McCabe	1976													
[51]	What every engineer should know about software engineering	P. A. Laplante	2007													
[102]	Unified Modeling Language (UML)	UML	-													
[11]	From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing	Boshernitsan, Doong, Savoia	2006													
[96]	Prioritization of Test Cases scenarios derived from UML Diagrams	R. Seth S. Anand	2012													

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work

Book Paper Online doc

	Title	Authors	Year	Topics							Chapters					
				T	A	C	G	O	Te	R	1	2	3	4	5	6
[33]	Whole test suite generation	G. Fraser A. Arcuri	2013													
[30]	Practical application of UML activity diagrams for the generation of test cases	Fernandez-Sanz Misra	2012													
[1]	Agitator	AgitarOne	-													
[12]	Korat: Automated Testing Based on Java Predicates	C. Boyapati, S. Khurshid, D. Marinov	2002													
[64]	Java Predicate	Oracle	-													
[54]	TestEra: A novel framework for automated testing of Java programs	D. Marinov, S. Khurshid	2001													
[17]	Testing Monadic Code with QuickCheck	K. Claessen, J. Hughes	2002													
[18]	QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs	K. Claessen, J. Hughes	2011													
[39]	Haskell	-	-													
[59]	Korat: A Tool for Generating Structurally Complex Test Inputs	A. Milicevic, S. Misailovic, D. Marinov, S. Khurshid	2007													
[4]	Alloy: a language and tool for relational models	-	-													
[55]	Isomorphism	Encyclopedia of mathematics	-													
[38]	Random testing	R. Hamlet	2002													
[35]	DART: directed automated random testing	P. Godefroid, N. Klarlund, K. Sen	2005													
[103]	Practical model-based testing: a tools approach	M. Utting, B. Legiard	2010													

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work

Book Paper Online doc

	Title	Authors	Year	Topics							Chapters							
				T	A	C	G	O	Te	R	1	2	3	4	5	6		
[26]	Model-based software testing	I. El-Far, J. Whittaker	2002															
[46]	Alcoa: the alloy constraint analyzer	D. Jackson, I. Schechter, H. Shlyahter	2000															
[68]	Java Architecture for XML Binding (JAXB)	-	-															
[65]	Java Servlet Technology	Oracle	-															
[66]	JavaServer Pages Technology	Oracle	-															
[41]	A Practical Model for Measuring Maintainability	I. Heitlager, T. Kuipers, J. Visser	2007															
[31]	Congestion Games with Linearly Independent Paths: Convergence Time and Price of Anarchy	Fotakis, Dimitris	2010															
[14]	SUS-A quick and dirty usability scale	J. Brooke	1996															
[10]	What is Usability?	N. Bevana, J. Kirakowskib, J. Maissela	1991															
[13]	SUS: A Retrospective	J. Brooke	2013															
[7]	Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale	A. Bangor, P. Kortum, J. Miller	2009															
[45]	IEEE Symposium on Visual Languages and Human-Centric Computing	-	-															
[49]	Automated software test data generation	B. Korel	1990															
[2]	The economics of software quality assurance	D. Alberts	1976															

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work

Book Paper Online doc

	Title	Authors	Year	Topics							Chapters					
				T	A	C	G	O	Te	R	1	2	3	4	5	6
[36]	Toward a theory of test data selection	J. Goode-nough, S. Gerhart	1975	■												■
[57]	Search-based Software Test Data Generation: A Survey	P. McMinn	2004	■	■											■
[70]	Executive Overview of OutSystems	OutSystems	-					■								■
[72]	How OutSystems solves the problem	OutSystems	-					■								■
[90]	Would You Like to Motivate Software Testers? Ask Them How	Santos, Magalhaes, Correia-Neto, Silva, Capretz	2017	■							■					
[32]	Continuous integration	M. Fowler M. Foemmel	2006					■				■				
[60]	A survey on model based test case prioritization	S. Mohanty, A. Acharya, D. Mohapatra	2011		■					■			■			
[89]	Test case prioritization: An empirical study	G. Rothermel, R. Untch, C. Chu, M. Harold	1999		■					■			■			

(T) Testing (A) Automatic Testing (C) Coverage Criteria (G) Graphs
(O) OutSystems (Te) Technologies (R) Related Work

Book Paper Online doc



Automated Test Generation Based on a Visual Language Applicational Model

Mariana Cabeda

FCT/UNL

Lisboa, Portugal

Email: m.cabeda@campus.fct.unl.pt

Pedro Santos

OutSystems

Linda-a-Velha, Portugal

Email: pedro.santos@outsystems.com

Abstract—This showpiece presents a tool that aids OutSystems developers in the task of generating test suites for their applications in an efficient and effective manner. The OutSystems language is a visual language graphically represented through a graph that this tool will traverse in order to generate test cases.

The tool is able to generate and present to the developer, in an automated manner, the various input combinations needed to reach maximum code coverage, offering a coverage evaluation according to a set of coverage criteria: node, branch, condition, modified condition-decision and multiple condition coverage.

Index Terms—Software test automation, Software test coverage, OutSystems language, Visual Programming Language, OutSystems applicational model.

I. DESCRIPTION

A. Introduction

The OutSystems [1] language, classified under Visual Programming Languages (VPLs), allows developers to create software visually by drawing interaction flows, UIs and the relationships between objects. Low-code tools reduce the

complexity of software development bringing us to a world where a single developer can create rich and complex systems in an agile way, without the need to learn all the underlying technologies [2]. As OutSystems aims at rapid application development, automating the test case generation activity, based on their applicational model, along with coverage evaluation, will be of great value to developers using OutSystems.

Software testing is a quality control activity performed during the entire software development life-cycle and also during software maintenance [3]. Two testing approaches that can be taken are manual or automated. While for manual testing, the test cases are generated and executed manually by a human sitting in front of a computer carefully going through application screens, trying various usage and input combinations; in automated testing both the tasks of generation and execution of the test cases can be executed resorting to tools. The tool hereby presented covers the aspect of the test case generation and not its execution.

B. Tool

This showpiece introduces a tool that aims at generating, in an automated manner, test cases for applications developed in the visual language OutSystems.

The algorithm behind this tool takes on the visual source code of an OutSystems application and generates all the necessary input combinations so that the set of generated test cases would be able to reach the entirety of its nodes and branches, or detect and identify unreachable execution paths, which in practice correspond to dead code.

As the OutSystems language is mainly visual and represented graphically through a graph, this tool resorts to graph search algorithms, breadth and depth-first search, in order to traverse these graphs and retain all necessary information.

Due to the extensibility of the OutSystems model, this tool currently supports an interesting set of nodes related to the logic behind client/server applications. Fig. 1 shows said nodes integrated within a simple graph example.

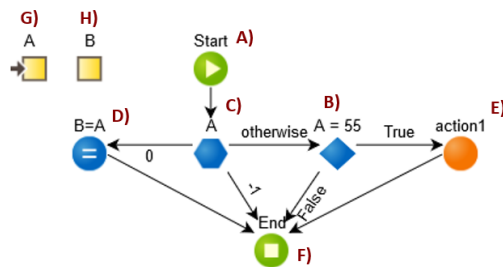


Fig. 1. OutSystems language nodes considered in this work: *Start* node (A) marks the start of a procedure; *If* node (B) expresses an if-then-else block behaviour; *Switch* node (C) representing a switch block behaviour; *Assign* node (D) indicating the attribution of values to variables; *Execute Action* node (E) represents a call to another procedure; *End* node (F) marks the end of a procedure. G) and H) represent input and local variables, respectively.

Along with the various input combinations that should be tested in order to achieve maximum code coverage and the identification of unreachable execution paths, it is also provided information on some warnings such as when variables are defined but never used in the trace of code analysed.

Software test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. This tool evaluates both the overall test suite as well as subsets of it in terms of node, branch, condition, modified condition-decision and multiple condition coverage [4-6].

For this, the algorithm traverses the graph, from the Start node consecutively following the nodes outgoing branches, applying a cause-effect graphing methodology [7] in order to reduce the generation of redundant combinations, and employing a boundary-value analysis [7,8] whenever a new decision point is reached in order to identify the values that should be tested for each individual condition.

The final test suite generated is prioritized according to two criteria: they are first organized in terms of the combined coverage they provide for both branches and nodes; the second criteria takes into account the number of decisions the path

corresponding to this test case encounters. This prioritization is also complementary, meaning that when the first "best" test case is found, the second test case to be displayed is the one that, together with the first one, helps to cover more nodes and branches. The same goes for the third pick and so on. This means that the first x test cases presented are the ones that will cover the most nodes and branches and no other combination of x test cases will be able to cover more code.

Fig. 2 shows the prototype for this tool, where the set of test cases generated are displayed in (A), with some warnings identified in (B) and the coverage results in (C).

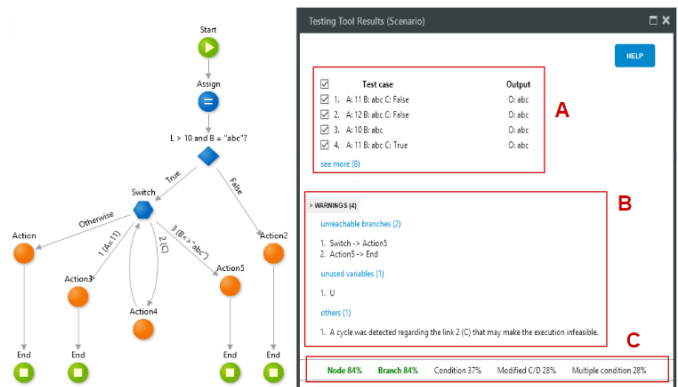


Fig. 2. Tool prototype window (expanded image at: <https://goo.gl/3yRWkA>)

A program is tested in order to add some value to it. This value comes in the form of quality and reliability, meaning that errors can be found and afterwards removed. This way, a program is tested, not only to show its behaviour but also to pinpoint and find as many errors as possible. Thus, there should be an initial assumption that the program may contain errors and then test it [8].

Nowadays, there is a high need for quick-paced delivery of features and software to customers, so automating tests is of the utmost importance. One of its several advantages is that it releases the software testers of the tedious task of repeating the same assignment over and over again, freeing up testers to other activities and allowing for a variety in the work as well as opening space for creativity. These factors are claimed to improve testers motivation at work [9].

As OutSystems aims at rapid application development, the automation of the test case generation activity, based on their applicational model, along with coverage evaluation, will be of great value to developers using OutSystems.

This tool is also of relevance to the VL/HCC community as it presents a solution for an issue that is very common within the visual languages paradigm. As the development of applications is still dominated by Textual Programming Languages (TPLs), a number of tools already allow automated testing over TPLs, but the same variety does not apply to VPLs. Tools such as the one here presented help increase the value brought by VPLs to developers.

II. PRESENTATION

This showpiece will be presented through video, showcasing its features (available at: <https://youtu.be/8GsY8NTNXdk>) as well as a demonstration that will involve the participation of users, consisting on an interactive exercise where the user will be able to experience the advantages brought on by this tool. This demonstration will consist of a simple two-part exercise, taking no longer than ten minutes, where one part will include the tool and the other will not. The results and feedback from this demonstration will be recorded for the purpose of evaluation of the tool.

Complementing this demonstration, there will also be a poster showcasing this tool's features alongside a set of experiments and corresponding results.

III. FUTURE WORK

This tool represents the introduction of automation of the test case generation activity and respective coverage evaluation within OutSystems applications. Therefore, some limitations are still in place. The future for this tool starts by expanding in terms of the types of nodes it supports for this language, as well as the datatypes it is able to evaluate, as currently the datatypes supported are the basic Integer, Boolean and Strings.

ACKNOWLEDGMENT

The authors would like to thank OutSystems for the support presented throughout the development of this tool.

REFERENCES

- [1] OutSystems <https://www.outsystems.com/>. Last accessed 12 July 2018
- [2] OutSystems: OutSystems - Agile Methodology DataSheet. OutSystems (2010) <https://www.outsystems.com/home/downloadsdetail/25/75/>. Last accessed 11 May 2018
- [3] K. Saravanan and E. Poorna Chandra Prasad: Open Source Software Test Automation Tools: A Competitive Necessity. *Scholedge International Journal of Management Development* **3**(6), 103–110 (2016)
- [4] C. Wenjing and X. Shenghong: A Software Function Testing Method Based on Data Flow Graph. In: 2008 International Symposium on Information Science and Engineering, pp. 28–31. IEEE, Shanghai, China (2008) 10.1109/ISISE.2008.23
- [5] Kshirasagar Naik, Priyadarshi Tripathy: *Software testing and quality assurance: theory and practice*. 1st edn. Wiley (2008)
- [6] Ammann, Paul and Offutt, Jeff: *Introduction to Software Testing*, pp.27–51. 1st edn. Cambridge University Press, New York, NY, USA (1999)
- [7] Ehmer, Mohd and Khan, Farmeena: A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications* **3**, 12–15 (2012)
- [8] Myers, Glenford J. and Sandler, Corey: *The Art of Software Testing*. John Wiley & Sons (2004)
- [9] Santos, Ronnie and C. de Magalhaes, Cleyton and Correia-Neto, Jorge and Silva, Fabio and Capretz, Luiz: Would You Like to Motivate Software Testers? Ask Them How. In: *Electrical and Computer Engineering Publications*, pp. 95–104 (2008) 10.1109/ESEM.2017.16

OS LANGUAGE OVERVIEW

The OutSystems platform allows developers to create end-to-end applications defining logic, UI, databases, processes and more. The documentation for the full OutSystems language can be found here [75].

Over the course of this appendix, the multiple tabs of Service Studio that allow for the developer to create the multiple layers of their applications are presented.

Processes

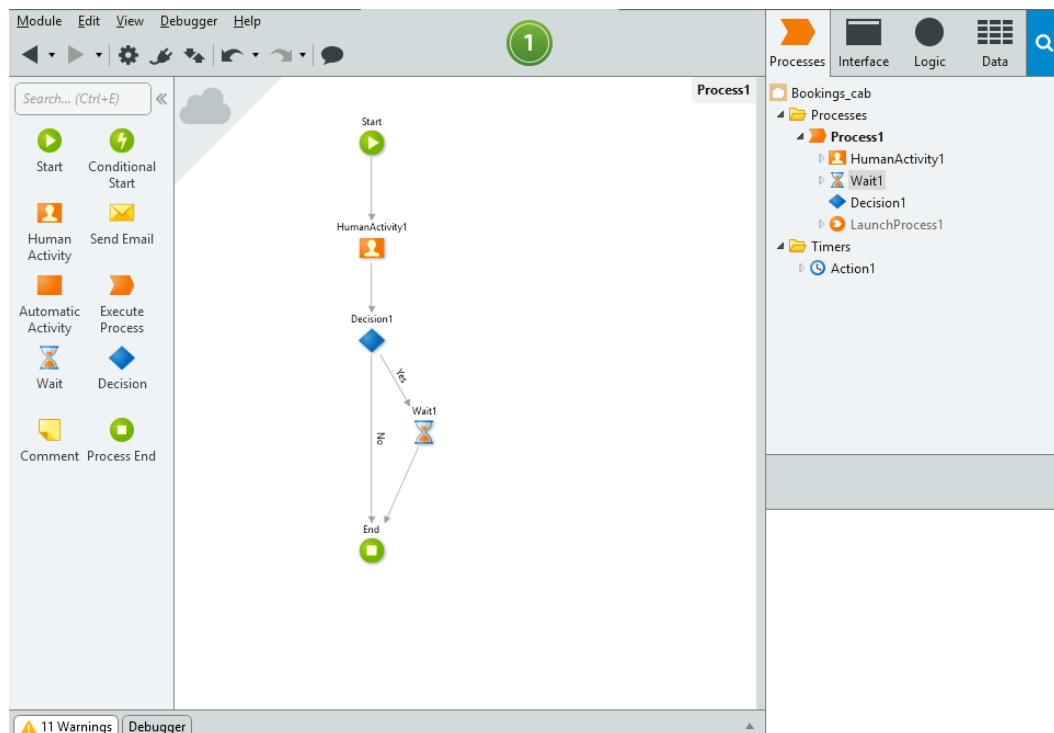


Figure I.1: Processes tab in Service Studio.

Figure I.1 shows the tab for Processes in Service Studio where the user can define and integrate business processes within their applications, based on activities which are executed over the course of an entity life cycle [77].

Interface

Figure I.2 shows the Interface tab in Service Studio, used to define the user interfaces of applications including screens and logic with them related. It also presents a multitude of pre-built widgets that can be easily integrated into the screens. The displayed widgets will vary depending on whether the user is developing a web or mobile application [74, 80].

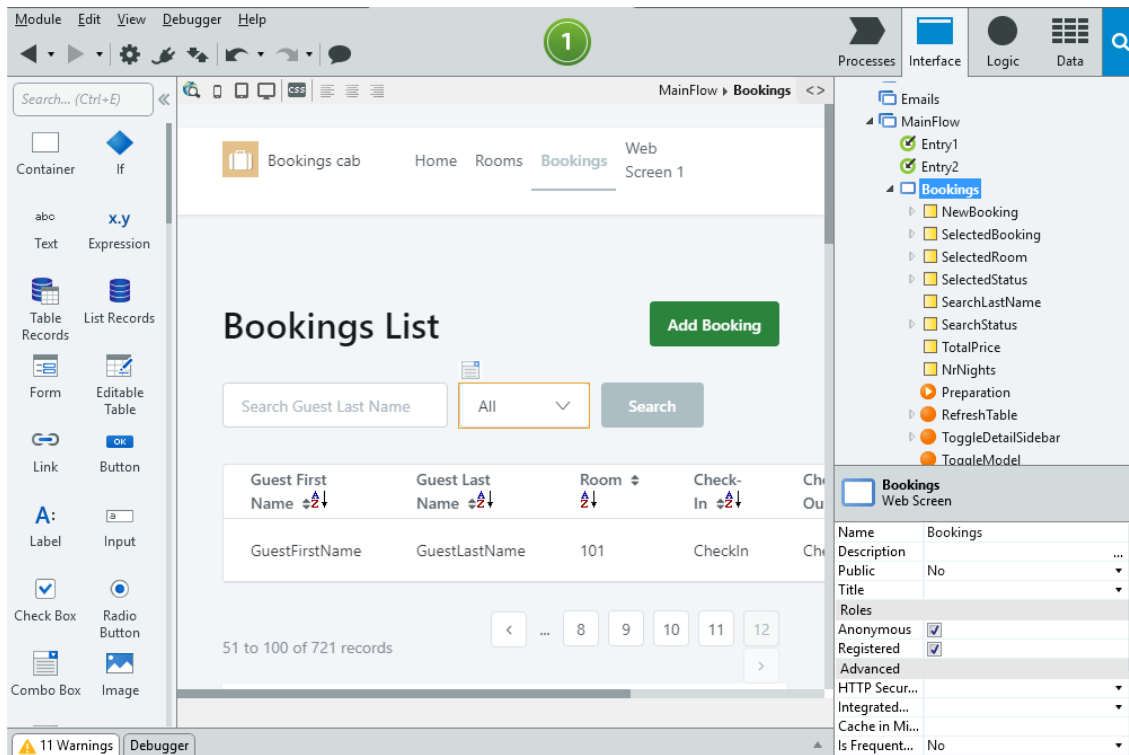


Figure I.2: Interface tab in Service Studio.

Logic

Figure I.3 shows the Logic tab in Service Studio, where the user can develop the server-side logic for their applications [73]. This is the focus of our work and so their elements will be individually detailed next. The nodes supported by the algorithm hereby presented are also featured and highlighted.



Used to indicate the beginning of a procedure.

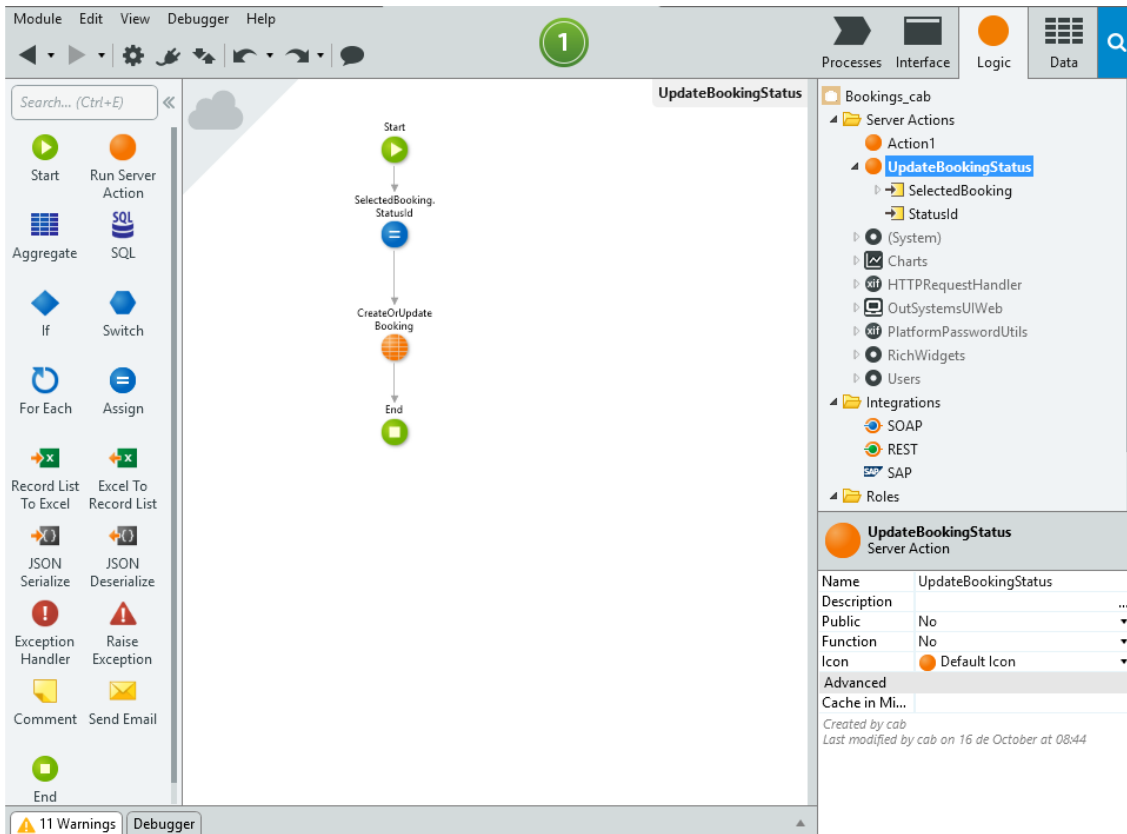


Figure I.3: Logic tab in Service Studio.

Server Action

Invokes a call to other server action. Can receive input and return output parameters according to its definition.

Aggregate

Used to fetch data from the database resorting to an optimized query.

SQL

Similarly to aggregates, it is used to fetch data from the database and here the user specifies the required SQL query.

If¹

¹Nodes covered by the algorithm presented by this work.

Contains a decision and decides on which way the flow shall progress according to the decision turning true or false, having both outgoing branches explicitly defined.

Switch¹

Contains a set of decisions and the flow shall continue over the decision turned as true. If none does, the information flow shall continue over the otherwise branch, always explicitly defined.

For Each

Iterates a specified path for each entry in a list.

Assign¹

Allows for multiple assignments where the value of a variable is set to another variable or value.

Record List to Excel

Used to convert a Record List into an Excel file.

Excel to Record List

Similar to the previous but this one converts the contents of an Excel file into a Record List.

JavaScript Object Notation (JSON) Serialize

Converts either a Record or a List Record to a **JSON** structure.

JSON Deserialize

Similarly to the previous it extracts the contents of a **JSON** object to either an Entity, Record or List.



Exception Handler

Captures and defines the behaviour that the program shall take after an exception is thrown.



Raise Exception

Throws an exception, ending the flow of the action.



Comment

Used to add comments to the flow. They can be used as TODOs by activating a flag and these will show up in TrueChange[78].



Send Email

Sends an email with a structure predefined by the user.



End¹

Indicates the end of a procedure flow.

Data

Figure I.4 shows the Data tab in Service Studio, where the required data model for the application is defined. In a similar manner to the Interface tab, Data will also look slightly different for mobile and web applications [69].

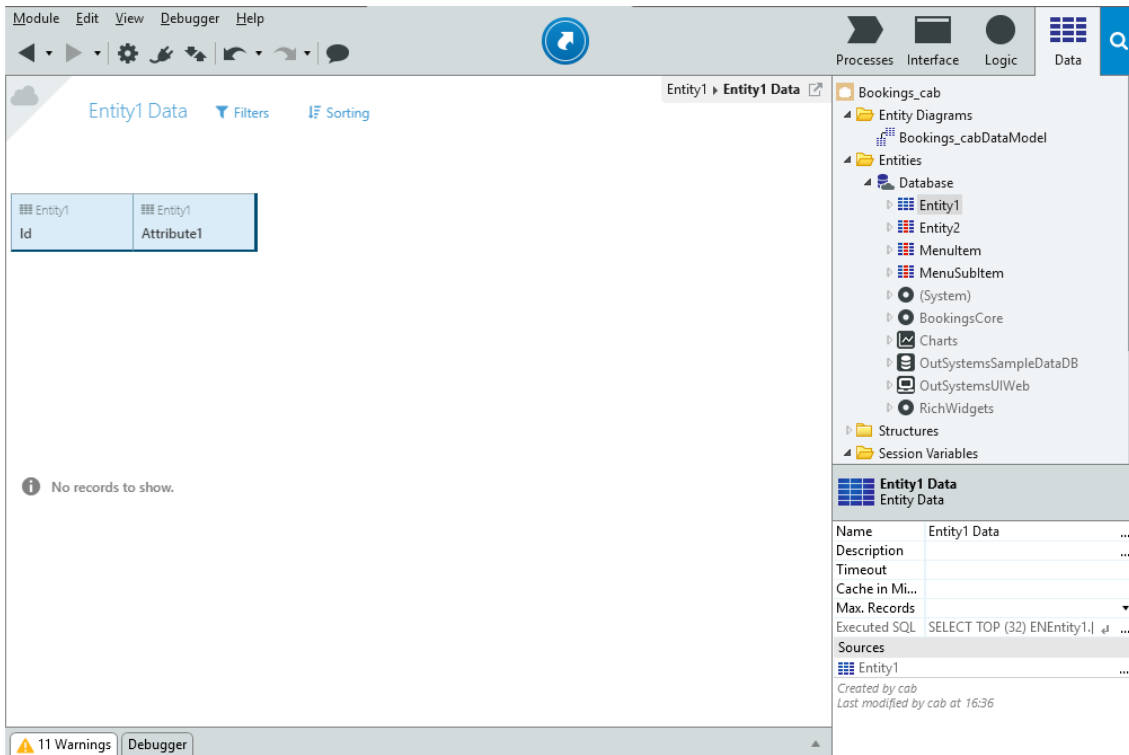


Figure I.4: Data tab in Service Studio.



TEST AUTOMATION TOOLS

II.1 Tools

Software testing tools are pieces of software which support the speedup of the testing activity.

In summary, the tools available in the market regarding software automation can be divided into two broad categories: Proprietary and Open Source Software Tools.

II.1.1 Proprietary software tools

These are the tools that require the purchase of a licence from the companies which developed the tool to be used. On the [Table II.1](#), we can see a list of proprietary software test automation tools.

Table II.1: List of proprietary software test automation tools [92].

Vendor	Product Name	Key Supremacy	Key Weakness
HP	Unified Functional Testing	Market Leader with string presence in UI Automation	High license cost
IBM	Rational Functional Tester	Ease of Use and Maintenance	Slow in innovation
Tricentis	TOSCA	Model Based Test Automation Framework	High license cost
Worksoft	Certify	Preferred for Packaged Applications Testing	Not reliable in Product Stability and Support

Oracle	Oracle Application Testing Suite	Good Customer and Support	Service	Challenge to maintain Scripts & does not support mobile applications
SmartBear	Test Complete	Offers support for multiple skill levels		No Support for Packaged Applications
Ranorex	GUI Test Automation	Straight forward model	licence	Complex UI object recognition
Progress	Telerik Test Studio	Supports Microsoft Technology applications		No Support for Packaged Applications
Automation Anywhere	Testing anywhere	Any- Strong customer support and ease of use		Complex Licensing Administration
Borland	Micro Focus SilkTest	Role-based testing		Needs improvement to deliver unified test automation solution
TestPlant	eggPlant	Use of intelligent image recognition algorithms		Difficult to maintain scripts
Original Software	TestDrive	Code free approach		Missing Mobile Testing

II.1.2 Open source software tools

Open Source testing tools have become competitive necessity to survive in the market and IT companies are gaining maximum advantage by adopting them, so we will look into them with further detail.

These are tools free of cost and the source code is available to modification/customization. The main tools in this category are presented on [Table II.2](#).

Table II.2: List of some of the popular open sources software test automation tools [92].

Vendor	Product Name	Key Supremacy	Key Weakness
Apache	Selenium	Web Driver supporting multiple browsers & multiple programming languages	Supports only web based applications
Apache	Geb	Using powerful Selenium WebDriver APIs with simple Groovy language	Specific to “Java based applications”
Apache	Windmill	Recorder tool that allows writing tests with learning a programming language	Supports only Web environment

Tyto Software	Sahi	Simple JavaScript based scripting tool with complete API which can run parallel scripts	User Interface is confusing
BSD	Watir (Web Application Testing in Ruby)	Supports Multiple OS & Supports playback of multiple scripts	Supports only ruby language
GitHub	Protractor	End to End framework for AngularJS Applications	Based on JavaScript
Cucumber Ltd.	Cucumber	Based on Behavior Driven Development, easy to write scripts	Supports only Web environment
GitHub	SpecFlow	Employs consistent Domain-specific language and is user friendly	Specific to .NET based applications
Grant Street Group	Tellurium	Works adopting "UI Module" approach	Supports only Web environment