UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Selection of heterogeneous test environments for the execution of automated tests

Miguel Alexandre Nunes Afonso

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa
Co-orientador: Prof. Doutor João Paulo Fernandes

Covilhã, Outubro de 2016

Acknowledgements

First of all I would like to thank Miguel Costa Antunes and António Melo for coordinating this dissertation in the industrial context at OutSystems. One special thanks also to Jens Smeds, for all the valuable feedback that he has providing me regarding this dissertation content. Your help has been huge and all the results obtained in this dissertation would not be possible without your help. I will never forget how good it was to work with you and all the help you have provided me!

Then, I would like to thank Professor Simão Melo de Sousa and Professor João Paulo Fernandes for coordinating this dissertation. Thank you for all the feedback provided and for enabling the possibility of doing this work in the industrial context at OutSystems.

I would also like to thank all the people that helped me at OutSystems during this time.

Continuing in portuguese ....

Os primeiros agradecimentos são para os meus pais e a minha irmã, uma familia que me transformou na pessoa que sou hoje.

Com a ajuda e incentivo dos meus pais, consegui prosseguir estudos e fazer esta dissertação, graças a todas as condições que eles me deram e pelas quais fico eternamente agradecido.

Agradeço à minha irmã pelo constante incentivo e preocupação, acompanhando-me durante este percuso e tentando sempre animar-me.

De seguida, um agradecimento gigantesco à minha namorada Rita que sem ela não teria sido possivel fazer este trabalho. Ajudaste-me nos momentos mais complicados e a tua ajuda e apoio fez realmente a diferença, permitindo-me terminar esta etapa.

Um grande obrigado aos meus padrinhos com o apoio que me deram ao longo dos anos em que estive a estudar, incentivando-me sempre a sonhar mais alto.

Para terminar, um grande obrigado a todos os meus amigos que me ajudavam, distraindo-me e animando-me quando mais precisava. Espero de não me esquecer de ninguém portanto obrigado Emanuel Grancho, Bruno Silva, Sérgio Costa, Fábio Rodrigues, Pedro Tavares e Nuno Pina.

Obrigado mais uma vez a todos e espero que vos possa pagar um fino para festejar a entrega desta dissertação!

Resumo

À medida que a complexidade do software aumenta o mesmo acontece com a dimensão das suites de testes automizados que permitem validar o comportamento esperado do sistema que está a ser testado. Quando isso ocorre, aparecem problemas para os programadores sob a forma de aumento de esforço necessário para gerir o processo de teste e maior tempo de execução das suites de teste.

Gerir manualmente milhares de testes automatizados é especialmente problemático uma vez que os custos recorrentes de garantir que os testes automatizados (ex: milhares) estão corretamente configurados para executar nos ambientes de testes disponíveis (ex: dezenas ou centennas), durante o tempo de vida dos produtos pode tornar-se gigantesco. Este problema aumenta substancialmente quando o sistema que está a ser testado é um produto altamente configurável, precisando de ser validado em ambientes heterogéneos, especialmente quando também estes ambientes destino de testes também evoluem frequentemente (ex: novos sistemas operativos, novos browsers, novos devices móveis, ...). O tempo de execução destas suites de testes torna-se também um problema enorme, dado que não é viável executar todos as suites de testes em todas as configurações possiveis.

Sendo uma parte integral do desenvolvimento de software, a forma de testar precisa de evoluir e libertar-se dos métodos convencionais. Esta dissertação apresenta uma técnica que estende um algoritmo existente que permite reduzir o número de execuções de testes, e estende-o, permitindo fazer a distribuição de casos de teste sobre múltiplos ambientes de teste heterogéneos.

O desenvolvimento, implementação e validação da técnica proposta na presente dissertação foram conduzidos no contexto industrial de uma empresa internacional de desenvolvimento de software. Foram utilizados cenários de desenvolvimento de software reais para conduzir experiências e validações, e os resultados demonstraram que a técnica proposta é eficaz em termos de eliminar o esforo humano envolvido na distribuição de testes.

Palavras-chave

Engenharia de Software, Testes de Software, Testes Automatizados, Controlo de Qualidade, Ambientes de Teste Heterogéneos, Minimização de Testes, Distribuição de Testes

Resumo alargado

Um objetivo comum das empresas que desenvolvem programas é garantir a entrega de produtos com alta qualidade ao utilizador final. Para garantir a alta qualidade dos produtos, é necessário testar os produtos, validando que estes cumprem as especificações e requisitos definidos.

As empresas que desenvolvem soluções, frequentemente criam soluções altamente configuráveis, tentando expandir as funcionalidades e serviços disponíveis no produto, enquanto mantêm e reutilizam um código fonte comum. Por exemplo, um mesmo produto pode correr em sistemas operativos diferentes ou ser acessível por browsers ou dispositivos móveis díspares. Esta customização é obtida ao permitir escolher variações pré-definidas na configuração do produto, transformado o sistema debaixo de teste num sistema altamente configurável. No caso de sistemas altamente configuráveis, a dificuldade da sua validação aumenta significamente, sendo a principal forma de o conseguir, a execução de testes em múltiplos ambientes heterogéneos.

À medida que a dimensão do produto em desenvolvimento aumenta, o mesmo acontece com o número de testes automatizados que permitem validar o comportamento esperado do sistema que está a ser testado. Distribuir os testes por múltiplos ambientes com configurações heterogéneas exige um grande esforço humano, tendo de ser selecionado para cada caso de teste os ambientes de teste em que devem ser executados.

O principal objetivo desta dissertação é propor uma técnica que permite eliminar o esforço humano na seleção dos ambientes de teste onde cada caso de teste deve ser executado. Com este objetivo em mente, pretende-se ao longo da dissertação apresentar uma técnica que tenha a capacidade de selecionar os ambientes de teste onde cada caso de teste deve ser executado. Uma solução deste género permite eliminar totalmente o esforço humano envolvido na gestão da distribuição dos casos de teste pelos múltiplos ambientes de teste heterogéneos.

Em resultado do levantamento do estado da arte que foi realizado, foi possível constatar que já foram desenvolvidos alguns esforços, tanto em contexto académico como industrial, com vista à otimização da distribuição de casos de teste. Estes trabalhos foram estudados no contexto da presente dissertação, com o objetivo de identificar técnicas e ferramentas que permitissem o objetivo antes referido. Analisando estes trabalhos, a maioria apresenta soluções usadas para distribuir casos de teste por múltiplos ambientes de teste que permitem especificar os requisitos que o ambiente deve cumprir para executar o caso de teste. Por outro lado, os requisitos que são possíveis de especificar são muito pouco flexíveis, não permitindo descrever um racional das variações em que os testes devem ser executados. As técnicas apresentadas selecionam um ambiente de teste para executar cada teste, enquanto que o objetivo desta dissertação é selecionar o mínimo número de ambientes de teste necessários para satisfazer todas as variações em que o teste precisa de ser executado. Para permitir a descrição deste racional, foram estudadas técnicas de minimização que permitem descrever os fatores que influenciam o comportamento esperado de um teste e geram o número mínimo de combinações desses fatores com que o teste deve ser executado.

Esta dissertação apresenta uma técnica que estende um algoritmo de minimização existente que permite reduzir o número de execuções de testes, e estende-o, permitindo fazer a distribuição de casos de teste sobre múltiplos ambientes de teste heterogéneos. As variações nas quais o

teste deve ser exercitado são descritas como fatores na especificação do teste e o conjunto mínimo de ambientes de teste onde o teste tem de ser executado é selecionado, garantindo que todas as variações necessárias são exercitados. No caso de não existirem ambientes de testes que satisfaçam todas as variações exigidas pela especificação do teste, a técnica retorna uma falha no caso de teste gerado que representa essa variação.

A técnica proposta foi implementada em prática num ambiente real, e nomeadamente no contexto de uma empresa internacional de desenvolvimento de software que desenvolve um produto altamente configurável. Esta mesma implementação foi utilizada para realizar a seleção dos ambientes de teste onde cada caso de teste tem de ser executado, tendo sido exercitado um conjunto de casos reais existentes na empresa.

Relativamente aos resultados obtidos, nos diversos cenários de validação a técnica proposta na presente dissertação apresentou o comportamento esperado, sendo capaz de selecionar os ambientes de teste esperados durante os vários exercícios. A técnica demonstrou ser eficaz em termos de eliminar completamente o esforço humano envolvido na distribuição de casos de teste pelos múltiplos ambientes de teste heterogéneos.

Abstract

As software complexity grows so does the size of automated test suites that enable us to validate the expected behavior of the system under test. When that occurs, problems emerge for developers in the form of increased effort to manage the test process and longer execution time of test suites.

Manual managing automated tests is especially problematic, as the recurring costa of guaranteeing that the automated tests (e.g.: thousands) are correctly configured to execute on the available test environments (e.g.: dozens or hundreds), on a regular basis and during the products lifetime may become huge, with unbearable human effort involved. This problem increases substantially when the system under test is one highly configurable product, requiring to be validated in heterogeneous environments, especially when these target test environments also evolve frequently (e.g.: new operating systems, new browsers, new mobile devices, ...).

Being an integral part of software development, testing needs to evolve and break free from the conventional methods. This dissertation presents a technique that extends one existent algorithm to reduce the number of test executions, and extend it, enabling to perform the test case distribution over multiples heterogeneous test environments.

The development, implementation and validation of the technique presented in this dissertation were conducted in the industrial context of an international software house. Real development scenarios were used to conduct experiments and validations, and the results demonstrated that the proposed technique is effective in terms of eliminating the human effort involved in test distribution.

Keywords

Software Engineering, Software Testing, Automated Testing, Quality Assurance, Heterogeneous Test Environments, Test Minimization, Test Distribution

Contents

List of Figures

List of Tables

Acronyms

PaaS Platform as a Service

IDE Integrated Development Environment

DE Development Environment

PS Platform Server

QA Quality Assurance

CM Combinatorial Model

RF Relevant Factors

TS Test Specification

TE Test Environment

TCL Test Case List

SM System Model

AC Available Combinations

RC Required Combinations

TC Test Case

HTML HyperText Markup Language

SUT System Under Test

CIT Combinatorial Interaction Testing

ACVC Ada Compiler Validation Capability

NMI NSF Middleware Initiative

XML EXtensible Markup Language

CSS Cascading Style Sheets

Chapter 1

Introduction

The overall aim of the software industry is to ensure the delivery of high quality software to the end user. To ensure high quality software, it is required to test it, validating that the software meets user's specifications and requirements.

Software companies build customizable software systems in order to expand their market through new functionality and services, while still reusing and maintaining a common code base.

This customization is achieved by allowing to choose pre-defined variations in the software configurations, transforming the software under test in one configurable system.

In this dissertation context, a configurable system is one generic code base and a set of mechanisms for implementing pre-planed variations in the system's structure and behavior. For example, if the system under test can be connected with different database engines, then the expected behavior when connected with all of the supported databases engines needs to be validated.

Using the previous example, the database is one *factor* that influences the behavior of the system under test and each *factor* needs to be exercised against all the possible pre-defined *values*, for instance, we need to validate the expected behavior when connected with Sql Server, Oracle or MySQL.

In this dissertation context, each configuration that may influence the behavior of the system under test will be referred as one *factor* and it is composed by the possible pre-defined *values* that it is possible to bind.

Many modern software systems are designed to be highly-configurable, presenting significant challenges for validation. The problem of testing a single software system has been replaced with the much harder problem of testing the set of software systems that can be generated by all the different possible bindings of configurations.

Besides validating the configuration options that the system under development supports, it is also required to validate the system behavior when installed in other existing systems.

One system must be tested with a wide variety of combinations of configurations, different operating systems, different web browsers, different mobile devices and other variations. Therefore, validating the behavior of the system under test includes to execute tests in multiple heterogeneous environments. An heterogeneous environment is composed by one or more available value for each factor.

For example, two heterogeneous test environments may be: a) one test environment with database Oracle and browser Chrome installed and b) one test environment with database Sql Server and browser Firefox installed.

Having to assure quality in multiple heterogeneous environments creates several problems related with test management. There are several different use cases to manage:

- tests that only can be executed in one concrete configuration, for example, one test that validates the support of T-SQL (Microsoft's proprietary extension to the SQL) query is only possible to execute in one system with Sql Server;
- tests that must run against all variations of one configuration, for example, one test that validates the usage of SQL querys in all the database engines;
- tests that are possible to execute in multiple configuration but run them in more than one do not add any value, for example, testing one database query that can execute in all possible databases, but it is enough to execute in one of the possibilities being irrelevant what is the selected database;
- tests that must exercise different combinations of configurations because they are influenced by more than one factor, for example, test that validates one component rendering in different browsers and operative systems .

Distributing tests over multiple heterogeneous test environments requires a decision on which tests should be executed in each environment. In this process, the person that is making that decision needs to know which are the *factors* that influence each test and evaluate which are the test environments where the tests should execute.

The previous task, selecting the test environments where each test should execute, assumes that the person that is making the selection knows the factor that influence each test. With the test suites growing in a large pace, the human effort required to keep the test suites updated and being executed in the proper test environments is cumbersome and this process is highly prone to human errors.

This dissertation elaborates on the subject of reducing the human effort required to manage the test distribution by the multiple heterogeneous environments.

In order to reduce the human effort required, a test management technique is proposed. The proposed technique is able to retrieve the required information from the test specification so that each test case can be assigned to be executed in the test environments that fulfill the requirements of the test case.

This technique allows the person that is creating the test to describe which factors are relevant to be exercised in the test execution. The test environments where the test should be executed will be automatically selected based on the previous factors and the available test environments to execute the test.

The proposed technique was implemented in a real software development environment and validated, using real test scenarios available at OutSystems. Furthermore, the proposed solution was implemented in the current test infrastructure and with the technologies that the company is using to create and execute its automated tests.

This environment provided a challenging context for the evaluation of the described technique. The evaluations that were conducted considered a significant number of scenarios and in all

those scenarios the proposed solution demonstrated to be effective.

The proposed technique demonstrated to be able to select the correct test environments were the tests should execute, selecting the minimum number of test environments required to exercise the factors described for each test.

In the remainder of this chapter, the context and motivation of this dissertation are described, the problem to be solved and dissertation goals are defined, the research methodology is presented, followed by an overview of the proposed solution, main contributions of this work and finally an overview of the structure of the current document.

1.1 Context and Motivation

The work presented in this document was conducted in the real software development context of OutSystems, an international software house founded in Portugal in 2001 and with its Research & Development group based in Portugal. All the work that is related with this dissertation was conducted at OutSystems Research & Development in the R&D Productivity Group, and was supported with a scholarship sponsored by OutSystems.

OutSystems is mainly known by developing the OutSystems Platform, that is in fact a piece of software that is highly configurable, with a large automated test suite and that has evolved through many years.

This is, therefore, a scenario that fits the conditions mentioned above in most of software systems currently being developed. The OutSystems platform is highly customizable, allowing to choose the database engine where the data is stored, the application server where it is deployed and multiple other variations.

This software provides customers with an IDE which allows to program mobile and web applications using visual models. The applications that are generated work in any desktop browser, mobile browser or if it is one native mobile application, work in different mobile operating systems (iOS and Android).

OutSystems test suite has more than 10000 automated tests that are currently being manually distributed to execute in multiple test environments. Therefore, OutSystems is a perfect context to solve the problem of test distribution by heterogeneous test environment and, better than that, a perfect context to validate the proposed solution in a real industrial software development context.

In OutSystems, the development methodology includes the process of quality assurance in multiple heterogeneous environment, mainly because of the highly configurable system that is the OutSystems Platform. Although the specific context of OutSystems served as a use case for the presented technique, the problems that are being solved are recurring concerns in industry.

The industry deals with scenarios where both software code and automated test cases are being developed over the years by many different developers, many of which might not be available anymore. Besides, test suites tend to grow with time, becoming large test suites with thousands

of tests. In such a context, it is simply not affordable to manually select the proper test environment where the test must execute: even if the rationale for the distribution of a particular test is still clear (which often is not), the number of tests in the suite invalidates any accurate manual selection.

Manual selection of the proper tests environments where each test should execute is highly prone to human errors. Currently, whenever one new test environment is created, the tests selected to execute on that test environment will be chosen based on the tests that are executing in one older similar environment. If there are errors making the selection of the test environment where the test should execute, over the time these errors get more and more severe, because they will be propagated to the new test environments. With the accumulation of these errors it is nearly impossible to understand the reason why a given test is selected to execute in a chosen heterogeneous environment.

The main motivation to this work is, therefore, to propose one automated test case distribution technique that is able to select the heterogeneous test environments where which test should execute, so that the tests are only selected to execute in the required test environments to assure quality. This work brings innovation in using one technique already explored in the literature (Combinatorial Interaction Testing, used as one test case minimization technique) and extend it to also achieve one new goal, transforming it to one test case minimization and distribution technique. Another goal is to demonstrate that the technique proposed on this dissertation is able to work in real scenarios, presenting one proof of concept showing the usage of the technique and the results obtained.

1.2 Problem Statement and Goals

The context and motivation described above clearly state that the software industry in general is suffering from the fact that assuring quality of highly configurable systems in multiple heterogeneous environments is not easy to achieve. The factors that affect the system behavior are in constant evolution, and they may appear from different sources:

- appearing in the system requirements one new *value* that must be supported in one factor;
- end of support from one specific value;
- adding or removing one factor that influences the system under test;

Depending on the code that the test is exercising, some *factors* may not be relevant to the test execution. Testing all the possible bindings of values for each factor is not always the best option, sometimes execute the test in all the available test environments do not add any value to the results. Having the possibility to describe only the relevant factors for the test is extremely important, creating different test requirements in the same test suite.

With the code, test suites and requirements of each test case growing with time, software maintainability tends to become harder and implying more costs. Therefore, it is important to develop techniques that automate the process of distributing the tests over heterogeneous test

environments. The ability to select the test environments where the test should execute based on the pre-defined variations that the test must exercise is crucial to solve this maintainability problem, and therefore this is the problem that this dissertation aims to solve.

So, more formally, the problem that this dissertation aims to solve and that is specifically concerned about test case distribution, might be defined as follows:

Problem Statement (Test Case Distribution Problem)

Given: A test *t* identified with the set of its relevant factors and, for each factor, the values that are important for it, $\{f_1, f_2, ..., f_n\}$ that need to be validated and a set of heterogeneous test environments $\{e_1, e_2, ..., e_m\}$.

Problem: Select the subset $\{e_i \| i \leftarrow e_1, e_2, ..., e_m\}$ where the test *t* should execute, so that a test is only selected for more than one environment if strictly necessary to exercise all the relevant factors and values of the test *t*.

The main goal of this dissertation is to solve the problem of test case distribution on highly configurable environments. More concisely, the challenge tackled in this work should include solutions that meet the following requirements:

R1) It must deal with different relevant factors for each test.

R2) It must scale to large and continuously growing systems.

R3) It must deal with a constant changes in factors that influence the system, test suite and test environments.

R4) It must deal with the lack of test environments that can fulfill the relevant factors, returning a test failure if there are not enough environments.

R5) It must apply to real environments, within real validation scenarios.

R6) It must be deterministic, given the same input must always return the same selection.

Furthermore, a sub-product of the described main goal is to, apart from solving the generic problem that is presented here, implement a solution prototype and validate that prototype in the real context of a real software development company.

1.3 Research Methodology

To achieve the goals mentioned in the previous section, the work conducted during this dissertation was divided in different tasks. The initial tasks were to study the described problem and existing related work and only then propose a solution. The following tasks were to implement the designed solution and finally validate the implemented solution.

Concisely, the research methodology used in this dissertation comprised the following tasks:

1. Characterization of the problem domain, and specifically the industrial context that led

to the formulation of the general problem that is to be solved.

2. Formulation and characterization of the problem that is to be solved during the work.
3. Survey on the academic and industrial publications, so that a complete overview of the test distribution problem might be elaborated.
4. Proposal of a solution to the described problem in an generic way, without any requirements that only fit in a specific industrial context.
5. Implementation of the proposed solution in a practice and in the context of a real software development company.
6. Evaluation of the behavior of the proposed and implemented solution, specifically demonstrating the theoretical wasted human effort reductions, using real chosen scenarios from daily test case management.
7. Analysis of the implemented solution and its evaluation results, pointing out future research directions and possible future work still needed to be done

1.4 Proposed Solution

A brief overview on the proposed solution is given in this section, being this description detailed on Chapter 4.

In order to select in an automated way in which test environments a test should be executed, there must be a way to evaluate if one heterogeneous test environments should execute the test. The tests are selected to executed in one test environment based on the relevant factors that need to be exercised for each test. This information needs to be described in each test case, in such a way, that enables both the developers and the test execution engine to understand the rationale behind the chosen relevant factors to the test.

The solution that is proposed in this dissertation describes the relevant factors for a test, using one modeling approach based on Combinatorial Interaction Testing techniques. The relevant factors are described in one model that contains: the factors that need to be exercised; the values that each factor can take; the number of parameters interactions that will be validated; and sub models definition enabling to assign different importance between factors interaction.

The relevant factors description, allows one combinatorial engine to calculate the minimum number of combinations that the test is required to execute, in such a way, that fulfil the requirements described in the relevant factors. Each combination is the assigning of one value to each factor of the relevant factors description.

For example, considering one test with the relevant factors description containing three factors: *A*, *B* and *C*. The factor *A* with the possible values $\{1, 2\}$, the factor *B* with the possible values $\{X, Y\}$ and *C* with the possible values $\{J, K, L\}$.

With the parameter interaction equals to one ($T = 1$) the combinatorial algorithm will return one group of combinations where each value will appear at least once. Two possible valid results for this example are:

- $\{(1, X, J), (2, Y, K), (1, X, L)\}$
- $\{(1, X, J), (2, Y, K), (1, Y, L)\}$

With this simple example, there are multiple valid solutions that fulfil ($T = 1$).

If the parameter interaction is two ($T = 2$) instead, the combinations generated need to contain all pairs of values between different factors. For example, one valid result is represented in the table 1.1. The result has six combinations instead of three, because the coverage of parameter interactions has increased, being more probable to detect faults.

Table 1.1: Pairwise example

A	B	C
1	X	J
2	Y	J
1	Y	K
2	X	K
1	X	L
2	Y	L

With $T = 3$, only exists one possible result because $T = N$, being N the total number of factors. In this case, the result will be all the possible combinations of the three factors, represented in the table 1.2.

Table 1.2: All combinations example

A	B	C
1	X	J
2	X	J
1	Y	J
2	Y	J
1	X	K
2	X	K
1	Y	K
2	Y	K
1	X	L
2	X	L
1	Y	L
2	Y	L

Describing the relevant factors for one test using combinatorial models enables to describe the rationale that should influence the test distribution over the heterogeneous test environments. With this description, the combinatorial engine is able to calculate the minimum number of required combinations that the test should execute and generate one new test case to be selected to execute in one heterogeneous test environment.

To be able to execute in one heterogeneous test environment, the test case requires all the assigned values to be available in that test environment. The description of the available values in one test environment is done by defining the values that are available for each factor in that test environment.

In the proposed solution, the available values and the relevant factors are only validated in compile time and to achieve that there is one extra component required. This component is designated by system model and contains all the possible factors and values known that can affect the behavior of the code under test. The system model also contains constraints between values interactions, enabling to define globally the unwanted combinations between values.

The solution that is proposed in this dissertation is designed to work in a highly distributed architecture where each test environment is able to select what is the proper action for each generated test case: a) execute the test case, b) other test environment is responsible to execute this test case or c) lack of test environment with all the required values to be able to execute the test case.

The proposed solution to the problem of test management has the ability to scale to any number of factors and respective values that may influence the test being distributed, enabling to select the correct test environments where each test case should execute, as will be described and shown during the rest of this document.

1.5 Main Contributions

The work described on this dissertation led to both practical (relevant to the industry) and research oriented contributions. The main contribution of this dissertation is the proposal of a test case automated distribution technique, which consists in extending the current Combinatorial Interaction Testing techniques, in such a way, that allows to distribute the test cases by heterogeneous test environments based on the relevant factors. In order to achieve the main contribution of this work, previous contextualization and research were conducted and they led to the following contributions:

1. Presentation of a complete and concise characterization of a real industrial software development scenario, so that the defined problem and its solution emerge from requirements that exist in the software industry.

2. Wide survey on the subject of automated test distribution, presenting related works to these topic and the different approaches, challenging the advantages and weaknesses of those approaches.

3. Proposal of an automated test case distribution technique, using one commonly studied field in the academy to achieve one new goal.

4. Implementation of one prototype in one real industry scenario, using real tests and the available test infrastructure.

5. Validation of the proposed technique with real test scenarios.

1.6 Document Structure Overview

This dissertation is organized in seven main chapters. Of those seven chapters, five make part of the body of this dissertation. Those chapters are preceded and succeeded by the Introduction and the Conclusion chapters, respectively. Each of the seven chapters of this dissertation can be summarized as follows:

- **Chapter 1** introduces the topic of this dissertation, starting with a contextualization of the problem and the motivation for solving it. Then, a more concise description of the problem is presented, together with the expected goals for the present work. The approach for solving the problem is presented next, followed by the main contributions of the work depicted in this dissertation. In last, an overview of this document's structure is presented.
- **Chapter 2** concisely describes the industrial context where the work that led to this dissertation was conducted, specifically the OutSystems industrial context, mentioning all the relevant details that are crucial for a correct understanding of the problem and the necessity of solving it.
- **Chapter 3** presents a wide and detailed state of the art on the areas of software development, fast feedback and test suite optimizations in general. In this chapter an analysis is done on the different approaches found in the literature, explaining the advantages and weaknesses of each of those presented approaches, as well as understanding where innovation opportunities exist and how.
- **Chapter 4** contains the proposed solution for the problem of test management, which is a test case distribution methodology that extends the capabilities of Combinatorial Interactions Testing approaches currently existing. This chapter intends to present an algorithmic, abstract and generalized description of the presented solution, in order for it to be suitable for implementation in different industrial contexts.
- **Chapter 5** details a practical implementation of the solution presented in Chapter 4. Specifically, a set of technologies and languages that were used in that implementation are described. Then, each different step of the implementation is fully described, with all the implementation details that are relevant to understand how the implemented solution works in practice and how can that implementation be reproduced in this or other contexts.
- **Chapter 6** presents the validation methodologies that were used to demonstrate the functionality and correctness of the proposed solution, and that were conducted in the context of OutSystems daily development tasks with real selected data. Then, the results obtained in the proof of concept are presented, allowing to validate the human effort reduction.
- **Chapter 7** includes some final remarks and summarizes the work presented in the body of this document. Furthermore, it also presents the main conclusions of this dissertation, together with some directions for future work.

Chapter 2

Industrial Context

This chapter presents and characterizes the industrial context where this dissertation work was conducted. The main goal for this chapter is to make clear all the relevant details about how OutSystems develops its main product, and from there understand where resides the problem of test management, define a field of intervention and set the relevant research areas that will be further studied in Chapter 4.

2.1 Introduction

Being able to release highly configurable software with fast release cycles is one huge challenge. Ensuring the delivery of high quality software to the end user, requires to execute test in multiple heterogeneous test environments, validating that the software meets user's specifications and requirements.

One highly configurable system must be tested in a wide variety of heterogeneous environments, representing different operating systems, different web browsers, different mobile devices and other variations. Therefore, validating the behavior of the system under test includes the execution of tests in multiple heterogeneous environments.

Distributing tests over multiple heterogeneous test environments requires a decision requires a decision of defining which test should execute on which test environment. This process requires that one person selects the test environment (or tests environments), decides which are the *factors* that influence each test and evaluate which are the test environments where the tests should execute.

The previous task, selecting the test environments where each test should execute, assumes that the person that is making the selection knows the factors that influence each test. With the test suites growing in a large pace, the human effort required to keep the test suites up-to-date and being executed in the proper test environments is cumbersome and this process is highly prone to human errors.

While the need for practices, methodologies and tools that reduce the human effort involved in distributing the tests over multiple heterogeneous test environments can also be witnessed at OutSystems, this is by no means a need that is only applicable to this company. In the particular case of OutSystems, this need was so strong that led to the creation of a R&D Productivity team, which focuses precisely on continuously improving tasks that relate to software development, and therefore aim to reduce the human effort of the developers.

In this chapter, the industrial context of OutSystems is described. While this context is partic-

ular of that company, it is believed that the characteristics that are relevant for this dissertation are common to most software development companies that develop highly configurable software. This will help to clarify the wide applicability and potential of the proposed technique. OutSystems and its main product are described, as well as the development processes and methodologies that have been adopted in the software development process. Furthermore, some detail on its infrastructure is given, enabling to understand the high heterogeneity present in the available test environments.

2.2 The Company and its main Product

OutSystems is an international software house that develops the OutSystems Platform, one software that is provided both on premises as well Platform as a Service solution allowing customers to create mobile and web applications using visual models. Furthermore, it also includes a full application lifecycle management for the developed applications, and is prepared for applications that scale from small to large enterprise installations with any number of users.

OutSystems Platform allows customers to take advantage of the benefits of generating standard and optimized Java or C# code using one visual programming language. OutSystems Platform is categorized by the industry as one low-code development platform, meaning that is a platform that enables rapid application delivery with a minimum of hand-coding, and quick setup and deployment of the developed applications.

Applications developed with the OutSystems Platform allow to develop only once for all devices (cross platform), and with different stack combinations (different combinations of application servers, operating systems and databases). The OutSystems Platform capabilities are represented in Figure 2.1.

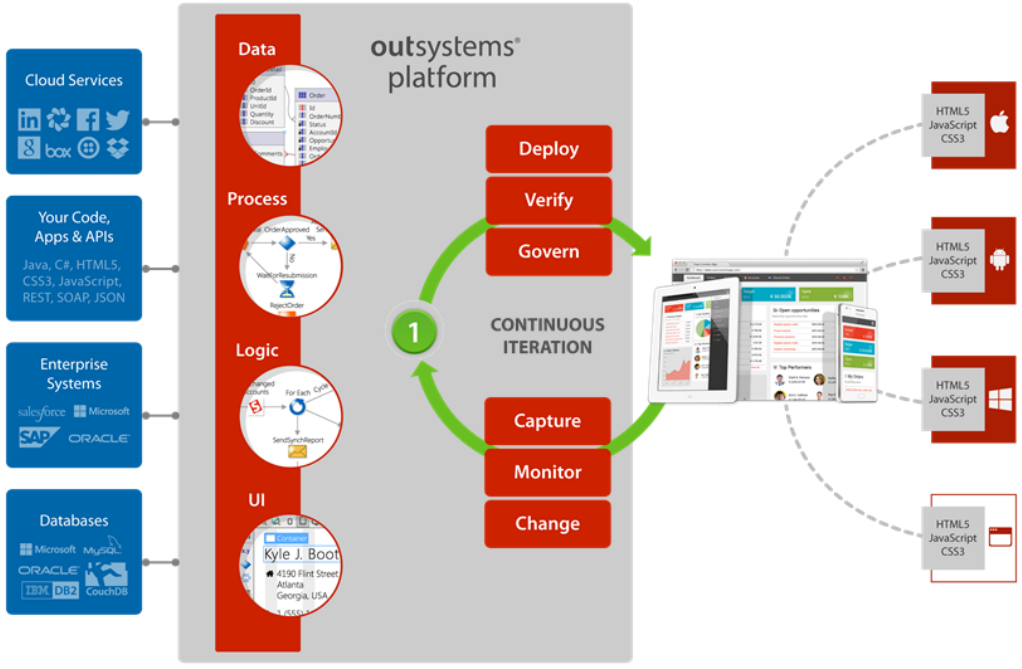

Figure 2.1: The OutSystems Platform

OutSystems Platform allows customers to reuse the code that they have already developed and integrate with OutSystems Platform. This capability enables to reuse the customers assets: the cloud services, the existing Java and C# code, the existing enterprise systems and the existing databases.

OutSystems Platform adds one layer of abstraction that enables to model and design the four different layers of the applications: Data, Process, Logic and User Interface(UI). With the development done, the developers have the ability to create the application with one click publish (generating the application in Java or C#, with the interface in HTML 5 and CSS 3).

Furthermore, the OutSystems Platform generates the application ready to work in multiple devices with different operating systems, deploying the application to the multiple devices. When the application is being used by the end users, the OutSystems Platform allows to capture feedback from real users, monitor the performance of the generated applications and change accordingly with the received feedback.

Figure 2.2 presents an example of the business logic of a program defined using a visual model. This definition is made within the IDE that is included in the OutSystems Platform, which is called Service Studio.

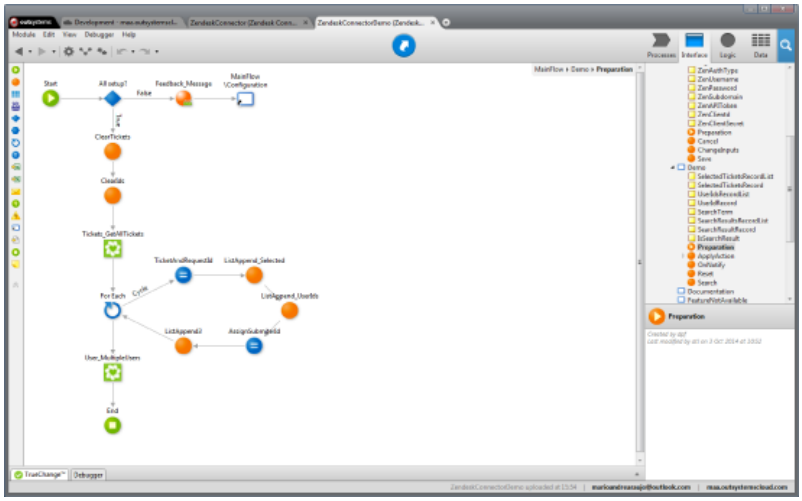

Figure 2.2: The OutSystems Platform - IDE Samples

Regarding the platform architecture, it can be divided in two big groups - the Development Environment (DE) and the Platform Server (PS). The DE is composed of two components: Service Studio and Integration Studio. The first one is the Integrated Development Environment already illustrated in Figure 2.3, and allows customers to use visual models in order to create mobile and web applications, that are then compiled and deployed by the PS. The second one is also a desktop application that allows users to integrate external libraries, services and databases in developed applications, extending the OutSystems Platform with additional functionality.

The Platform Server is a set of components and services that take care of all the steps necessary to generate, build, package and deploy native C# and Java web applications on top of different application servers, database engines and even operating systems. The Platform Server is composed by two web applications: Service Center and LifeTime. These tools allow the customers to manage, monitor and troubleshoot the generated applications. Including the capabilities to

manage the generation applications across different environments, define security polices and manage the complete life cycle of applications.

In Figure 2.3 an overview of the OutSystems Platform components and their interactions is presented.

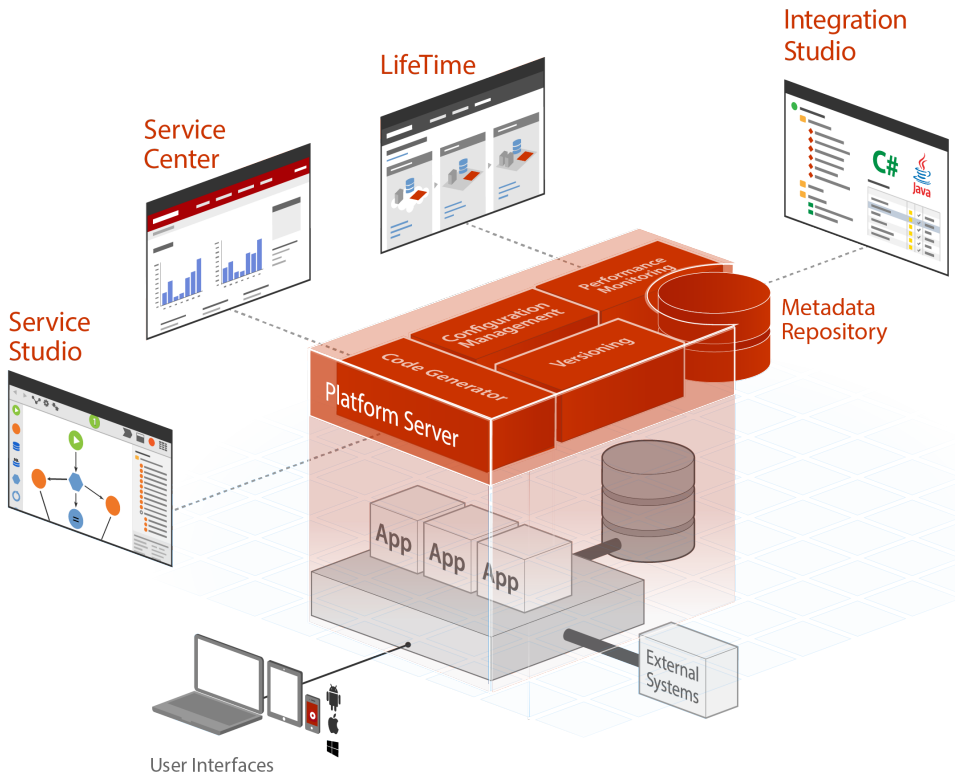

Figure 2.3: The components of the Outsystems Platform

Evaluating the presented description, the OutSystems Platform is a highly configurable system, with different components, developed in multiple technologies and designed to work on top of already existent systems. In the next sections more details will be given in what concerns the platform development and the actual quality assurance process.

2.3 The Development Team and Methodology

Currently, the OutSystems development team is composed by more than one hundred collaborators and continue growing at a fast pace. Each collaborator is changing and developing new code, changing existing test cases and creating test cases daily, several times each day. The OutSystems R&D group is organized in different teams working daily on the development of the OutSystems Platform. Each team is responsible by one or more parts of the Platform's code, including the development and quality assurance of the code.

The code that arrives to production is therefore developed by several different team members.

The process to validate the OutSystems Platform new developments includes the automatic execution of tests in multiple test environments. All test cases are executed in the multiple heterogeneous test environments, so that the software developers may have a good confidence that their changes in the code did not break it or to make sure that they adapted the tests accordingly to the desired changes.

The company uses Agile methodologies in software development, which is also relevant for the context of this work regarding the fact that Agile methodologies require fast feedback loops and frequent software releases. One automated quality assurance process enables the developers to reduce the feedback loop and the human effort required to deliver software with high quality.

Correctly distributing the tests over multiple heterogeneous test environments is a task that is very time consuming and highly prone to human errors, therefore this is the problem that this dissertation aims to solve.

2.4 Platform Development and Quality Assurance

The OutSystems underlying code is mostly written in .NET, using the C# programming language. Some other languages are also used, such as JavaScript and TypeScript. Since the platform developed is available for working in .NET and Java, most of the .Net code is translated into Java using one internal tool.

In what concerns the current quality assurance process, after a developer performs changes in the source code of the platform, a build is created and the platform is installed in different environments with different configurations. After the installation, a set of automated tests is executed in multiple heterogeneous test environments. This process will provide a constant feedback about the impact of changes in the platform to the developer.

The test environments are created with the goal of representing the majority of configurations most commonly used by the company clients, because testing all possible combinations among supported configurations in one valid period of time is not feasible.

The test environments can be composed by one single virtual machine or by more than one. One simple example where it is required more than one virtual machine is when the test case is validating the expected behavior on one mobile device communicating with the server. The OutSystems Platform requirements can be divided in 2 types of requirements: a) Platform Server and b) End User target environments.

The Platform Server runs over two different stacks: .Net and Java. Each one of these stacks is highly configurable, allowing to choose between different application servers, database engines and operating systems. The .Net stack can be installed in the operative system Microsoft Windows Server, while the Java stack can be installed in RedHat, CentOS or Oracle Linux. Another possible configuration is the application server where the Platform Server is deployed. The application server that OutSystems support are Microsoft Internet Information Services for the .Net stack and JBoss, WebLogic or WildFly for the Java stack. At last, the Database Management System that OutSystems support are Microsoft SQL Server for the .Net stack and Oracle or MySQL for the Java and .Net stacks.

Looking at the end user requirements, the Outsystems Platform generates web based applications that need to be validated in several desktop browsers and mobile browsers. Validating the behavior of the generated applications with different resolutions is also one requirement. The newest version of the platform (just released at the time of this writing) will also generate native mobile applications, supporting two different mobile operative systems: iOS and Android. Some tests require to be executed in multiple devices against multiple variations of Platform Server.

Distributing tests with so many different requirements over multiple heterogeneous test environments is highly time consuming and the probability of human errors occur is huge. Being the OutSystems Platform developed for several years, its code base and test suite are increasingly growing. At the moment, OutSystems test suite has more than 10 000 automated tests that need to be distributed over multiple heterogeneous test environments.

Increasing the complexity of this problem, at OutSystems each development team has a different set of test environments. There are several teams working on the same version control branch and having each team a different group of test environments, every time that one new test is created, all the teams will need to select the test environments where the test should be executed (by default, a test will run in all test environments sharing the same branch).

With one test suite with more than 10 000 test cases, the cost of running all test cases in too many test environments is huge, requiring too much resources and consuming too much time. Many of the sets of test environments available for each team are composed by 10 test environments, meaning that the full execution of the test suite will imply the execution of 100 000 test cases (10 000 test cases multiplied by 10 test environments). The wasted effort to troubleshoot the test cases that fail, because the test environment do not have all the required configurations to execute the test case, is huge.

The human effort required to keep the tests properly distributed over all the heterogeneous environments is cumbersome. The description presented of the different variations of the platform represents the current requirements of the platform. These requirements have been changing with time, and the test suites and test environments are always being updated, increasing the human effort required. This is the biggest motivation for this work: being able to have automated systems that reduce the human effort even when software is large and complex, and assure that all the test cases are executed in the proper test environments independently of the requirements evolution.

2.5 Conclusion

In the current chapter, an overview of the industrial context for this dissertation was presented. Specifically, OutSystems and the product it develops were described, and also how the company develops it and how automated tests play an important role in guaranteeing the quality of the software.

Understanding this context was crucial to understand where the human effort could be reduced, and it became clear that optimizations should emerge from one of the Quality Assurance (QA)

steps, which is automated test cases distribution.

Distributing the tests over multiple heterogeneous test environments is time consuming and prone to human errors, therefore solving this problem will reduce the required human effort and avoid the human errors propagation.

The description of the OutSystems context that was made in this chapter also helps to clarify why this research has focused on optimizing test distribution. This is the main reason why this chapter comes before the State of the Art (Chapter 4), because this industrial contextualization is crucial for pointing the correct research directions.

Chapter 3

State of the Art

This chapter presents the current state of the art on areas that are related with test distribution and minimization. To be more concrete, research areas that are related with optimizing the distribution of test suites over multiple environments are interesting for the context of the problem that is being solved as are minimization techniques that enable to minimize the number of test environments where each test will execute. It is of great interest to understand what has been done so far in both academic and industrial contexts regarding the goal of reducing the human effort spent in test management.

3.1 Introduction

A common aim within the software industry is to ensure the delivery of high quality software. The creation and execution of tests are crucial in the process of guaranteeing high quality. These tests intend to validate the software logic, integrations between all the parts that compose the system under test and simulate the software behavior as a whole.

Many modern software systems are designed to be highly-configurable, increasing the difficulty of the validation. In these cases, a proper validation is only achieved by executing tests in multiple heterogeneous environments.

Having to assure quality in multiple heterogeneous environments creates several challenges related with test management, mainly to select the test environments where each test case should execute. With the test suites growing at a large pace, the human effort required to keep the test suites up-to-date and being executed in the proper test environments is cumbersome and highly prone to human errors.

The problem of selecting the test environments where each test case should (or not) execute, may be divided in two smaller problems: distribute the test over multiple environments and minimize the number of test environments required to execute the test cases. These two problems have been studied in the past and it is possible to find multiple published research work in the literature focused in solving these problems individually.

The different main topics found during this research work, that address the final goal of selecting the test environments where each test case should execute, may be divided in the following two main categories:

1. Distribute the test cases over multiple test environments, being the related work found in the literature described in Section 3.2;
2. Test suite minimization techniques, introduced in Section 3.3.

3.2 Test Distribution

Test Distribution is one research field focused in improving the usage of the available test environments. Until recently, the costs of allocating dedicated computational resources for testing were high. However, the current proliferation of Virtualization [CB10], Grid [FK99] and Cloud [Wei07] technologies have drastically reduced the hardware maintenance expenses, being more affordable to most companies have more resources allocated to validate the expected behavior of the products that are developing. As a result, the research in this area increased.

In 2001, Kapfhammer first described the conceptual foundation, design and implementation of an approach to distribute the execution of test suites across multiple machines [Kap01]. Such strategy aims to reduce time costs by operating on the principle that large workloads can be divided into smaller ones, which are then executed concurrently.

The tool built to put this proposal into practice was Joshua [Kap01], that is an extension of the well known JUnit [BG98] testing framework, a test description and automation framework for Java software systems.

While Kapfhammer's work has been one great breakthrough, the absence of capability to handle heterogeneous system configurations is definitely a downside.

Although software components may be exhaustively tested in a development configuration before going to production, the ability to run tests in a variety of system configurations is not only useful but also mandatory. Executing tests in a variety of system configurations allow to find specific errors in the coexisting of different configurations and software defects that would otherwise be undetected if the software was tested within an homogeneous configuration [DCBM06].

With this fact in consideration, other approaches seek to bridge this gap by exploring the good features provided by general purpose Grid software. Grid computing, which is characterized by large-scale sharing and collaboration of dynamic resources, has quickly become a mainstream technology in distributed computing [FK99].

The high levels of parallelism provided by a Grid are useful to reduce the time spent in the test execution, make the testing process of very time consuming test suites a faster task. Also, being a highly heterogeneous environment, the Grid can be used to increase the confidence of the results obtained from the test suite [DCBM06].

Based on the previous observations, three tools have been developed to explore the characteristics of Grids for software testing: Metronome [PCG+06], GridUnit [DCBM06] and Selenium [Dav12]. Metronome was built to run on top of the Condor distributed batch system [LML87] [TTL05].

Metronome allows the user to submit a build and test routine, creating internally in the framework a single job to Condor that multiply by multiple Condor jobs for each platform targeted by the routine. Condor ensures that these jobs are executed on test environments that satisfy the requirements of the job. Figure 3.1 represents the Metronome architecture (presented originally as NSF Middleware Initiative (NMI) Build & Test System).

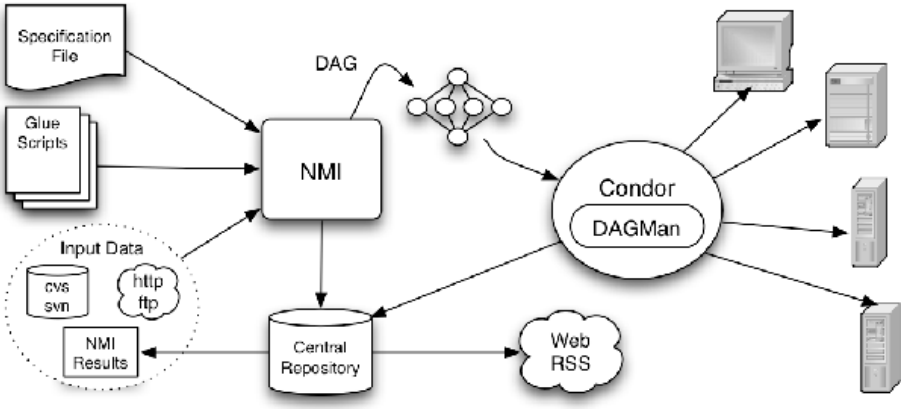

Figure 3.1: NMI Framework Architecture. Picture taken from [PCG+06]

Condor uses a central element responsible of planning and scheduling jobs for execution, designated by negotiator. Each test environment provides the negotiator with a list of the available capabilities in the test environment, including: system properties, pre-installed software and current activity.

After Condor collects this information from both involved (job and test environments), the negotiator match jobs with test environments that satisfy each others requirements. The matched job and test environment communicate directly with each other and then the job is transferred by Condor to the test environment for execution.

The framework will warn users if they submit a job (build or test) with a requirement that cannot be satisfied by any test environment.

The second tool, GridUnit [DCBM06], was developed to distribute the execution of JUnit test suites and, despite benefiting from the Grid computational power, the underlying features and principles are basically those of Joshua [Kap01].

The main features of GridUnit are those appointed by Kapfhammer [Kap01] as important aspects that a test distribution tool must consider in order to improve the cost effectiveness of the testing process:

- **Transparent and Automatic Distribution:** GridUnit consider each test as an independent task, scheduling its execution on the grid without any user intervention.
- **Test Case Contamination Avoidance:** Each test is executed using one different test environment, preventing that the execution of a test changes the normal outcome of other tests.
- **Test Load Distribution:** The job scheduler achieves load distribution by allocating each test for execution in the first available test environment.
- **Test Suite Integrity:** GridUnit test runner executes each unit test as an independent task. For each test, it creates an instance of the *TestCase*, calls the *setUp()* method, calls the *testMethod()*, calls the *tearDown()* method and then destroys the instance.

- **Test Execution Control:** GridUnit graphical user interface (represented in Figure 3.2) provides controls to start and stop the execution of the tests of a given test suite. It also monitors the execution of the tests and presents the result of the execution of each test.

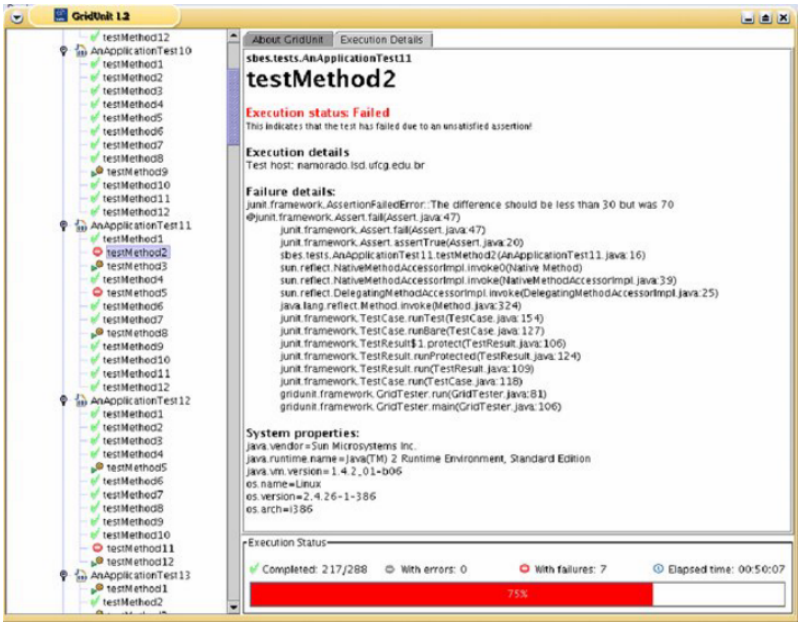

Figure 3.2: GridUnit graphical user interface [DCBM06]

The third tool, Selenium [Dav12] is a set of different software tools designed to automate testing of web applications. All of these tools together provide a rich set of testing functionality, compatible with all the most widespread software platforms and web browsers. Selenium is composed by three different components: Selenium IDE, Selenium WebDriver and Selenium Grid.

The Selenium IDE is a prototyping tool for creating automated tests, that is implemented as a plugin for the web browser Mozilla Firefox. It provides simple interface that contains the record functionality, enabling to record the steps performed by a user when using the web application.

With basic knowledge of HTML syntax and the Selenium IDE it is possible to create automated tests very fast and easily, without any programming skills. For completeness, one example of a simple test recorded in the Selenium IDE is shown on Figure 3.3, showing the simple test of logging into the web application.

From tests created by this method the Selenium IDE can also generate scripts in programming languages C#, Java, Python and Perl.

The Selenium IDE is also able to generate scripts for the following unit test frameworks: JUnit (Java), NUnit (.Net), RSpec (Ruby), unittest (Python) or TestNG (Java).

Selenium WebDriver is the latest framework from the Selenium family. WebDriver uses a controller (designated by driver), that allows WebDriver to execute commands directly with the web browser, using browser native bindings for automate the test execution.

Selenium WebDriver allows to define the desired capabilities that are required to execute the

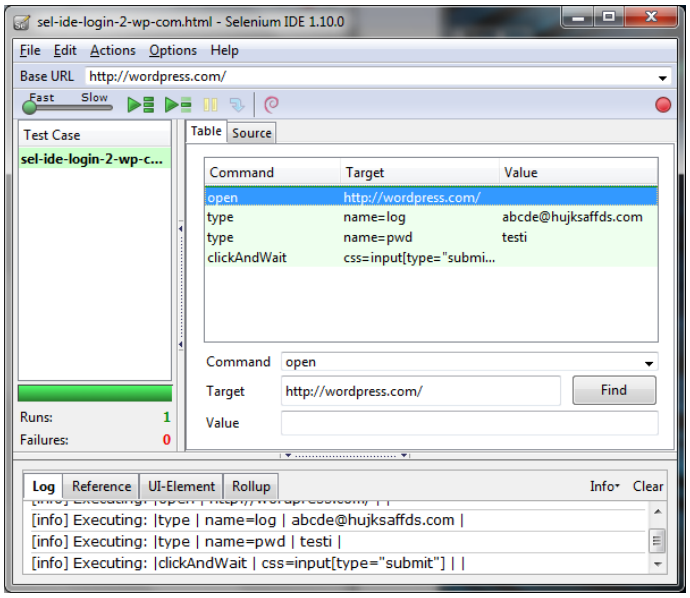

Figure 3.3: Selenium IDE

test case in the test specification, throwing one error if the test is select to execute in one test environment that do not fulfil all the desired capabilities.

At last, Selenium Grid is a tool enabling parallel execution of various automated tests using multiple remote test environments with different configurations. Selenium Grid allows to execute large test suites that are highly time consuming in a faster way, distributing the tests to be executed by multiple test environment with different configurations. Grid consists of so-called *Hub* and one or more *Nodes*.

The *Hub* maintains configuration information for each authorized *node* and it is responsible for routing the test executions to the *node* with the correct configuration (e.g. Windows 10 with Chrome web browser).

With desired capabilities described in the test specifications and the available capabilities described in each node, the *Hub* is able to select the test environment that has all the desired capabilities on the test specification to execute the test.

After this overview about general research works on the test case distribution problem, some works have presented solutions that partially solve our problem.

Current test distribution techniques allow to define requirements (or desired capabilities) that the test environment must fulfil to be able to execute the test specification. Unfortunately, the current approaches only allows to describe simple requirements, not allowing to describe one rational with the several factors (e.g.: multiple operative systems, multiple browsers and only one resolution) that may influence the System Under Test (SUT) expected behavior.

The goal of this dissertation is to execute each test specification in the minimum set of test environments that exercise all the factors described during the creation of the test specification.

With the goal of executing the tests in the minimum set of test environments, minimization techniques were also studied, which will be presented in Section 3.3. The goal is to present

techniques that allows to describe different factors that influence each test, in such a way, that enables both the developers and the test execution engine to understand the rationale behind the selected test environments to execute each test case.

3.3 Test Minimization

In many cases (for example, during integration testing), the SUT has a number of factors (also designated as parameters) that may influence the behavior of the test. Each factor may have multiple values. For instance, we may consider as System Under Test (SUT) one web application that may be used over different browsers with different resolutions. In this case, the browser and resolution are the factors that may influence the system behavior and the values that they can take are the values of each factor.

To test such a system, we may try all combinations of values and see if there is any error. But the cost might be too high, if the factors or the amount of values that influence is high, because the number of test executions for each test case will be cumbersome. Fortunately, experiences indicate that many systems exhibit erroneous behaviors due to the combination of just a few factors' values. Thus it is not necessary to check the combination of all factors. This motivated the use of Combinatorial Interaction Testing (CIT), whose goal is to cover as many combinations of factors as possible, using a small number of combinations.

CIT techniques are a useful cost-benefit compromise that enables to minimize the number of combinations without drastically compromising the functional coverage. Assuming that a test function has N factors(F) given in a set $F = \{f_1, f_2, ..., f_N\}$ with each factor having multiple possible values $\{f_i\} = \{v_{i,1}, v_{i,2}, ..., v_{i,M}\}$.

Beside the set of factors ($F = \{f_1, f_2, ..., f_N\}$) and respective values ($\{f_i\} = \{v_{i,1}, v_{i,2}, ..., v_{i,M}\}$) for each factor, CIT techniques receive one additional information, that is the order (T), being T one integer defined in the interval $1 \leq T \leq N$.

With $T = 1$, the goal of combinatorial techniques is to generate a set of combinations R, where each combination contains N values, one for each factor f_i and collectively, all combinations in R have all values (v) of each factor at least once ($T = 1$).

When $T = 2$, the goal of combinatorial techniques is to generate a new set of combinations R, also containing each combination N values, one for each factor f_i and collectively, all combinations in R cover all possible pairs ($T = 2$) of values (each value belonging to different factors).

The same applies for any T, higher T increments the number of generated combinations. When $T = N$, being N the total number of factors, the combinatorial techniques will generate a set of combinations R with all the combinations between the values of each factor.

During the last 30 years, CIT has been applied in many areas. Wallace and Kuhn [WK01] analysed software-related failures of some medical devices that led to recalls by the manufacturers. They found that many flaws could have been detected by testing all pairs ($T = 2$) of factors settings, reducing significantly the number of test executions. The applications used during this study are normally small to medium sized embedded systems.

Khun et al. [RMR02] analysed 329 error reports generated during the development and integration testing of a large distributed system developed at Nasa Goddard Space Flight Center. The application is a data management system that gathers and receives large quantities of scientific data. They found that the number of conditions required to trigger a failure is at most six ($T = 6$).

Mandl described using orthogonal arrays in the testing of a compiler [Man85] for testing compilers— yields, the informational equivalent of exhaustive testing at a fraction of the cost. The method has been used successfully in designing some of the tests in the Ada Compiler Validation Capability (ACVC) test suite.

D. Cohen et al. described the automatic efficient test generator, AETG [CDFP97], that provides t-way interaction coverage between program inputs. They describe a study in which they capture code coverage for the Unix utility program sort. Code coverage metrics are presented for several models of the configuration space, but the authors do not examine all the possible configurations, rather they focus only on the sample selected using it to define the test cases.

Later, Dunietz et al. [DES+97] examine code coverage for various orders of interaction testing. Again the focus is on the sample of the configuration test suite, rather than on using the entire set of configurations.

Kevin Burr and William Young [BY98] have used AETG [CDFP97] to generate small efficient sets of test vector, which a test driver can then execute automatically. They used the code coverage to indicate missing functionality from AETG's. With the experiments done on Nortel's internal e-mail system, they were able to cover 97% of branches with less than 100 test cases, as opposed to 27 trillion exhaustive test cases.

Kuhn et al [RMR02] investigated error reports from two large open-source projects: the Mozilla web browser and the Apache web server. They have achieved two main conclusions:

1. more than 95% of errors in the softwares studied would be detected by test cases that cover all 4-way combinations of values ($T = 4$);
2. browser and server software were similar in the percentage of errors detectable by combinations of degree 2 ($T = 2$) through 6 ($T = 6$).

Later, proposed by Tatsumi, in the paper on the Test Case Design Support System (used in one industrial context, at Fujitsu Ltd [Tat87]), talks about two standards for creating test arrays: one with all combinations covered exactly the same number of times (orthogonal arrays) and one with all combinations covered at least once.

The previous empirical studies show that software failures in many domains were caused by combinations of relatively few conditions, even when the number of factors involved is huge.

However, as shown by Smith et al. [SFM00] and Bach and Shroeder [BS04], pairwise ($T = 2$) must be used appropriately and with caution, like any technique that is reducing the number of combinations to be executed.

Because the problem of finding a minimal array that covers all pairwise ($T = 2$) combinations of a given set of test factors is NP complete [LT98], a considerable amount of research has

understandably gone into efficient creation of such arrays. Several strategies were proposed in an attempt to minimize the number of tests that were produced [GOA05].

Authors of these combinatorial test-case–generation strategies often describe additional considerations that must be taken into account before their solutions become practical. In many cases, they propose methods of handling these in the context of their generation strategies. Tatsumi [Tat87] mentions constraints as a way of specifying unwanted combinations (or, more generally, dependencies among test factors).

Sherwood [She94] explores adapting t-wise strategy to describe limitations of the test domain, meaning that are combinations that are impossible to be successfully executed in the context of the given SUT.

Cohen et al. [CDFP97] describe seeds that allow specifying combinations that must appear in the generated combinations and covering combinations with mixed-strength arrays. The seeds allow to explicitly specify important combinations and can be used to minimize change in the output when the test domain description is modified ant the results need to be regenerated.

Later, Jacek Czerwonka presented PICT [Cze], a test-case–generation tool that has been in use at Microsoft Corporation, implementing both the t-wise–testing strategy and features that make the strategy feasible in the practice of software testing. This tool focus specially in the usability of the Combinatorial Interaction Testing (CIT) techniques.

PICT is a command line tool, that receives one plain text file (combinatorial model) and the order of the factors interaction to that model (T).

Listing 3.1 represents one combinatorial model description to PICT, where three factors are described: *A*, *B* and *C*. Using the example presented before, the factor *A* with the possible values $\{1, 2\}$, the factor *B* with the possible values $\{X, Y\}$ and the factor *C* with the possible values $\{J, K, L\}$.

```
A:      1, 2
B:      X, Y
C:      J, K, L
```

Listing 3.1: Combinatorial Model provided to PICT

With this inputs, the generation process in PICT is composed by two main phases: preparation and generation.

In the preparation phase, PICT computes all of the information that is necessary for the generation phase, that is the set of all parameter interactions (P) that must be represented in the set of generated combinations (R). Each combination (defined in PICT as slot) of values (v) to be covered is reflected in a parameter-interaction structure (P), that is the result from the preparation phase. Figure 3.4 represents the parameter interaction set (P) created by PICT at the end of the preparation phase.

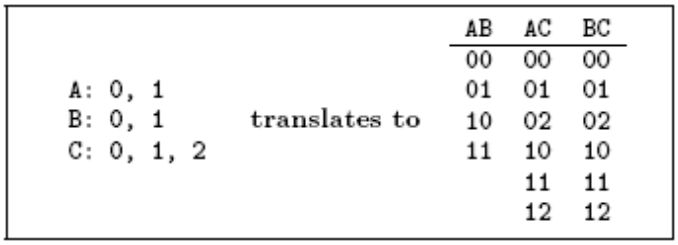

Figure 3.4: Parameter Interaction Structure in PICT [Cze]

The generation phase works with a greedy heuristic that is building one test case at a time, locally optimizing the solution in one deterministic way.

The generation algorithm does not assume anything about the combinations to be covered. It operates on a set of parameter interactions (P) that is produced in the preparation phase, generating the minimum set of combinations (R) that fulfil the combinatorial model provided with the respective T.

Executing PICT using the command line interface, sending T as one command line argument returns one minimum set of combinations (R) that fulfil the combinatorial model described with the respective T. Table 3.1 represents one example of the set of combinations (R) returned by PICT, when being executed with the combinatorial model described in Listing 3.1 and with $T = 2$.

Table 3.1: PICT obtained result

A	B	C
1	X	J
2	Y	J
1	Y	K
2	X	K
1	X	L
2	Y	L

Analysing the Table 3.1, it is possible to validate that the generated set of combinations (R) cover all possible pairs ($T = 2$) of each factor.

The goal of the techniques studied and presented in this Section, was to find one technique that allows to describe the different factors that influence each Test Specification (TS), in such a way, that enables both the developers and the test execution engine to understand the rationale behind the selected test environments to execute each test case.

PICT flexibility regarding the combinatorial model description, allows to describe the factors that may influence the behavior of one System Under Test (SUT), in one descriptive format that one test execution engine is able to parse and one developer is able to understand. With the description of the factors (e.g. browser, operating system and others) that may influence the behavior of one System Under Test (SUT), it is possible to generate the minimum set of combinations where each Test Specification (TS) should be executed.

Our proposed solution, described in Chapter 4, uses one extend version of PICT to generate the minimum set of combination that must be exercised for each Test Specification (TS). For each combination that is generated, later designated by Required Combinations (RC), creates one

new Test Case (TC) and selects only one heterogeneous Test Environment (TE) to execute the previously created Test Case (TC). The generated combination is used as requirement to select the Test Environment (TE) where each created Test Case (TC) will be selected to execute.

3.4 Conclusion

The current chapter presented an overview on several research areas that are related with the problem of selecting the test environments where each test specification should execute.

There are some researches that focus on test distribution, which aim to enable the execution of several test cases at the same time. Test distribution divides large workloads into smaller ones, which are then solved concurrently over a set of machines.

The test distribution solutions that were presented enable to distribute the test cases over multiple heterogeneous test environments. Furthermore, they enable to describe requirements that the test environments must fulfil to be able to execute the test cases.

The test requirements that are possible to specify are not flexible, do not allowing to describe one rational with the factors that influence the System Under Test (SUT) behavior. Besides, the presented techniques select one Test Environment (TE) for each Test Specification (TS), while the goal of this dissertation is to select the minimum set of test environments that enable to validate the expected behavior of the Test Specification (TS) execution.

Combinatorial techniques allows to describe the factors that are relevant to be exercised and their respective values in one combinatorial model, generating the minimum set of combinations that fulfil the combinatorial model accordingly with the respective T.

This type of description allows one algorithm to evaluate the factors that were described in the Test Specification (TS), and calculate the minimum set of combinations that fulfil the relevant factors described by the developer.

With the goal of executing the test specification in the minimum number of available test environments, one technique was designed (presented in Chapter 4) that enables to describe the relevant factors to one test specification using combinatorial models, commonly used in Combinatorial Interaction Testing (CIT) techniques.

The proposed technique generate the minimum set of combinations required to fulfil the combinatorial model and uses the generated combinations to create multiple test cases, having each test case one combination as requirement to select the test environment where each test case is selected to execute.

Chapter 4

Proposed Solution

This chapter presents the solution we propose to solve the problem of automated test case distribution that was defined on Chapter 1.

Our solution is able to deal with modern software systems and the actual needs of software developers, but generalized in such a way that is does not contain any programming language or environment dependent characteristics.

4.1 Introduction

The problem to be solved in the context of this dissertation is not only a standard test case distribution problem. It is also a test case distribution problem whose solution must conform to the context of highly configurable software systems and developed with modern software development practices. These generic requirements arise from the need for a solution that is efficient, scalable and deterministic, meaning that the selected test environments where each Test Case (TC) is selected to execute always exercise the code under test with all the required variations.

The heterogeneity of many software systems and the usage of several different technologies require that the proposed test case distribution technique must rely on some abstractions and programming language independent assumptions. Therefore, the only assumption that the presented solution makes is that there must be a way to add information to the test specifications and retrieve the configurations from the available test environments. Apart from that, all the characteristics of the proposed test distribution technique render that solution as suitable to multiple different software development contexts of large scale and complexity.

The main goal of the current chapter is to present the solution for the test case distribution problem over heterogeneous test environments, that aims to solve a real generic industrial problem and not only a problem that is specific of a particular company.

The proposed solution, described in detail in Section 4.3, allows to describe the factors that influence each Test Specification (TS) using one combinatorial model, designated by Relevant Factors (RF). With the Relevant Factors (RF) described in the Test Specification (TS), the proposed solution will generate the minimum set of Required Combinationss where each Test Specification (TS) must be executed. For each generated required combination, the proposed solution creates one new Test Case (TC) that uses the Required Combinations (RC) as requirement to select the Test Environment (TE) where the Test Case (TC) is selected to execute.

The test discovery algorithm will run on all Test Environments, it is deterministic and it returns

similar Test Cases on all Test Environments. The only difference in the Test Cases in the different Test Environments is one attribute, the attribute represents one of the following alternatives:

1. the Test Case (TC) is selected to be executed on the Test Environment (TE);
2. the Test Case (TC) is selected to be executed on other Test Environment (TE);
3. does not exist any Test Environment (TE) that is capable of execute the Test Case (TC).

The proposed solution is one test discovery algorithm, that uses one combinatorial algorithm to generate the minimum set of combinations that fulfil one combinatorial model. This means, that the combinatorial algorithm is one dependency that is required by the test discovery algorithm and is used multiple times during the test discovery algorithm execution.

The initial idea of this work was to use PICT algorithm [Cze] (presented in Section 3.3) as combinatorial algorithm, using one existing implementation that is open source [Mic15]. Unfortunately using PICT as combinatorial algorithm produces unwanted results to the test discovery algorithm (explained in Section 6.2). Instead, we proposed one extended version of PICT as combinatorial algorithm, explaining the modifications done to the original PICT algorithm in Section 4.2.

The proposed solution can be described in two parts, the test discovery algorithm and the combinatorial algorithm. In this chapter, the two algorithms are described in the following order:

1. **Combinatorial algorithm** - has the responsibility of generate the combinations that fulfil one Combinatorial Model, presented in the Section 4.2. It is explained first because it is one dependency of the test discovery algorithm;
2. **Test discovery algorithm** - is the algorithm that creates the required Test Cases to fulfil the Relevant Factors (RF) described in one Test Specification (TS) and selects the Test Environment (TE) where each Test Case (TC) will execute, explained in the Section 4.3.

4.2 Combinatorial Algorithm

The combinatorial algorithm chosen to be used in this implementation was one extended version of the PICT algorithm [Cze]. This section will justify this choice based on the requirements of the described solution, describing the behavior of the original PICT algorithm and explain the changes done to the PICT algorithm.

The main goal of the combinatorial algorithm is to generate the minimum set of combinations (R) to fulfil one Combinatorial Model (CM) (described in Subsection 4.2.1). The combinations generation is done in two main phases that are done sequentially: preparation (explained in Subsection 4.2.1) and generation (described in Subsection 4.2.3).

4.2.1 Combinatorial Model

The combinatorial algorithm receives the description of the Combinatorial Model (CM) in the most customizable way possible, allowing to generate the minimum set of combinations (R) that fulfil the provided Combinatorial Model (CM). The Combinatorial Model (CM) is composed by five elements, two mandatory (factors and T) and the remaining are optional. The CM is represented by:

$$CM = \{factors, T, subModels, constraints, availableCombinations\}$$

The first element of this description is the $factors$, that is one set containing the factors $\{f_1, f_2, ..., f_n\}$ and the respective possible values $f_i \mapsto \{v_{i,1}, v_{i,2}, ..., v_{i,m}\}$ for each factor:

$$\begin{aligned} factors = \{ & \\ & f_1 \mapsto \{v_{1,1}, v_{1,2}, ..., v_{1,m}\} , \\ & f_2 \mapsto \{v_{2,1}, v_{2,2}, ..., v_{2,o}\} , \\ & \dots , \\ & f_n \mapsto \{v_{n,1}, v_{n,2}, ..., v_{n,p}\} \\ \} & \end{aligned}$$

The second element, T, represents the order of factors interactions $1 \leq T \leq N$, N being the number of $factors$ described in the Combinatorial Model (CM).

This element allow to define the strength of the generated combinations, meaning that lower T generates less combinations and higher T will generate more combinations. If $T = 1$, the generated combinations produced by the algorithm contain each value at least once.

For example, considering one CM , already used as example in Section 3.3, containing three factors: *A*, *B* and *C*. The factor *A* with the possible values $\{1, 2\}$, the factor *B* with the possible values $\{X, Y\}$ and *C* with the possible values $\{J, K, L\}$. The Combinatorial Model (CM) is represented by:

$$\begin{aligned} CM = & \quad \{factors, T, \emptyset, \emptyset, \emptyset\} \\ factors = \{ & \\ & A \mapsto \{1, 2\} , \\ & B \mapsto \{X, Y\} , \\ & C \mapsto \{J, K, L\} \\ \} & \\ T = & \quad 1 \end{aligned}$$

With the parameter interaction equals to one ($T = 1$) the combinatorial algorithm will produce three combinations where each value will appear at least once. There are multiple valid solutions that fulfil the CM previously described (with $T = 1$). Two possible valid results ($combinations_A$ and $combinations_B$) for this example are the following:

$$
\begin{aligned}
combinations_A = \quad & \{c_1, c_2, c_3\} \\
c_1 = \{ & \\
& (A = 1), (B = X), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 2), (B = X), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 2), (B = X), (C = K) \\
\} &
\end{aligned}
\qquad
\begin{aligned}
combinations_B = \quad & \{c_1, c_2, c_3\} \\
c_1 = \{ & \\
& (A = 1), (B = X), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 2), (B = Y), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 1), (B = Y), (C = L) \\
\} &
\end{aligned}
$$

If the parameter interaction is two ($T = 2$) instead, the combinations generated need to contain all pairs of values between different factors. The result has six combinations instead of three, because the coverage of parameter interactions has increased. For example, two valid results are the following:

$$
\begin{aligned}
combinations_A = \quad & \{c_1, c_2, c_3, c_4, c_5, c_6\} \\
c_1 = \{ & \\
& (A = 1), (B = X), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 2), (B = Y), (C = J) \\
\} & \\
c_3 = \{ & \\
& (A = 1), (B = Y), (C = K) \\
\} & \\
c_4 = \{ & \\
& (A = 2), (B = X), (C = K) \\
\} & \\
c_5 = \{ & \\
& (A = 1), (B = X), (C = L) \\
\} & \\
c_6 = \{ & \\
& (A = 2), (B = Y), (C = L) \\
\} &
\end{aligned}
\qquad
\begin{aligned}
combinations_B = \quad & \{c_1, c_2, c_3, c_4, c_5, c_6\} \\
c_1 = \{ & \\
& (A = 1), (B = Y), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 2), (B = X), (C = J) \\
\} & \\
c_3 = \{ & \\
& (A = 1), (B = Y), (C = K) \\
\} & \\
c_4 = \{ & \\
& (A = 2), (B = X), (C = K) \\
\} & \\
c_5 = \{ & \\
& (A = 1), (B = Y), (C = L) \\
\} & \\
c_6 = \{ & \\
& (A = 2), (B = X), (C = L) \\
\} &
\end{aligned}
$$

With $T = 3$, only exists one possible result because $T = N$, N being the total number of factors. In this case, the result will be all the possible combinations of the three factors is the following:

$$
\begin{aligned}
combinations = \quad & \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}\} \\
c_1 = \{ & \\
& (A = 1), (B = X), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 2), (B = X), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 1), (B = Y), (C = L) \\
\} & \\
c_4 = \{ & \\
& (A = 2), (B = Y), (C = J) \\
\} & \\
c_5 = \{ & \\
& (A = 1), (B = X), (C = K) \\
\} & \\
c_6 = \{ & \\
& (A = 2), (B = X), (C = L) \\
\} & \\
c_7 = \{ & \\
& (A = 1), (B = Y), (C = J) \\
\} & \\
c_8 = \{ & \\
& (A = 2), (B = Y), (C = K) \\
\} & \\
c_9 = \{ & \\
& (A = 1), (B = X), (C = L) \\
\} & \\
c_{10} = \{ & \\
& (A = 2), (B = X), (C = J) \\
\} & \\
c_{11} = \{ & \\
& (A = 1), (B = Y), (C = K) \\
\} & \\
c_{12} = \{ & \\
& (A = 2), (B = Y), (C = L) \\
\} &
\end{aligned}
$$

The Combinatorial Model (CM) is composed by five elements, the two previously described are mandatory ($factors$ and T) and the remaining are optional ($subModels$, $constraints$ and $availableCombinations$). Following, the optional elements will be described because they will be used in the test discovery algorithm explained further, in Section 4.3.

The third element of the Combinatorial Model (CM) is the $subModels$ that are important to limit the combinatorial explosion of certain factors. There are scenarios where the interactions between some factors are more relevant than others (the importance of this element will be further described in Section 4.3, explained in what cases is this element used).

$$
\begin{aligned}
subModels = \{ & \\
& \{T_1, \{factors_1\}\} , \\
& \dots , \\
& \{T_n, \{factors_n\}\}
\end{aligned}
$$

The algorithm presented in PICT [Cze] already supports $subModels$, allowing to define one dif-

ferent order for one subset of factors. $T = 2$ for two factors interactions (for example, the $factors\ A$ and B), being that definition represented bellow:

$$\begin{aligned} CM = & \{factors, T, subModels, \emptyset, \emptyset\} \\ factors = \{ & \\ & A \mapsto \{1, 2\}\ , \\ & B \mapsto \{X, Y\}\ , \\ & C \mapsto \{J, K, L\} \\ \} & \\ T = & \ 1 \\ subModels = \{ & \\ & \{2, \{A, B\}\} \\ \} & \end{aligned}$$

There are multiple valid solutions that fulfil previous CM, two possible valid results ($combinations_A$ and $combinations_B$) for the previous Combinatorial Model (CM) description are represented bellow:

$$\begin{aligned} combinations_A = & \{c_1, c_2, c_3, c_4\} \\ c_1 = \{ & \\ & (A = 1), (B = X), (C = J) \\ \} & \\ c_2 = \{ & \\ & (A = 2), (B = Y), (C = J) \\ \} & \\ c_3 = \{ & \\ & (A = 1), (B = Y), (C = K) \\ \} & \\ c_4 = \{ & \\ & (A = 2), (B = X), (C = L) \\ \} & \end{aligned} \qquad \begin{aligned} combinations_B = & \{c_1, c_2, c_3, c_4\} \\ c_1 = \{ & \\ & (A = 1), (B = Y), (C = J) \\ \} & \\ c_2 = \{ & \\ & (A = 2), (B = X), (C = J) \\ \} & \\ c_3 = \{ & \\ & (A = 1), (B = X), (C = K) \\ \} & \\ c_4 = \{ & \\ & (A = 2), (B = Y), (C = L) \\ \} & \end{aligned}$$

Regarding the forth element, designated by $constraints$, it is important to exclude unwanted combinations from the generated combinations.

The algorithm presented in PICT [Cze] already allows to specify constraints in the form of propositional forms, using **IF-THEN** statements to describe the limitations of a particular domain.

$$\begin{aligned} constraints = \{ & \\ & if(f_1 = v_{1,1}) then (f_2 = v_{2,1}), \\ & if(f_2 = v_{2,2}) then (f_3 \in v_{3,1}, v_{3,2}), \\ & \ldots \\ & if(f_n = v_{n,i}) then (f_m = v_{m,j}) \\ \} & \end{aligned}$$

For example, if $(A = 1)$ is only possible to combine with $(B = Y)$ then it is possible to add that constraint to the Combinatorial Model (CM) in the following way:

$$
\begin{aligned}
CM = \quad & \{factors, T, subModels, constraints, \emptyset\} \\
factors = \{ & \\
& A \mapsto \{1, 2\} \ , \\
& B \mapsto \{X, Y\} \ , \\
& C \mapsto \{J, K, L\} \\
\} & \\
T = \quad & 1 \\
subModels = \{ & \\
& \{2, \{A, B\}\} \\
\} & \\
constraints = \{ & \\
& if(A = 1) then(B = Y) \\
\} &
\end{aligned}
$$

With the previous constraint added, it is not possible to have $(A = 1)$ and $(B = X)$ in the same combination of the generated combinations.

The model continues to require all values at least once and all pairs for the factors interaction (the $factors$ A and B). With this requirements in the CM description, two possible valid results ($combinations_1$ and $combinations_2$) are represented bellow:

$$
\begin{aligned}
combinations_A = \quad & \{c_1, c_2, c_3\} \\
c_1 = \{ & \\
& (A = 2), (B = Y), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 1), (B = Y), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 2), (B = X), (C = L) \\
\} &
\end{aligned}
\qquad
\begin{aligned}
combinations_B = \quad & \{c_1, c_2, c_3\} \\
c_1 = \{ & \\
& (A = 2), (B = X), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 1), (B = X), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 2), (B = Y), (C = L) \\
\} &
\end{aligned}
$$

At last, the fifth element, is the $availableCombinations$. The $availableCombinations$ is one set of combinations received in the Combinatorial Model (CM) and is represented bellow:

$$
\begin{aligned}
availableCombinations = \quad & \{c_1, ..., c_m\} \\
c_1 = \{ & \\
& (f_1 = v_{1,i}), ..., (f_m = v_{m,j}), \\
\} & \\
& ... \\
c_m = \{ & \\
& (f_1 = v_{1,k}), ..., (f_m = v_{m,l}) \\
\} &
\end{aligned}
$$

This element will enable to generate the minimum set of combinations R that fulfil the Combinatorial Model (CM), adding first combinations to the set R that are not contained in the set $availableCombinations$ and only then adding combinations to the set R that are contained in the set $availableCombinations$. The importance of this element will be explained with more detail in Section 4.3.

This element is the only element that PICT implementation does not receive and is the element that will allow to achieve the expected results in the test discovery algorithm. This results

achieved by adding this element will be analysed and further discussed in Section 6.2, being compared the obtained results when using our proposed test discovery algorithm or using the original PICT implementation.

For example, if one combination ($\{(A = 2), (B = X), (C = J)\}$) is contained in the set of $availableCombinations$, the Combinatorial Model (CM) is represented in the following way:

$$
\begin{aligned}
CM = \quad & \{factors, T, subModels, constraints, availableCombinations\} \\
factors = \{ & \\
& A \mapsto \{1, 2\}\ , \\
& B \mapsto \{X, Y\}\ , \\
& C \mapsto \{J, K, L\} \\
\} & \\
T = \quad & 1 \\
subModels = \{ & \\
& \{2, \{A, B\}\} \\
\} & \\
constraints = \{ & \\
& if(A = 1) then(B = Y) \\
\} & \\
availableCombinations = \{ & \\
& \{(A = 2), (B = X), (C = J)\}, \\
\} &
\end{aligned}
$$

With this requirements in the CM description, two possible valid results ($combinations_A$ and $combinations_B$) are represented bellow:

$$
\begin{aligned}
combinations_A = \quad & \{c_1, c_2, c_3\} \\
c_1 = \{ & \\
& (A = 2), (B = Y), (C = L) \\
\} & \\
c_2 = \{ & \\
& (A = 1), (B = Y), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 2), (B = X), (C = J) \\
\} & \\
combinations_B = \quad & \{c_1, c_2, c_3\} \\
c_1 = \{ & \\
& (A = 2), (B = X), (C = J) \\
\} & \\
c_2 = \{ & \\
& (A = 1), (B = X), (C = K) \\
\} & \\
c_3 = \{ & \\
& (A = 2), (B = Y), (C = L) \\
\} &
\end{aligned}
$$

The algorithm presented in PICT do not receives this element in the Combinatorial Model (CM) description. We adapted the PICT algorithm to receive this element in the Combinatorial Model (CM) description and generate the combinations giving preference to combinations that are contained in the $availableCombinations$.

In the following Sub Sections, the combinatorial algorithm will be described, explaining the ex-

tensions done in the PICT algorithm. The extensions done are important to achieve the expected results in the test discovery algorithm presented in Section 4.3.

4.2.2 Preparation

In the preparation phase, the combinatorial algorithm receives one Combinatorial Model (CM) and computes all of the information that is necessary for the generation phase, that is the set of all parameter interactions to be covered, designated by P. Each combination of values to be covered is reflected in one slot contained in the set of parameter interactions (P).

For example, receiving the Combinatorial Model (CM) containing three factors: *A*, *B* and *C*. The factor *A* with the possible values $\{0, 1\}$, the factor *B* with the possible values $\{0, 1\}$ and *C* with the possible values $\{0, 1, 2\}$. The Combinatorial Model (CM) , with ($T = 2$) is represented by:

$$
\begin{aligned}
CM = \quad & \{factors, T, \emptyset, \emptyset, \emptyset\} \\
factors = \{ \quad & \\
& A \mapsto \{0, 1\} \ , \\
& B \mapsto \{0, 1\} \ , \\
& C \mapsto \{0, 1, 2\} \\
\} \quad & \\
T = \quad & 2
\end{aligned}
$$

The preparation phase creates the set of parameter interactions (P) to be covered, represented in Figure 4.1. With the Combinatorial Model (CM) previously described, one set of parameter interactions (P) is created containing three parameter interactions: *AB*, *AC* and *BC*. Each of these has a set of slots that correspond to possible value combinations that participate in a particular factor interaction (four slots for *AB* each and six slots for *AC* and *BC*). In this example, the slots contained in the parameter interaction *AB* are the set of slots $AB = 00, 01, 10, 11$.

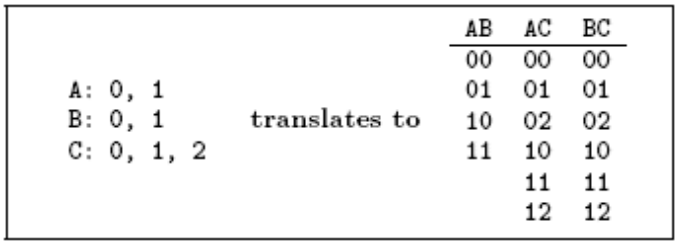

Figure 4.1: Parameter Interaction Structure in PICT [Cze]

The transformation from the CM received to the set of parameter interaction P is the same that is described in PICT algorithm [Cze]. This transformation has not been modified from the original PICT algorithm.

In this phase, the PICT algorithm assigns each slot to *UNCOVERED* or *EXCLUDED*. In the generation phase, the *UNCOVERED* slots will be converted to *COVERED*.

In this phase, the proposed combinatorial algorithm assigns each slot to *UNCOVERED*, *EXCLUDED* or *UNAVAILABLE*. In the generation phase, the *UNCOVERED* and *UNAVAILABLE* slots will be converted to *COVERED*.

If any constraints were defined in a model, they are converted into a set of exclusions (combination between values that can not appear in the set of generated combinations R). Corresponding slots are then marked *EXCLUDED* in the of parameter interaction P and, therefore, removed from the combinations to be covered. PICT set the slots the remaining slots to *UNCOVERED* in this phase.

In the proposed combinatorial algorithm in this dissertation, the slots can also be marked as *UNAVAILABLE* meaning that the remaining slots can be marked as *UNCOVERED* or *UNAVAILABLE*. If the slot is part of one combinations in the set $availableCombinations$, continues to be marked as *UNCOVERED*, otherwise, the slot is marked as *UNAVAILABLE*.

The slot becomes *COVERED* when the algorithm produces a complete combination in generation phase that satisfies that particular slot, adding the produced combination to the set of generated combinations R.

PICT algorithm ends when there are no *UNCOVERED* slots. The proposed combinatorial algorithm terminates when there are no *UNCOVERED* or *UNAVAILABLE* slots. With the parameter interactions structure created, the next phase is the generation of the minimum number of combinations that fulfil the CM, covering incrementally the slots existent in the parameter interaction structure P.

4.2.3 Generation

The generation phase works with a greedy heuristic that builds one combination of the set of generated combinations R at a time, locally optimizing the solution in one deterministic way.

PICT algorithm incrementally creates combinations that adds to the set of generated combinations R, working with a greedy heuristic that is represented in the Figure 4.2, retrieved from the PICT [Cze] algorithm.

```
# Assume test cases r_1,...,r_{i-1} are already produced
# Slots in P reflecting combinations selected by r_1,...,r_{i-1} are set to covered

If there are any unused seed combinations not violating any exclusions
    Add a seed combination to r_i
    Mark all slots in P covered by the seed combination as covered

While there are parameters with no values in r_i

    If r_i is empty
        Choose a parameter interaction p from P with most uncovered slots
        Pick the first uncovered combination from p
    Else
        # Assume values l_1,...,l_{k-1} have already been chosen and added to r_i

        Look at subset Q of P that covers at least one parameter with no
         representation in l_1,...,l_{k-1}

        Look at slots in Q which values are consistent with already chosen values in l_1,...,l_{k-1}

        If there exist uncovered combinations
            Pick a slot with values which when added to r_i would cover the most uncovered
             combinations with l_1,...,l_{k-1} and the resulting partial test case r_i would not
             contain an excluded combination
        Else
            Pick randomly a covered combination which when added to l_1,...,l_{k-1} would not
             contain an excluded combination

    Add values of this combination to r_i
    Mark the chosen combination in P as covered
```

Figure 4.2: PICT [Cze] heuristic alghorithm

The proposed combinatorial algorithm in this dissertation, will also create combinations incrementally in one deterministic way. The heuristic used to create the combinations will be explained during this Sub Section.

PICT algorithm gives preference to the *UNCOVERED* slots in the parameter interaction structure when it is selecting the next slot to change to *COVERED*. The implemented combinatorial algorithm gives preference to the *UNAVAILABLE* slots and only if there is no *UNAVAILABLE* slots to choose, it evaluates the *UNCOVERED* slots.

The reason for prioritizing the *UNAVAILABLE* slots over the *UNCOVERED* slots is to minimize the number of combinations generated by the combinatorial algorithm that are not contained in the $availableCombinations$ set.

The generation phase does not assume anything about the combinations to be covered. It operates on a set of parameter interaction (P) that is produced in the preparation phase. The algorithm produces one set of combinations R that fulfils the set of parameter interaction P generated in the preparation phase:

$$
\begin{aligned}
R = \{ & \\
& c_1, c_2, ..., c_m \\
\} & \\
c_i = \{ & \\
& (f_1 = v_{1,k}), ..., (f_n = v_{n,l}) \\
\} &
\end{aligned}
$$

The proposed heuristic algorithm (represented in algorithm 1) is one greedy algorithm optimizing the solution incrementally, expanding the combinations one at a time. While the combination that is expanding do not have all the factors with one value assign, the algorithm continues to expanding the combination (r_i).

Algorithm 1 Generation algorithm

```
1: procedure Generation algorithm(P, AC)
2:     R ← ∅
3:     N ← NumberOfFactors(P)
4:     # While P has unavailable or uncovered slots
5:     while unavailable ∈ P OR uncovered ∈ P do
6:         r_i = ∅
7:         Available ← TRUE
8:         # while there are factors with no values in r_i
9:         while |r_i| ≠ N do
10:            if r_i = ∅ then
11:                slot ← SelectInitialSlotToCover(P)
12:                Available ← IsSlotAvailable(P)
13:            else
14:                slot ← SelectSlotToExpandCombination(P, r_i, AC, Available)
15:            end if
16:            r_i ← AddSlotToPartialResult(slot, r_i)
17:            P ← MarkSlotsAsCovered(slot)
18:        end while
19:        R ← AddCombinationToResult(r_i, R)
20:    end while
21: end procedure
```

If the combination that is being generated (r_i) is empty (there is not any value assigned to any factor), the algorithm starts by selecting one initial slot (explained in the algorithm 2) and evaluate if the slot exists in the $availableCombinations$ or not.

Algorithm 2 Select Initial Slot To Cover algorithm

```
1: procedure Select Initial Slot To Cover algorithm(P)
2:     # Choose a parameter interaction q from P with most unavailable or uncovered slots
3:     Q ← SelectParameterInteractionWithMoreUnavailableorUncoveredSlots(P)
4:     if unavailableinQ then
5:         # Choose a unavailable slot from Q
6:         slot ← SelectUnavailableSlot(Q)
7:     else
8:         # Choose a uncovered slot from Q
9:         slot ← SelectUncoveredSlot(Q, AC)
10:    end if
11:    return slot
12: end procedure
```

If the slot do not exist in any the set $availableCombinations$, the algorithm marks the current combination that is generating (r_i) with $Available = False$, allowing the following iterations of the algorithm to know that is expanding one combination that is not available.

For each slot that is selecting, the algorithm prioritizes the *UNAVAILABLE* slots. Only when there are not *UNAVAILABLE* slots to choose, the algorithm evaluates the *UNCOVERED* slots.

This prioritization enables to minimize the number of combinations generated by the combinatorial algorithm that are not contained in $availableCombinations$, covering first the *UNAVAILABLE* slots and only after the *UNCOVERED* slots.

The slot selected to be covered is then added to the partial result (r_i), that is the combination being incrementally build at each iteration of the heuristic algorithm.

After the first iteration, the partial result (r_i) will contain the values $v_1, ..., v_{k1}$ that exist in the selected slot.

The algorithm to select the following slots to add to the partial result is very similar, being described in the algorithm 3.

Algorithm 3 Add Slot To Partial Result

```
1: procedure Add Slot To Partial Result(P, r_i, AC, Available)
2:     # Assume values v_1, ..., v_k1 have already been chosen and added to r_i
3:     # Choose a parameter interaction q from P with most unavailable or uncovered slots
4:     Q ← SelectParameterInteractionWithMoreUnavailableorUncoveredSlots(P)
5:     Q ← RemoveInconsistentSlotsWithPartialResult(Q, PR)
6:     if unavailableinQ then
7:         # Pick an unavailable slot with values which that would cover the most unavailable combinations
8:         slot ← SelectUnavailableSlot(Q, AC)
9:     else if uncoveredinQ then
10:        if Available = False then
11:            # Pick a uncovered slot with values which when added to PR would cover the most uncovered or unavailable combinations
12:            slot ← SelectUncoveredOrCoveredSlot(Q)
13:        else
14:            # Pick a uncovered or covered slot with values which when added to PR would cover the most uncovered slots and values v_1, ..., v_k1 of PR are contained in the availableCombinations
15:            slot ← SelectUncoveredOrCoveredSlot(Q, PR, AC)
16:        end if
17:    else
18:        # All slots of Q are already covered
19:        if Available = False then
20:            # Pick any slot of Q
21:            slot ← SelectCoveredSlot(Q)
22:        else
23:            # Pick a covered slot with values v_1, ..., v_k1 which when added to PR are contained in the availableCombinations
24:            slot ← SelectCoveredSlot(Q, PR, AC)
25:        end if
26:    end if
27:    return slot
28: end procedure
```

For each slot selection, the algorithm prioritizes the ***UNAVAILABLE*** slots and only if there is no ***UNAVAILABLE*** slots to choose, it evaluates the ***UNCOVERED*** slots. If there is no more ***UNCOVERED*** slots, the algorithm evaluates the ***COVERED*** slots.

If the algorithm is expanding one combination with $Available = TRUE$ then it is not possible to have ***UNAVAILABLE*** slots, because they have been selected in the Select Initial Slot to Cover (algorithm 2).

Expanding one combination that is contained in the $availableCombinations$ set, identified by having the flag $Available = TRUE$, the algorithm evaluates each possible slot to expand.

The algorithm selects one slot where the partial result obtained, with the slot being evaluated incremented to the partial result, is also contained in the $availableCombinations$.

The slot selected to be covered is then added to the partial result (r_i), that is the combination being incrementally build at each iteration of the heuristic algorithm.

After this expansion, the partial result (r_i) will contain the values $v_1, ..., v_{k1}$ that exist in the selected slot and all the values that existed previously in the partial result (r_i).

This proposed algorithm enables to minimize the number of combinations generated by the combinatorial algorithm that are not contained in $availableCombinations$, respecting all the other requirements defined in the Combinatorial Model (CM).

4.3 Test Discovery Algorithm

The test discovery algorithm is the main work of this dissertation. The algorithm that selects the Test Environment (TE)s where each test will execute.

This selection is of extreme importance since each Test Specification (TS) has different Relevant Factors (RF) descriptions, meaning that:

1. not all Test Environment can execute all the Test Specification, because there are TS that require values that may not be available in the TE;

2. running the same test in multiple TE can lead to unnecessary wasted time in test execution without adding value to the test results, if these multiple executions never reveal different faults.

When designing the proposed technique, the major concerns were that it must be scalable as the configurations space grows with time and must be applicable to any number of heterogeneous Test Environment.

In order to address the previous challenges, the proposed algorithm on this dissertation distributes the work between the heterogeneous TE. The decision if a test case is to execute or not, will be taken in each TE. The distributed nature of this solution, enables to scale the proposed technique to any number of TE.

This algorithm requires three input parameters: Test Specification (TS), Test Environment (TE) List and the System Model (SM).

The Test Specification (TS) is one pair which contains: actions and Relevant Factors (RF).

$$TS = \quad (actions, RF)$$

The actions are the steps that the system should execute, validating the expected behavior. The Relevant Factors (RF) is one model, composed by a set of required values to be exercised for each factor, the number of factors interactions (T) and sub models definition enabling to assign different importance between factors interaction:

$$
\begin{aligned}
RF = \quad & \{factors, T, subModels\} \\
factors = \{ \quad & \\
& f_1 \mapsto \{v_{1,1}, v_{1,2}, ..., v_{1,m}\} \ , \\
& f_2 \mapsto \{v_{2,1}, v_{2,2}, ..., v_{2,o}\} \ , \\
& \dots \ , \\
& f_n \mapsto \{v_{n,1}, v_{n,2}, ..., v_{n,p}\} \\
\} & \\
T = \quad & N \\
subModels = \{ \quad & \\
& \{T_1, \{f_1, f_2, ..., f_m\}\} \ , \\
& \dots \ , \\
& \{T_n, \{f_1, f_2, ..., f_m\}\} \\
\} &
\end{aligned}
$$

For example, given one TS that need to be exercised against variations in three different configurable factors: platform stack, application server and database engine. Following is the representation of the values that need to be exercised for each of the previous factors in the example Test Specification (TS).

$$
\begin{aligned}
factors = \{ \quad & \\
& PlatformStack \mapsto \{.Net, Java\} \ , \\
& ApplicationServer \mapsto \{IIS, JBoss, WebLogic, WildFly\} \ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} &
\end{aligned}
$$

The number of parameters interactions *(T)* is also specified in the relevant factors. This element is one integer in the interval $1 \leq T \leq N$, being N the number of factors in the Relevant Factors (RF) description. In the previous example, $T \leq 3$, because the total number of factors N is 3. This element allows to define the strength of the generated test suite, meaning that lower T generates less test cases and higher T will generate more test cases, giving more confidence but requiring more variations in the Test Environments to execute.

At last, the Relevant Factors (RF) allows to define sub-models. This allows certain factors to be t-wised combined first, and then that product is used for creating combinations with factors on the upper level of the hierarchy. This is a useful technique that enables to limit the combinatorial explosion of certain parameter interactions, prioritizing the interactions more relevant to the Test Specification (TS).

Using the previous example, we can define the Relevant Factors (RF) with $T = 1$ meaning that is required to generate combinations with each value at least one. If it is described one Sub Model containing the factors $\{PlatformStack, Database\}$ with $T = 2$, then the generated combinations need to have the values of the factor $ApplicationServer$ at least once and all pairs $(T = 2)$ of the values contained in the factors $\{PlatformStack, Database\}$. The $subModel$ is represented bellow:

$$
\begin{aligned}
subModels = \{ \quad & \\
& (T = 2, \{PlatformStack, Database\}) \\
\} &
\end{aligned}
$$

This description of the relevant factors enables to have flexibility to generate the test cases, allowing to create models that can fit all the type of test specifications. Following is the complete representation of the Relevant Factors (RF) example previously described:

$$
\begin{aligned}
RF = \quad & \{factors, T, subModels\} \\
factors = \{ & \\
& PlatformStack \mapsto \{.Net, Java\} \ , \\
& ApplicationServer \mapsto \{IIS, JBoss, WebLogic, WildFly\} \ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} & \\
T = \quad & 1 \\
subModels = \{ & \\
& (T = 2, \{PlatformStack, Database\}) \\
\} &
\end{aligned}
$$

The second parameter required is the Test Environment list (designated by TEL), that is a set containing the available values for each factor in each test environment. The TEL is represented by:

$$
\begin{aligned}
TEL = & \{ \\
& \{TE_1, TE_2, ..., TE_n\} \\
& \} \\
TE_i = & \{ \\
& f_1 \mapsto \{v_{1,1}, v_{1,2}, ..., v_{1,m}\} \ , \\
& f_2 \mapsto \{v_{2,1}, v_{2,2}, ..., v_{2,o}\} \ , \\
& \dots \ , \\
& f_n \mapsto \{v_{n,1}, v_{n,2}, ..., v_{n,p}\} \\
& \}
\end{aligned}
$$

Following is the representation of two test environments (TE_1 and TE_2), with the values available for each of the previous factors.

$$
\begin{aligned}
TEL = & \{ \\
& \{TE_1, TE_2\} \\
& \} \\
TE_1 = & \{ \\
& PlatformStack \mapsto \{.Net\} \ , \\
& ApplicationServer \mapsto \{IIS\} \ , \\
& Database \mapsto \{SqlServer\} \\
& \} \\
TE_2 = & \{ \\
& PlatformStack \mapsto \{Java\} \ , \\
& ApplicationServer \mapsto \{WildFly\} \ , \\
& Database \mapsto \{MySQL\} \\
& \}
\end{aligned}
$$

At last, it is required one structure to validate the factors and respective values described in the Test Specification (TS) and Test Environment (TE). This structure, named System Model (SM), contains all the possible factors with the respective values and the constraints between the different values in the form of propositional constraints:

$$
\begin{aligned}
SM = \quad & \{factors, constraints\} \\
factors = \{ & \\
& f_1 \mapsto \{v_{1,1}, v_{1,2}, ..., v_{1,m}\} \ , \\
& f_2 \mapsto \{v_{2,1}, v_{2,2}, ..., v_{2,o}\} \ , \\
& \dots \ , \\
& f_n \mapsto \{v_{n,1}, v_{n,2}, ..., v_{n,p}\} \\
\} & \\
constraints = \{ & \\
& if(f_1 = v_{11}) then(f_2 = v_{21}), \\
& if(f_2 = v_{22}) then(f_3 \in v_{31}, v_{32}), \\
& \dots \\
& if(f_n = v_{ni}) then(f_m = v_{mj}) \\
\} &
\end{aligned}
$$

Constraints describe limitations to the domain that need to be exercised, allowing to exclude unwanted combinations from the obtained results. The constraints are described in the SM because they are global and they do not change for each Test Specification. In our proposed approach we are using constraints in the form of propositional forms, using **IF-THEN** statements to describe the limitations of a particular test domain.

Using the previous example, the application server supported in the OutSystems Platform are Microsoft Internet Information Services for the .Net stack and JBoss, WebLogic or WildFly for the Java stack. Another condition is also that the SqlServer database is only available in the .Net stack. Following is the representation of the System Model (SM) described in this example:

$$
\begin{aligned}
SM = \quad & \{factors, constraints\} \\
factors = \{ & \\
& PlatformStack \mapsto \{.Net, Java\} \ , \\
& ApplicationServer \mapsto \{IIS, JBoss, WebLogic, WildFly\} \ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} & \\
constraints = \{ & \\
& if(PlatformStack = .Net) then(ApplicationServer = IIS) \ , \\
& if(PlatformStack = Java) then(ApplicationServerin(JBoss, WebLogic, WildFly)) \ , \\
& if(PlatformStack = .Net) then(Database = SqlServer) \\
\} &
\end{aligned}
$$

With all the required information, the test discovery algorithm is able to select the Test Environment (TE) where each Test Specification (TS) should execute. The goal of this algorithm is to generate the required Test Cases that need to be executed to fulfill the Relevant Factors description. Algorithm 4 presents the test discovery algorithm.

Algorithm 4 Test Discovery

```
1: procedure Test Discovery(TS, TEL, SM)
2:     TCL ← ∅
3:     RF ← ExtractRelevantFactors(TS, SM)
4:     AC ← CalculateAvailableCombinations(RF, TEL)
5:     RCL ← CalculateRequiredCombinations(RF, AC)
6:     for all RC_i ∈ RCL do
7:         TE ← SelectBestEnvironment(RC_i, TEL)
8:         TC ← CreateTestCase(RC_i, TE)
9:         TCL ← Add(TC, TCL)
10:    end for
11: return TCL
12: end procedure
```

The test discovery starts by extracting the Relevant Factors (RF) from the Test Specification, validating if each of the values is present in the System Model.

Besides having the model of the Relevant Factors (RF) created, it is also required to have the Available Combinations (AC). This information is calculated in the $CalculateAvailableCombinations$ method, based on the information described in the list of Test Environments (defined by TEL). The method evaluates the values that are available on each TE and calculates the Available Combinations in the set of Test Environments (defined by TEL). This step is explained in the algorithm 5.

Algorithm 5 Calculate Available Combinations

```
1: procedure CalculateAvailableCombinations(RF, TEL)
2:     AC ← ∅
3:     for all  dote_i ∈ TEL
4:         AV ← FilterValuesThatExistInRF(RF, te_i)
5:         # the TE do not have all the factors available
6:         if |AV| ≠ N then
7:             continue
8:         end if
9:         # create one CM with the values available on the te_i and T = N
10:        CM_i ← CreateCombinatorialModel(AV, N, ∅, ∅, ∅)
11:        # calculate the available combinations in the Test Environment te_i
12:        ac_i ← GenerateCombinations(CM_i)
13:        AC ← Add(ac_i)
14:    end for
15:    return AC
16: end procedure
```

The method $CalculateAvailableCombinations$ iterates over each Test Environment contained in the TEL. For each te_i in TEL, filter the values that are available in the te_i and are contained in the Relevant Factors (RF) description. If te_i do not contain all the $factors$ required by the Relevant Factors (RF) description, the method do not continue to evaluate te_i and analyses the following te_{i+1}.

If te_i has all the $factors$ required by the Relevant Factors (RF) description, the following step is to create one Combinatorial Model (CM_i) with the $factors$ and respective $values$ available in te_i and with $T = N$, being N the total number of $factors$.

Executing the $GenerateCombinations(CM_i)$ with this CM, returns all the Available Combinations (represented by ac_i) from the Test Environment te_i. The method $GenerateCombinations(CM_i)$ uses the algorithm explained in the Section 4.2.

At last, the Available Combinations (ac_i) from the Test Environment te_i, are added to the Available Combinations (AC) from all TEs (TEL). For example, with two Test Environment (TE_A and TE_B) in the set TEL, with the following values available for each of factors in each TE:

$$
\begin{aligned}
TEL = & \{ \\
& \{TE_A, TE_B\} \\
& \} \\
TE_A = & \{ \\
& PlatformStack \mapsto \{.Net\}\ , \\
& ApplicationServer \mapsto \{IIS\}\ , \\
& Database \mapsto \{SqlServer\} \\
& \} \\
TE_B = & \{ \\
& PlatformStack \mapsto \{Java\}\ , \\
& ApplicationServer \mapsto \{WildFly\}\ , \\
& Database \mapsto \{MySQL\} \\
& \}
\end{aligned}
$$

The Available Combinations returned by the $CalculateAvailableCombinations$ method are the following:

$$
\begin{aligned}
AC = & \{ \\
& \{ac_1, ac_2\} \\
& \} \\
ac_1 = & \{ \\
& (PlatformStack = .Net), (ApplicationServer = IIS), (Database = SqlServer) \\
& \} \\
ac_2 = & \{ \\
& (PlatformStack = Java), (ApplicationServer = WildFly), (Database = MySql) \\
& \}
\end{aligned}
$$

With the Available Combinations (AC) calculated, the next step in the test discovery algorithm (presented in algorithm 4) is to calculate the Required Combinations (RC) to execute.

In this step, it is generated the minimum number of combinations that are required to execute fulfilling the requirements described in the Relevant Factors (RF). This selection is done in the method $CalculateRequiredCombinations$, presented in the algorithm 4.

With the information of the Available Combinations (AC) for the Test Environment (TE), this method can give preference to generate combinations that are available in the TEL. The algorithm inside this method is the algorithm 6.

Algorithm 6 Calculate Required Combinations

1: **procedure** Calculate Required Combinations(RF, AC, SM)
2: $factors \leftarrow GetFactorsFromRelevantFactors(RF)$
3: $T \leftarrow GetTFromRelevantFactors(RF)$
4: $subModels \leftarrow GetSubModelsFromRelevantFactors(RF)$
5: $constraints \leftarrow GetConstraintsFromSystemModel(SM)$
6: # calculate the CM that requires the minimum set of combinations that fulfil RF
7: $CM \leftarrow CreateCombinatorialModel(factors, T, subModels, constraints, AC)$
8: # returns the required combinations to execute
9: $RC \leftarrow GenerateCombinations(CM)$
10: **return** RC
11: **end procedure**

The goal of the method $CalculateRequiredCombinations$ is to generate the minimum set of combinations that are required to execute, fulfilling the requirements described in the Relevant Factors (RF).

The first step is to extract the required information to create the Combinatorial Model. The $factors$, T and $subModels$ are extracted from the Relevant Factors because they can be different for each Test Specification. The $constraints$ are extracted from the System Model because they apply to all Test Specification.

With the Available Combinations (AC) received from the previous step (algorithm 5), one Combinatorial Model is created with all the previously gathered information:

$$
\begin{aligned}
CM = \quad & \{factors, T, subModels, constraints, AC\} \\
factors = \{ \quad & \\
& PlatformStack \mapsto \{.Net, Java\} \ , \\
& ApplicationServer \mapsto \{IIS, JBoss, WebLogic, WildFly\} \ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} \quad & \\
T = \quad & 1 \\
subModels = \{ \quad & \\
& (T = 2, \{PlatformStack, Database\}) \\
\} \quad & \\
constraints = \{ \quad & \\
& if(PlatformStack = .Net) then(ApplicationServer = IIS) \ , \\
& if(PlatformStack = Java) then(ApplicationServer in(JBoss, WebLogic, WildFly)) \ , \\
& if(PlatformStack = .Net) then(Database = SqlServer) \\
\} \quad & \\
AC = \{ \quad & \\
& \{(PlatformStack = .Net), (ApplicationServer = IIS), (Database = SqlServer)\} \ , \\
& \{(PlatformStack = Java), (ApplicationServer = WildFly), (Database = MySql)\} \\
\} \quad &
\end{aligned}
$$

Following, the method $GenerateCombinations(CM)$ is executed to calculate the minimum set of combinations that fulfil the Combinatorial Model previously created (CM), giving preference to generate combinations that are available in Available Combinations (AC).

The method $GenerateCombinations(CM)$ uses the algorithm 1, explained in the Section 4.2. The result obtained from the $GenerateCombinations(CM)$ method is the following:

$$
\begin{aligned}
RC = & \{ \\
& \{rc_1, rc_2, rc_3, rc_4, rc_5, rc_6\} \\
& \} \\
rc_1 = & \{ \\
& (PlatformStack = .Net), (ApplicationServer = IIS), (Database = SqlServer) \\
& \} \\
rc_2 = & \{ \\
& (PlatformStack = .Net), (ApplicationServer = IIS), (Database = Oracle) \\
& \} \\
rc_3 = & \{ \\
& (PlatformStack = .Net), (ApplicationServer = IIS), (Database = MySql) \\
& \} \\
rc_4 = & \{ \\
& (PlatformStack = Java), (ApplicationServer = JBoss), (Database = Oracle) \\
& \} \\
rc_5 = & \{ \\
& (PlatformStack = Java), (ApplicationServer = WebLogic), (Database = Oracle) \\
& \} \\
rc_6 = & \{ \\
& (PlatformStack = Java), (ApplicationServer = WildFly), (Database = MySql) \\
& \}
\end{aligned}
$$

With the Required Combinations (RC) generated, it is required to evaluate each combination and find what is the best TE that should execute each rc_i (method $SelectBestEnvironment$, executed in the algorithm 7). The algorithm inside the method $SelectBestEnvironment$ is the algorithm 6.

Algorithm 7 Select Best Environment

```
1: procedure Select Best Environment(rc_i, TEL)
2:     for all  doTE_i ∈ TEL
3:         # check if all the values of RC are available in the Test Environment TE_i
4:         if rc_i ∈ TE_i then
5:             return TE_i
6:         end if
7:     end for
8:     # return empty if there is not any Test Environment that has all the values of one combination
9:     return ∅
10: end procedure
```

One Test Environment (TE) is able to execute the combination (rc_i) if contains all the values in the combination rc_i. If there is not any Test Environment (TE) available with all the values in the combination rc_i, then the result is $\emptyset$. If more than one TE has available all the values in the required combination rc_i, only one is selected, being selected the first Test Environment (TE) that fulfil all the values.

For each generated Required Combinations (RC), the test discovery algorithm creates one new Test Case (TC) that uses the Required Combinations (RC) as requirement to select the Test Environment (TE) where the Test Case (TC) is selected to execute.

With the TE selected to execute rc_i, the next step (method $CreateTestCase$ in the algorithm

4) is to generate the Test Case (TC) for the previous required combination rc_i and add it to the Test Case List (TCL), that will be returned from the test discovery algorithm. The algorithm inside the $CreateTestCase$ method is represented in algorithm 8.

Algorithm 8 Test case constructor

```
1: procedure Create Test Case(TS, rc_i, TE)
2:     if TE = CurrentTestEnvironment then
3:         discoveryStatus ← Enabled
4:     else if TE = ∅ then
5:         discoveryStatus ← Missing
6:     else
7:         discoveryStatus ← Disabled
8:     end if
9:     actions ← ExtractActionsFromTestSpecification(TS)
10:    TC ← CreateTestCase(actions, rc_i, discoveryStatus) return TC
11: end procedure
```

The $CreateTestCase$ method creates the Test Case (TC) containing the Required Combinations (RC) as requirement that the Test Environment (TE) must fulfil to be able to execute the Test Case (TC). The test case is created with one $discoveryStatus$ attribute, that may have one of the following values assigned:

- $Enabled$ - if the TE selected to execute the Test Case is the same Test Environment where the test discovery algorithm is executing;
- $Missing$ - if there is not any TE with all the values contained in the combination rc_i, enabling to identify that there are not Test Environments to fulfil rhe required variations described in the Relevant Factors (RF);
- $Disabled$- if the Test Case is selected to execute in another Test Environment, meaning that the TE selected to execute the Test Case is not the same Test Environment where the test discovery algorithm is executing.

With the $discoveryStatus$ assigned, the method creates the Test Case with all the required information to execute:

- $actions$ - the $actions$ that the Test Case (TC) should execute to validate the expected behavior of the System Under Test;
- rc_i - the variation of the Relevant Factors described in the TS, used as requirement to select the Test Environment (TE) ;
- $discoveryStatus$ - enabling to know if the Test Case (TC) is $Enabled$, $Disabled$ or $Missing$ in the Test Environment (TE) where the test discovery algorithm is executing.

At the end of the test discovery algorithm, Test Case List (TCL) is produced, that contains the multiple TC discovered in the TE where the algorithm is executing.

The Test Case List (TCL) is different depending on the TE where the test discovery algorithm is executing. Following are the results obtained when the algorithm is executing in two different TE (TE_A produces the TCL_A and TE_B produces the TCL_B).

$$
\begin{array}{rl}
TCL_A = & \{tc_1, tc_2, tc_3, tc_1, tc_2, tc_3\} \\
tc_1, = \{ & \\
 & actions, rc_1, Enabled \\
\} & \\
tc_2 = \{ & \\
 & actions, rc_2, Missing \\
\} & \\
tc_3 = \{ & \\
 & actions, rc_3, Missing \\
\} & \\
tc_4, = \{ & \\
 & actions, rc_4, Missing \\
\} & \\
tc_5 = \{ & \\
 & actions, rc_5, Missing \\
\} & \\
tc_6 = \{ & \\
 & actions, rc_6, Disabled \\
\} &
\end{array}
\qquad
\begin{array}{rl}
TCL_B = & \{tc_1, tc_2, tc_3, tc_1, tc_2, tc_3\} \\
tc_1, = \{ & \\
 & actions, rc_1, Disabled \\
\} & \\
tc_2 = \{ & \\
 & actions, rc_2, Missing \\
\} & \\
tc_3 = \{ & \\
 & actions, rc_3, Missing \\
\} & \\
tc_4, = \{ & \\
 & actions, rc_4, Missing \\
\} & \\
tc_5 = \{ & \\
 & actions, rc_5, Missing \\
\} & \\
tc_6 = \{ & \\
 & actions, rc_6, Enabled \\
\} &
\end{array}
$$

The test discovery algorithm returns the same number of TC in the TCL. Each tc_i contains the same $actions$ that the test should execute to validate the expected behavior of the System Under Test and the same rc_i representing the variation that the Test Case (TC) is exercising.

The difference between the TCL generated in the two Test Environments is the $statusDiscovery$ of each Test Case tc_i. The $statusDiscovery$ allows to determine if the Test Case (TC) is selected to be executed in that TE ($statusDiscovery = Enabled$), is selected to execute in another Test Environment ($statusDiscovery = Disabled$) or there is no Test Environment ($statusDiscovery = Missing$) with all the values to execute the TC.

4.4 Conclusion

This chapter presented a solution to the problem of the automated Test Case minimization and distribution.

Section 4.3 presented the core algorithm for test discovery and selection that will run in each Test Environment (TE), while Section 4.2 presented the algorithm needed by the discovery algorithm for determining the relevant combinations of environmental factors.

The main goal of the Proposed Solution is to only execute test in more than one environment if strictly necessary to fulfill the Relevant Factors on the Test Specification.

All the stages of this proposed technique were presented in a way that they may fit different industrial contexts, independently of the Test Specification language or the Test Environment infrastructure. The technique only requires a way to recover the Relevant Factors description from the Test Specification and the available values from the test environments.

Regarding the related work that exists in both academic and industrial fields, described in Chapter 3, the present technique brings novelty because it is extending the capabilities of the Combinatorial Interaction Testing techniques, transforming them from one test case minimization technique to one test case minimization and distribution technique.

Chapter 5 will describe a specific implementation of this test selection technique, which will describe more concisely how can this technique be used in practice.

Chapter 5

Solution Implementation

This chapter presents the implementation details of the solution presented in the Chapter 4. The main goal of this implementation work was to demonstrate the applicability of the proposed technique in practice adapted to the industrial context where this work was developed, the OutSystems industrial context.

5.1 Introduction

In the present chapter, the toolset and programming languages used for this implementation are described, followed by the reasons why the different tools were chosen. Furthermore, a brief explanation is presented on the test infrastructure available at OutStystems and how it was adapted to support the proposed solution.

This implementation was designed to the needs of the industrial context of OutSystems. But because the proposed solution in Chapter 4 is generic, it can be easily adapted to any other industrial context

5.2 Programming Languages

For the implementation of the presented technique several programming languages were used. Before describing the implementation details of the proposed solution, such languages are first described.

The first programming language used was C# language, an object oriented programming language and was mainly used to implement the test discovery algorithm presented in Section 4.3.

The second programming language was C++, that is also an object oriented programming language and was the main language used to modify existing version of PICT, enabling to extend it, implementing the combinatorial algorithm describe in 4.2. The motives to implement one combinatorial algorithm and not use PICT are presented further, at Section 6.2

Some scripting languages were also used such as Batch [Ric95] and XSLT [Cla99], those languages were specially used to integrate the proposed solution in the automating testing system at OutSystems.

5.3 Toolset

In the proposed solution described in Chapter 4, two main algorithms were presented:

1. **combinatorial algorithm** - responsible to generate the minimum set of combinations that fulfil one Combinatorial Model;

2. **test discovery algorithm** - responsible to select the Test Environments where each Test Specification will be executed, using the combinatorial algorithm.

This section will describe the choices for those two implementation aspects, as well as justifying that same choices.

5.3.1 Test Discovery Algorithm

The test discovery algorithm was implement in NUnit. NUnit is a unit-testing framework designed to work with all .NET languages. This unit-testing framework is entirely written in C# language and is available as open-source software.

Nunit provides a easy way to extend its internal behavior, through the NUnit Addins API. The NUnit Addins API is an extension mechanism that allows one to introduce new, or change existing behaviour without the need to modify NUnit itself. Using this extension mechanism, one new type of test specification was implemented, that will be described in the section 5.5.1.

In the context of this dissertation, NUnit was used to implement the test discovery algorithm proposed and execute the test cases. Also it is relevant to mention that the Outsystems Platform tests depend on custom OutSystems' extensions to NUnit, which are necessary to integrate the tests with the OutSystems tests automation infrastructure, and also to extend NUnit with the support for new types of test cases need by OutSystems' developers.

The main motivation to choose NUnit to implement the test discovery algorithm is because most of the tests actually existent at OutSystems are NUnit tests, existing also some tests in multiple other testing frameworks (JUnit, Jasmine and others).

5.3.2 Combinatorial Algorithm

The combinatorial algorithm implementation was done in C++, extending the current PICT implementation [Mic15] that is written in C++, being currently one open source project.

The main goal of the combinatorial algorithm is to generate the minimum number of Required Combinations (RC) to fulfil the Relevant Factors (RF).

5.4 OutSystems Quality Assurance Process

The quality assurance process (described in Section 2.4), starts when a developer makes one change in the source code of the platform and commits the change to one control version system. After the commit is performed, the platform is compiled and installed in multiple heterogeneous test environments with different configurations.

The test discovery is part of the installation process in a Test Environment (TE). At the end of the installation process, the test discovery is automatically executed in each Test Environment where the new version of the code was installed. The test discovery is done by calling NUnit to inspect all tests solutions, where all methods annotated with valid test specification attribute (e.g., Test attribute) will be identified as one test specification. The test discovery produces a list of all identified tests that is published into the automation system for later execution.

In the OutSystems test automation infrastructure the same test is typically discovered in all test environments. Developers need to manually disable those tests for any test environment where the test is not intended to run. This is done in the test automation management application called Engineering Dashboard, one internally developed tool responsible to manage the test automation infrastructure. Figure 5.1 presents one web page of the Engineering Dashboard, where it is possible to view the test results from the last test execution on the test environment.

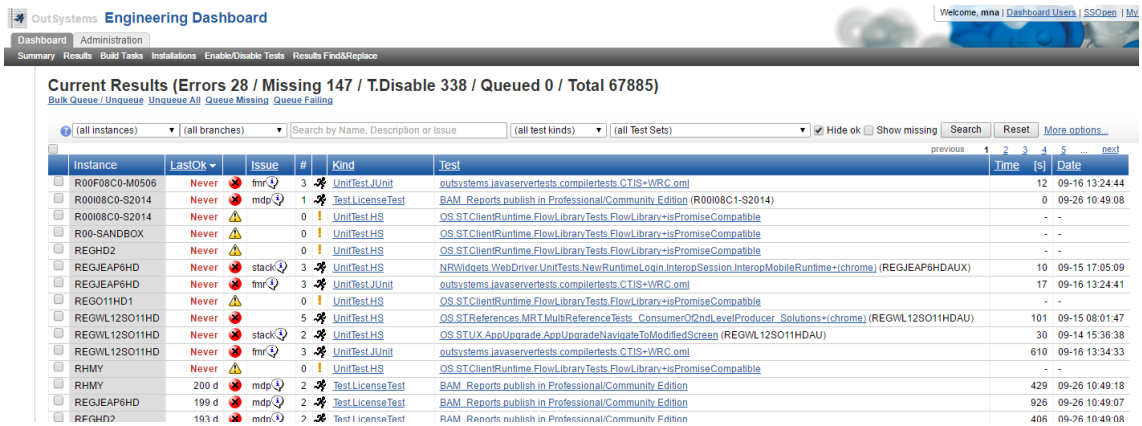

Figure 5.1: Results of test case execution in the Engineering Dashboard

The Engineering Dashboard is one web application that enables developers to perform multiple operations, being the most commonly used the following:

1. request one build for a given branch;
2. install one previously done build to one test environment;
3. manage the current available heterogeneous test environments;
4. execute tests on one test environment;
5. disable or enable tests from one Test Environment.

The proposed test discovery algorithm, presented in the Section 4.3 automatically discovers the Test Case, without any more human effort involved, with the expected status:

- $Enabled$ - if the Test Environment (TE) is the TE selected to execute the Test Case (TC);
- $Missing$ - if there is not any Test Environment (TE) with all the values required by the Test Case (TC);
- $Disabled$- if the Test Case (TC) is selected to execute in another Test Environment (TE).

The implementation described in this chapter will show how it was possible to apply the proposed solution to the OutSystems context without any major change to their test automation system, allowing the elimination of all human effort related to disabling tests in test environments.

The last step of the quality assurance process is to execute the discovered Test Case in each Test Environment where the platform is installed. This is the step where a battery of automated tests that include unit, integration and system tests are executed in order to guarantee the product quality.

All the tests that are ”Enabled” in the Test Environment will be executed and also, all the tests marked as ”Missing” are considered failures by the OutSystems test automation engine. Developers reacts to this as they do with test execution failures, troubleshoot and fixing the problem.

The Engineering Dashboard was the main tool to the quality assurance process in OutSystems when this work started. Currently this is no longer the only tool responsible by the test execution, but that will be further discussed in the future work described in Chapter 7.

5.5 Implementation

This section aims to describe the implementation details of the test discovery algorithm in the OutSystems Context.

As described in Chapter 4, the proposed test discovery algorithm requires three input parameters: Test Specification, the Test Environment List (TEL) and the System Model. Following, it will be presented one description of the implementation of these inputs.

5.5.1 Test Specification

The Test Specifications are written using C#, each Test Specification (TS) tthat requires to be validated in multiple variations of configurations, is implemented as a C# method identified with the a new test attribute *CombinatorialTestCase*.

The *CombinatorialTestCase* is one attribute that extends the TestAttribute existent in NUnit. This presence of this attribute to NUnit over one method, allows to the NUnit discover the method as one Test Specification (TS). Figure 5.2 represents one Test Specification (TS) identified with the *CombinatorialTestCase*.

The actions that the system should execute to validate the expected behavior are inside the method. The Relevant Factors (RF) are statically described inside one object (in this example,

```
[CombinatorialTestCase(typeof(WebDriver))]
public void Test_Specification_Example()
{
    // actions that the system should execute
}
```

Figure 5.2: Example of one Test Specification (TS)

the WebDriver), being referenced only in the TS the object that contains the Relevant Factors (RF) description.

Figure 5.3 presents the description of the Relevant Factors (RF) for one Test Specification that needs to be executed in three different browsers (Chrome, Firefox and IExplorer) against the two different supported platform stacks (.Net and Java) and at least once in the three different supported databases (Oracle, MySQL and SqlServer).

```
public class WebDriver : ICombinatorialIterator {

    public RelevantFactors relevantFactorsDescription()
    {
        RelevantFactors RF = new RelevantFactors(T = 1);

        /*Add the Browser*/
        RF.addModelValue("Browser", "Chrome");
        RF.addModelValue("Browser", "Firefox");
        RF.addModelValue("Browser", "IExplorer");

        /*Add the ServerRuntime*/
        RF.addModelValue("Platform Stack", ".Net");
        RF.addModelValue("Platform Stack", "Java");

        /*Add the server database*/
        RF.addModelValue("Database", "Oracle");
        RF.addModelValue("Database", "MySQL");
        RF.addModelValue("Database", "SqlServer");

        /*Different importance to different factors*/
        SubModel subModel = new SubModel(T = 2);
        subModel.addFactor("Browser");
        subModel.addFactor("Platform Stack");
        RF.subModels.Add(submodel);

        return RF;
    }
}
```

Figure 5.3: Example of relevant factors description

Figure 5.3 represents the description of three factors with the respective values, being required that each value appear at least once ($T = 1$). Additionally to the described factors, the Figure 5.3 shows the description of a sub model requiring all pairs ($T = 2$) for the two factors browser and platform stack.

The decision to isolate the Relevant Factors in one different object has been taken to allow the same Relevant Factors (RF) description to be reused by different Test Specifications.

The developers are able to create new Relevant Factors (RF) descriptions by creating new classes that implement the interface *ICombinatorialIterator*, one interface created during this work and requires the implementation of the method *relevantFactorsDescription()*.

When NUnit is executing the Test Discovery, evaluates the type of Test Specification (TS) and expand it accordingly. Evaluating one Test Specification (TS) identified with the attribute *CombinatorialTestCase*, NUnit will delegate the generation of the test cases that must be executed to the *CombinatorialTestCaseBuilder*. The *CombinatorialTestCaseBuilder* is explained further in the Section 5.6.

5.5.2 Test Environment List

The Test Environment (TE) list is one set containing all the available test environments and which values are available in each test environment. As stated before, OutSystems has an internal application responsible for managing all the test environments, the Engineering Dashboard.

The Engineering Dashboard had already one web page that enabled to described one test environment characteristics, which already accounted for most of the values needed (database, operating system, platform stack and others). The remaining values where added (browser) and additionally, a new web service was implemented in the Engineering Dashboard to get the list of all Test Environments descriptions.

Figure 5.4 shows the existing web page where all the values of each factor appear and the users have to select what are the values available in that test environment.

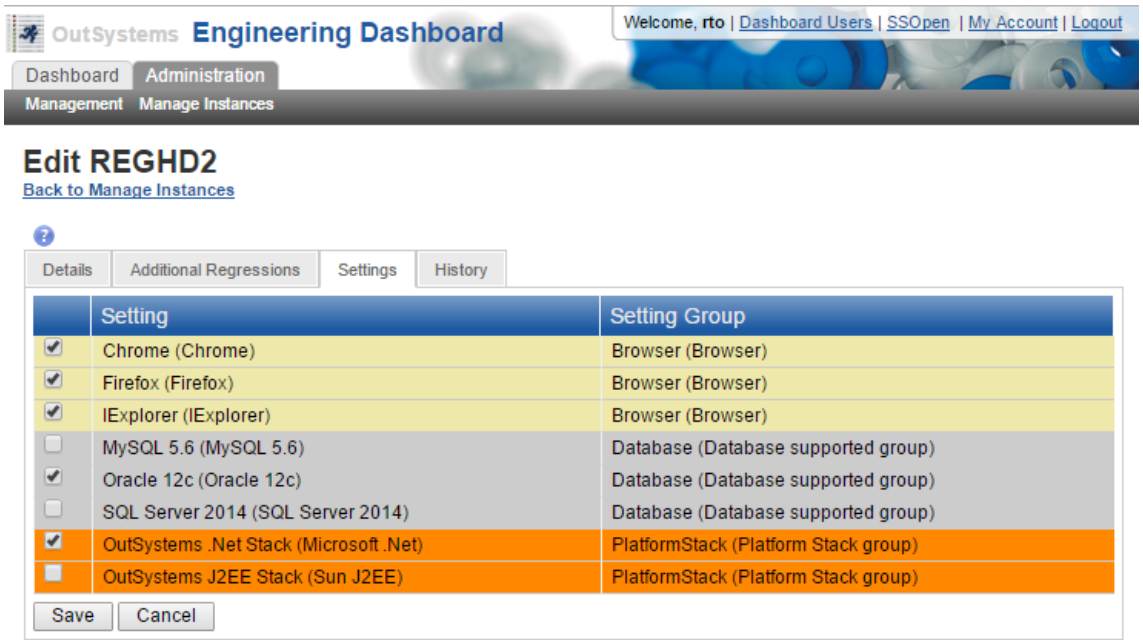

Figure 5.4: Example of available values description

This tool saves the available values in each Test Environment and one new web service was created that provides this information when requested. The web service returns the available values from the test environments that are sharing the same code version.

The existent installation process at OutSystems has been changed to call this web service before starting the test discovery. The result obtained by the web service is stored to one file in one configured location in the Test Environment (TE), being available in each Test Environment to be used by the test discovery algorithm.

5.5.3 System Model

The System Model (SM) aggregates all the known factors and all the possible values for each factor. This structure is required to validate that the factors and respective values referenced by the Test Environment list and the Test Specification are valid. Being this System Model shared between all the test suites, also contains the global constraints.

The System Model may evolve between different versions of the OutSystems Platform. The description of the System Model (SM) is done in one EXtensible Markup Language (XML) file, allowing to have different System Models for different versions of the OutSystems Platform. The System Model (SM) is stored with the code in the same version control system, being able to evolve at the same pace that the code and respective requirements evolve.

The reason to use a file, enabling to have different versions depending of the system under test version, was because the factors and values that are relevant for the system under test can change with time. Figure 5.5 shows the description of the possible values for the factors *Browser* and *Platform Stack*.

```
<SystemModel>
  <ModelFactor name="Browser">
    <ModelValue name="Chrome" />
    <ModelValue name="Firefox" />
    <ModelValue name="IExplorer" />
  </ModelFactor>
  <ModelFactor name="Platform Stack">
    <ModelValue name=".Net" />
    <ModelValue name="Java" />
  </ModelFactor>
  ...
</SystemModel>
```

Figure 5.5: Description of factors and respective values in the System Model

5.6 Test Discovery

The goal of this step is to generate the required Test Case List (TCL) that need to be executed to fulfill the Relevant Factors on each Test Specification, executing test discovery algorithm presented in Section 4.3, implemented in the class *CombinatorialTestCaseBuilder*.

The *CombinatorialTestCaseBuilder* is one class that extends the existent *ITestCaseBuilder* in NUnit, and represents a new test builder , the Nunit entity responsible to instantiate test cases from test specifications annotated with valid test attributes.

Furthermore, the *CombinatorialTestCaseBuilder* will be responsible to instantiate the Test Cases required for the Test Specifications annotated with attribute *CombinatorialTestCase*.

The *CombinatorialTestCaseBuilder* inspects the class that receives as parameter (in the previous example, WebDriver) and retrieves the Relevant Factors (RF) described in the class, executing the method *relevantFactorsDescription()*, defined in the interface *ICombinatorialIterator*.

The remaining information required to execute the test discovery algorithm are the list with the available Test Environments (TEL) and the System Model (SM).

OutSystems installation process has been changed to call the web service presented in Sub Section 5.5.2 before starting the test discovery. The result obtained by the web service is stored to one file in one configured location in the Test Environment (TE), being fetched by the *CombinatorialTestCaseBuilder*.

The System Model (SM) is other file present in the version control system, being available in the Test Environments during the test discovery.

With all the required information in the *CombinatorialTestCaseBuilder*, it executes the implemented test discovery algorithm presented in the Section 4.3 .

Every time that the *CombinatorialTestCaseBuilder* needs to execute the combinatorial algorithm, the request is done using the command line interface, reused from PICT implementation. The combinatorial algorithm implementation, is one command line interface tool, extending the available PICT implementation and designated by combinatorial engine.

The *CombinatorialTestCaseBuilder* creates the Combinatorial Model (CM), presented in Section 4.2, transforming the internal used structures to the format required by the combinatorial engine. As described in Section 4.2, the Combinatorial Model (CM) is composed by five elements, two mandatory (factors and T) and the remaining are optional, being the CM represented by:

$$CM = \{factors, T, subModels, constraints, availableCombinations\}$$

The combinatorial engine receives the Combinatorial Model (CM) separated by the three following inputs:

1. **inputFile** is one plain text file containing the $\{factors, subModels, constraints\}$. Listing 5.1 represents one example of the **inputFile** for one Test Specification (TS) belonging to the ”WebDriver” group.

```
Browser:         Chrome, Firefox, IExplorer
PlatformStack:   .Net, Java
Database:        Oracle, MySQL, SqlServer
{Browser, ServerRuntime}@2
IF [Database] = ”SqlServer” THEN [PlatformStack] = ”.Net”;
```

Listing 5.1: availableCombinations file example.

2. **availableCombinations** is other plain text file, that is optional and passed the name to command line with the argument ”/t”. table 5.1 represents one example of available combinations for one set of Test Environments.

Table 5.1: *availableCombinations* file example.

Browser	PlatformStack	Database
IExplorer	Java	MySQL
Firefox	Java	MySQL
Chrome	Java	MySQL
IExplorer	.Net	SqlServer
Firefox	.Net	SqlServer
Chrome	.Net	SqlServer
IExplorer	Java	Oracle
Firefox	Java	Oracle
Chrome	Java	Oracle

3. **T** is one command line argument, identified with "/o", being passed during the command line call to the combinatorial engine. Listing 5.2 represents one example of the call made to the combinatorial engine.

```
combinatorialengine inputFile /o 1
    /t availableCombinations > outputFile
```

Listing 5.2: combinatorial engine command line execution.

The combinatorial engine will execute the combinatorial algorithm, presented in Section 4.2, and return the minimum number of combinations that fulfil the Combinatorial Model (CM) description. The result of the combinatorial engine is one plain text file, designated by **outputFile**. Table 5.2 represents one example of the combinations returned by the combinatorial engine.

Table 5.2: Combinatorial engine obtained result

Browser	PlatformStack	Database
IExplorer	.Net	SqlServer
IExplorer	Java	Oracle
Chrome	Java	MySQL
Chrome	.Net	SqlServer
Firefox	.Net	SqlServer
Firefox	Java	Oracle

This approach has the advantage of decoupling the two implementations of the two algorithms, enabling to easily reusing the combinatorial algorithm to multiple implementations of the test discovery algorithm. This advantage is important for implement the test discovery algorithm in other test framework (e.g., JUnit), reusing the implementation of the combinatorial algorithm that is done in C++ and can be executed in both Windows and Linux based test environments.

At the end of the test discovery, one list (TCL) containing the Test Cases is published to the Engineering Dashboard using its web service API, with the $statusDiscovery$ of each Test Case (TC) assigned as $Enabled$, $Disabled$ or $Missing$. The $statusDiscovery$ allows to determine if the TC is selected to be executed in that TE ($statusDiscovery = Enabled$), is selected to execute in another TE ($statusDiscovery = Disabled$) or there is no TE ($statusDiscovery = Missing$) with all the values to execute the TC.

The TCL generated on each of the Test Environments will be very similar, changing only the $statusDiscovery$ of each Test Case (TC). Figure 5.6 represents the $statusDiscovery$ of each Test

Case (TC) in one set of eight Test Environments. The example will be further detailed in Section 6.4.

Test Case	R91-SAN DBOX	R91F08A0 -M0506	R91I07C0- M0506	R91I08C0- O-RDS	R91I08C0- S2012	R91I08C0- S2014	R91J06A0 -O1200	R91W12A 0-O1100
Chrome;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Chrome;Java;MySQL	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Firefox;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Firefox;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled
IExplorer;.Net;SqlServer	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
IExplorer;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled

Figure 5.6: $statusDiscovery$ of each Test Case (TC) exercising one Test Specification (TS)

The main goal of this prototype was to be able to distribute the test specifications by the available Test Environments using combinatorial models to express the Relevant Factors (RF) of each Test Specification (TS). To have the minimum impact possible in the Engineering Dashboard and have the results visible in the Test Environments, the generated tests are being discovered in both test environments with the following syntax:

(Test Case Name)(Combination)[Discovery Status]

Figure 5.7 represents the obtained result in the NUnit graphical user interface. The Test Cases discovered as $Enabled$ appear identified by one green icon with a check symbol in the middle, the $Missing$ appear identified by one yellow icon with a exclamation mark and the $Disabled$ appear identified by one grey symbol.

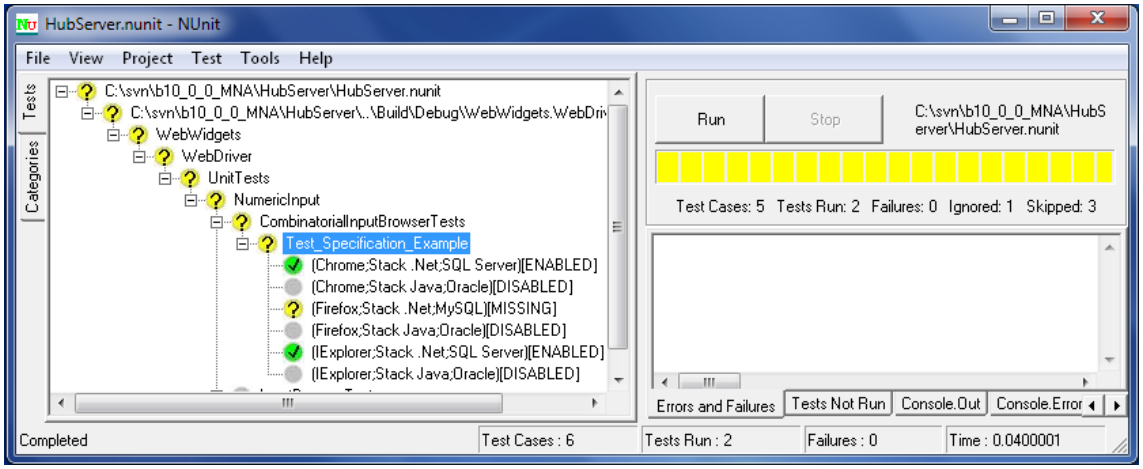

Figure 5.7: Obtained result in NUnit Graphical User Interface

5.6.1 Test Execution

Having the Test Cases that must be executed in each Test Environment, the final step is to execute the automated Test Case List (TCL) that have been returned from the test discovery. Besides the previously generated Test Case List (TCL), the implementation has one additional

parameter to the test execution. The optional parameter is the execution mode, having two possible values:

1. $Strict$ - one execution mode that throws one failure for each Test Case with $Missing$ assigned to the $statusDiscovery$;
2. $Tolerant$ - one execution mode that skips all the Test Cases with $Missing$ assigned to the $statusDiscovery$;

By default, the execution mode has the value $Tolerant$. This is the default value because in most of the cases the goal is to execute in most of the variations that are available. For example, in one developer machine, the developer it is highly probable that the developer does not have all the required values by all the Test Cases. In this example, having this parameter defined as $strict$ will not throw any failure because of insufficient Test Environments to fulfil all the Test Cases.

If the Test Case are being executed in one environment that must have enough available environments to fulfill all the required values in the different Test Case, changing this parameter to $Strict$ will generate one failure for each Test Case (TC) that do not have any environment available to execute.

During the test execution, the Engineering Dashboard selects for each test environment the list of Test Cases with $statusDiscovery = Enabled$ on that Test Environment. At the end of the test execution is produced one report, with the results obtained from the execution of each Test Case (TC).

5.7 Conclusion

The current Chapter aimed to describe an implementation in practice of the test case minimization and distribution technique that was proposed in Chapter 4.

This implementation served as a prototype to assess the ability of the proposed technique to work in practice and in a real scenario. Furthermore, this implementation was crucial to perform several validations and experiments (which will be further detailed in Chapter 6).

Summarizing the current chapter, the two following algorithms were implemented: combinatorial algorithm and test discovery algorithm. This chapter enabled to present one implementation of the proposed technique in one real industry scenario, conducted in the OutSystems context.

The next chapter will describe following validation scenarios that were designed to assess the functionality and correctness of the presented test case minimization and distribution technique implementation.

Chapter 6

Experimental Validation

In this chapter the solution that was presented in this dissertation is evaluated in practice. The work described in this chapter has the goal of validating whether the proposed solution comprises with the goals that were initially established in Chapter 1.

For the validation of the proposed solution, the prototype implementation developed was subject to a set of concrete experiments that are next described in detail.

The results that were observed are very promising and indicate that our solution accomplishes the objectives that were set for it in the first place, allowing to select the heterogeneous test environments where each test case should be executed.

6.1 Introduction

Many modern software systems are designed to be highly-configurable, increasing the difficulty of the validation. The validation of the system under test is only achieved by executing tests in multiple heterogeneous environments.

Having to assure quality in multiple heterogeneous environments creates several problems related with test management, mainly to select the test environments where each test case should execute. With the test suites growing in a large pace, the human effort required to keep the test suites updated and being executed in the proper test environments is cumbersome and this process is highly prone to human errors.

In such a context, it is simply not affordable to manually select the proper test environment where the test must execute: even if the rationale for the distribution of a particular test is still clear (which often is not), the number of tests in the suite invalidates any accurate manual selection. With the accumulation of human errors in the selection of test environments it is nearly impossible to understand the reason why a given test is selected to execute in a chosen heterogeneous environment.

In this work it was proposed one automated test case distribution technique that is able to select the heterogeneous test environments where which test should execute, so that the tests are only selected to execute in the required test environments to exercise the Relevant Factors (RF) described in the Test Specification (TS).

The goal of this chapter is to present a set of different validations that were conducted to verify that the implemented prototype is able to significantly reduce the human effort required to manage the test distribution by the multiple heterogeneous environments.

The implemented prototype is able to retrieve the required information from the Test Specification (TS) so that each Test Case (TC) can be assigned to be executed in the Test Environments that fulfill the requirements of the test case (calculated based on the Relevant Factors (RF) described in the Test Specification (TS)).

This technique allows the developer that is creating the test to describe which factors are relevant to be exercised by the test. The test environments where the test should be executed will be automatically selected based on the test's required factors and the available test environments to execute the test.

The validations that were performed are validations where one sample of real test specifications available at OutSystems were selected to be evaluated, the Relevant Factors (RF) were described in the Test Specification (TS) with the help pf OutSystems developers, and the distribution obtained was evaluated if it matches with the expected. The remainder of this chapter will describe the validation scenarios in detail.

6.2 Evaluating PICT to be used as the combinatorial algorithm

The first kind of validations that were performed was to assure that our proposed solution generates the minimum number of Test Cases to exercise the Relevant Factors (RF) described in the Test Specification (TS). Chronologically, this was the first validation that was conducted in order to evaluate the correctness of the generated Test Cases in the Test Case List (designated by TCL) returned by the combinatorial algorithm.

The first step was to recover one informal description of the rationale how one Test Specification (TS) should be distributed. The first analysed example was a Test Specification (TS) that needed to be executed in all browsers against all platform stacks and where all databases should be exercised at least once. Internally, at OutSystems, this description is shared by one group of Test Specifications, designated by "WebDriver".

A new class designated by "WebDriver" was created containing the previous informal requirement described in the Relevant Factors description, with the three factors that must be exercised with the respective values, being required that each value appears at least once ($T = 1$) except for the two factors browser and platform stack ($T = 2$). Next, is presented the complete representation of the Relevant Factors (RF) example previously described:

$$
\begin{aligned}
RF = \quad & \{factors, T, subModels\} \\
factors = \{ & \\
& Browser \mapsto \{Chrome, Firefox, IExplorer\}\ , \\
& PlatformStack \mapsto \{.Net, Java\}\ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} & \\
T = \quad & 1 \\
subModels = \{ & \\
& (T = 2, \{Browser, PlatformStack\}) \\
\} &
\end{aligned}
$$

The description presented allows to understand the rationale used to distribute the test, meaning that is understandable to the the person that is writing the Test Specification (TS) and to

the implemented prototype. With the description of the Relevant Factors (RF) done, one new Test Specification (TS) was created to validate the prototype based on one already existent Test Specification (TS) existent at OutSystems. The Test Specification (TS) was designated by ”DecimalInput” and is one TS validating that the OutSystems Platform supports decimal inputs in the three browsers against all platform stacks, and each database is exercised at least once. The Test Specification ”DecimalInput” is represented in Figure 6.1.

```
[CombinatorialTestCase(typeof(WebDriver))]
public void DecimalInput()
{
    //actions validating that the OutSystems Platform supports decimal inputs
    actions();
}
```

Figure 6.1: Test Specification ”DecimalInput”

The goal of this experiment was to assure that our proposed solution requires the minimum number of test executions to exercise the Relevant Factors (RF) described in the Test Specification (TS). The first experiment was done with only one Test Environment (TE) available, without having all the values required by the Relevant Factors (RF) description in the Test Specification (TS).

The only Test Environment provided to the algorithm in the first experiment was the most common setup in the OutSystems developers' local environment, being the set of Test Environment List (designated by TEL) composed by only one TE with the values represented in Figure 6.2.

Factor	Value	Developer
Browser	Chrome	Yes
	Firefox	Yes
	IExplorer	Yes
Platform Stack	.Net	Yes
	Java	No
Application Server	IIS	Yes
	JBoss	No
	WebLogic	No
	Wildfly	No
Database	Oracle	Yes
	MySQL	No
	SqlServer	No

Figure 6.2: TEL with the developer Test Environment (TE)

At last, it is required the System Model (SM), one structure to validate the factors and respective values described in the Test Specification (TS) and Test Environment (TE). This structure

contains all the possible factors with the respective values and the constraints between the different values in the form of propositional constraints. The System Model used on this exercise was the following:

$$
\begin{aligned}
SM = \quad & \{factors, constraints\} \\
factors = \{ & \\
& Browser \mapsto \{Chrome, Firefox, IExplorer, Edge\}\ , \\
& PlatformStack \mapsto \{.Net, Java\}\ , \\
& ApplicationServer \mapsto \{IIS, JBoss, WebLogic, WildFly\}\ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} & \\
constraints = \{ & \\
& if(PlatformStack = .Net) then(ApplicationServer = IIS)\ , \\
& if(PlatformStack = Java) then(ApplicationServer in(JBoss, WebLogic, WildFly))\ , \\
& if(PlatformStack = .Net) then(Database = SqlServer) \\
\} &
\end{aligned}
$$

Analysing the previously described System Model (SM) there are several factors and constraints that are not relevant to the Test Specification (TS) that is being exercised. This is the most common use case because the complete description of factors that influence the system behavior is huge, being only described in the System Model (SM) description once and used on the test discovery algorithm for each Test Specification (TS).

The first version implemented of the test discovery algorithm used PICT algorithm instead of the combinatorial algorithm presented in Section 4.2. PICT was used only to calculate the minimum number of Required Combinations (RC) that fulfil the Relevant Factors (RF) description, without evaluating the combinations that are available in the Test Environments.

Exercising this first version of the test discovery algorithm, using the original PICT algorithm, demonstrated to have one unwanted behavior. The Test Cases contained in TCL were composed by multiple Test Cases that do not have any Test Environment (TE) with all the required values to execute the Test Case (TC).

Exercising the first version of the test discovery algorithm with three previously described inputs (TS, TEL and SM) and being executed the algorithm in the Test Environment identified as ”developer”, the Test Case List obtained from the first version of the algorithm was the one represented in Figure 6.3.

Test Case	Developer
Chrome;.Net;SqlServer	Missing
Chrome;Java;Oracle	Missing
Firefox;.Net;SqlServer	Missing
Firefox;Java;Oracle	Missing
IExplorer;.Net;MySQL	Missing
IExplorer;Java;MySQL	Missing

Figure 6.3: TCL from the first version

Analysing the obtained results on the developer Test Environment, the minimum number of test cases to be executed was generated. With the Relevant Factors (RF) described in the Test Specification (TS), the minimum number of test cases that fulfil the Relevant Factors (RF) description is 6, exactly the obtained number.

All the Test Case (TC) contained in the generated TCL have been created with the $statusDiscovery = Missing$, meaning that the only Test Environment (TE) available to execute the Test Specification (TS) will not be able to execute any of the Test Case (TC).

This is one unwanted behavior because the Test Environment (TE) is capable to execute three possible Test Cases that were not generated. Figure 6.4 represents the three Test Cases that the Test Environment (TE) has all the required values and were not generated by the first version of the test discovery algorithm.

Test Case	Developer
Chrome;.Net;Oracle	Enabled
Firefox;.Net;Oracle	Enabled
IExplorer;.Net;Oracle	Enabled

Figure 6.4: Not generated Test Cases that are available in the Test Environment (TE)

With the Test Cases obtained with the first version of the test discovery algorithm, the only Test Environment (TE) available to execute the Test Specification (TS) will not be able to execute any of the Test Case (TC) and this is the expected result.

The problem previously presented was the main motivation to present one extended version of the PICT algorithm, adding the capability of receiving the available combinations, that was presented in Section 4.2.

The following section will present the obtained results with the proposed combinatorial algorithm, validating that the generated TCL matches with the expected result.

6.3 Evaluating the proposed combinatorial algorithm

Using PICT as the combinatorial algorithm in our proposed solution do not provided the expected results. To fulfil this need, one combinatorial algorithm was proposed that also receives in the Combinatorial Model (CM) description the available combinations in the Test Environments.

Exercising the test discovery algorithm with the three previously described inputs (TS, TEL and SM) and being executed the algorithm in the Test Environment identified as ”developer”, the Test Case List obtained from the test discovery algorithm is the represented in Figure 6.5.

Test Case	Developer
Chrome;DotNet;Oracle	Enabled
Chrome;Java;Oracle	Missing
Firefox;DotNet;SqlServer	Missing
Firefox;Java;Oracle	Missing
IExplorer;DotNet;Oracle	Enabled
IExplorer;Java;MySQL	Missing

Figure 6.5: TCL using the proposed combinatorial algorithm

Analysing the obtained results on the developer Test Environment, the minimum number of test cases to be executed was generated. With the Relevant Factors (RF) described in the Test Specification (TS), the minimum number of test cases that fulfil the Relevant Factors (RF) description is 6, exactly the obtained number.

One Test Case (TC) that was possible to be generated and was not generated is the represented in the Figure 6.6, designated by tc_{other}.

Test Case	Developer
Firefox;DotNet;Oracle	Enabled

Figure 6.6: Alternative Test Case (tc_{other}) that the Test Environment (TE) is able to execute

The tc_{other} is one possible Test Case (TC) that the Test Environment (TE) where the test discovery algorithm is executing has all the values required. The tc_{other} was not generated because it will

have to replace one of the previously generated test cases in order to keep the minimum number of test cases that fulfil the Relevant Factors (RF) description.

Replacing any of the previous Test Case (TC) contained in the TCL will exclude one of the variations that need to be exercised and the generated TCL will no longer fulfil the described Relevant Factors (RF). For example, the tc_3 and the tc_{other} are very similar, changing only one value assigned to one factor. The tc_3 has the value "SqlServer" assigned to the factor "Database" and the tc_{other} has the value "Oracle" assigned to the same factor. If the tc_3 was replaced by the tc_{other}, the TCL would contain one more Test Case (TC) with the $statusDiscovery$ as "Enabled" but the value "SqlServer" would no longer be exercised in any of the Test Cases contained in TCL, violating the test requirements of exercising each database at lest once.

The main goal of the test discovery algorithm is to generate the minimum number of Test Cases required to fulfil the Relevant Factors (RF) description. Analysing the obtained TCL, the prototype returned the expected result: the minimum number of Test Cases with the maximum number of Test Case (TC) with $statusDiscovery = Enabled$.

All the Test Cases contained in TCL respect the constraints described in the System Model (SM) and the minimum number of Test Cases in TCL is achieved. The results obtained from this exercise validate that one optimal solution to this scenario was found.

6.4 Validations using multiple Test Environments

In this section, the test discovery is exercised when it is executed in multiple real test environments that are sharing the same code and the same test suite. This exercise will allow to validate that the proposed solution will obtain the expected results when handling with real test environments.

The main goal of performing this validations, is to assure that each Test Case (TC) contained in the TCL generated by the test discovery algorithm in each Test Environment (TE) contains the same Test Cases, only changing the $statusDiscovery$ of each Test Case (TC).

In these validations it will be used the Test Specification (TS) and System Model (SM) described in the Section 6.2. To validate that the proposed solution has the expected behavior, the Test Environments that are being used to validate the latest version of the product released to the company customers were used to exercised our prototype. Being the latest version of the OutSystems Platform, the set of Test Environments available, designated by TEL, was composed by eight Test Environments with the following values available for each factor represented in Figure 6.7.

Factor	Value	R91-SANDBOX	R91F08A0-M0506	R91I07C0-M0506	R91I08C0-O-RDS	R91I08C0-S2012	R91I08C0-S2014	R91J06A0-O1200	R91W12A0-O1100
Browser	Chrome	No	Yes	No	No	No	Yes	Yes	No
	Firefox	No	Yes	No	No	No	Yes	Yes	No
	IExplorer	No	Yes	No	No	No	No	Yes	No
Platform Stack	.Net	Yes	No	Yes	Yes	Yes	Yes	No	No
	Java	No	Yes	No	No	No	No	Yes	Yes
Application Server	IIS	Yes	No	Yes	Yes	Yes	Yes	No	No
	JBoss	No	No	No	No	No	No	Yes	No
	WebLogic	No	No	No	No	No	No	No	Yes
	Wildfly	No	Yes	No	No	No	No	No	No
Database	Oracle	No	No	No	Yes	No	No	Yes	Yes
	MySQL	No	Yes	Yes	No	No	No	No	No
	SqlServer	Yes	No	No	No	Yes	Yes	No	No

Figure 6.7: TEL for the last version of the product

Analysing Figure 6.7, the eight Test Environments available to execute automated tests have one Platform Stack and one Database available on each Test Environment (TE). Regarding the browsers, they are only available to be used in the execution of automated tests in three Test Environments (R91F08A0-M0506, R91I08C0-S2014 and R91J06A0-O1200). This means that any Test Case (TC) generated by the test discovery algorithm will only be able to execute in one of these three Test Environments, because the remaining Test Environments do not have any value available of the factor Browser.

One of the Test Environment (R91I08C0-S2014) presented is missing one of the required browsers in the Relevant Factors (RF), the browser ”IExplorer”. The goal with this set of Test Environments is to validate is validate that the test discovery algorithm creates one Test Case (TC) that exercises the variation with the $platformStack = .Net$ and $browser = IExplorer$, one interaction that is not possible to validate in any of the Test Environment (TE) and should be discovered as ”Missing”.

Exercising the algorithm with the previously described inputs (TS, TEL and SM) and being executed the algorithm in the set of eight Test Environments, the Test Case List obtained from the test discovery algorithm is the presented in Figure 6.8. The TCL generated on each of the Test Environments will be very similar, changing only the $statusDiscovery$ of each Test Case (TC).

Test Case	R91-SAN DBOX	R91F08A0 -M0506	R91I07C0- M0506	R91I08C0- O-RDS	R91I08C0- S2012	R91I08C0- S2014	R91J06A0 -O1200	R91W12A 0-O1100
Chrome;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Chrome;Java;MySQL	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Firefox;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Firefox;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled
IExplorer;.Net;SqlServer	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
IExplorer;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled

Figure 6.8: $statusDiscovery$ of each Test Case (TC) exercising the DecimalInput Test Specification (TS)

Based on the data showed in the Figure 6.8, is possible to validate that the minimum number of test cases to be executed was generated. With the Relevant Factors (RF) described in the Test Specification (TS), the minimum number of test cases that fulfil the Relevant Factors (RF) description is 6, exactly the obtained number.

One pair that requires to be exercised to fulfil the Relevant Factors (RF) was not available in any Test Environment (TE). One Test Case (TC) was created to exercise the variation with the $platformStack = .Net$ and $browser = IExplorer$, one interaction that is not possible to validate in any of the Test Environment (TE) and is discovered as ”Missing” in all Test Environments.

All the remaining Test Cases were generated in all the Test Environments with $statusDiscovery = Disabled$, except for one Test Environment (TE) where each Test Case (TC) is generated with $statusDiscovery = Enabled$.

More validations have been done with more sets of available Test Environments, confirming that the minimum number of Test Cases are always generated with the maximum number possible of Test Case (TC) with $statusDiscovery = Enabled$ in the Test Environments.

Without the proposed solution, the same Test Specification (TS) was being discovered in the eight Test Environments with three different Test Cases in each Test Environment (TE), representing the three browsers, meaning that one Test Specification (TS) when executed in all the Test Environments was executed twenty four times, representing the twenty four variations (eight Test Environments multiplied by three browsers).

In order to minimize the number of test executions there were already other mechanisms that allow that one Test Environment do not discover the Test Specification (TS), only generating three Test Cases (the three different browsers) in the desired Test Environments. This is not the desired behavior, because the Test Specification (TS) were directly selected to execute in each Test Environment, and the rationale behind this selection was lost or not clear.

The proposed solution allows the person that is creating the test to describe which factors are relevant to be exercised in the test execution. The person that is creating the test, is the person with more context about the variations that the test should exercise.

The test environments where the test should be executed will be automatically selected based on the previous factors and the available test environments to execute the test. The rationale behind the selection of the Test Environments where the test will be executed is understandable

to the person creating the Test Specification (TS) and to the test discovery algorithm.

The TCL obtained during this exercise enabled to obtain only six Test Cases to the previously described Test Specification (TS), reducing from the previously twenty four executions to six when the Test Specification (TS) is executed in all Test Environments.

6.5 Adding IExplorer to one Test Environment

In the previous exercise it was not available any Test Environment (TE) with $platformStack = .Net$ and $browser = IExplorer$. When this situation happens, the developer will troubleshoot the problem and it is possible to configure one Test Environment (TE), adding the required browser to one Test Environment (TE) with $platformStack = .Net$. After solving the problem, it is only required to update the Test Environment description.

The set of Test Environments available, designated by TEL, is composed by eight Test Environments. Figure 6.9 represents the available values for each factor with IExplorer added to one Test Environment (TE).

Factor	Value	R91-SANDBOX	R91F08A0-M0506	R91I07C0-M0506	R91I08C0-O-RDS	R91I08C0-S2012	R91I08C0-S2014	R91J06A0-O1200	R91W12A0-O1100
Browser	Chrome	No	Yes	No	No	No	Yes	Yes	No
	Firefox	No	Yes	No	No	No	Yes	Yes	No
	IExplorer	No	Yes	**Yes**	No	No	No	Yes	No
Platform Stack	.Net	Yes	No	Yes	Yes	Yes	Yes	No	No
	Java	No	Yes	No	No	No	No	Yes	Yes
Application Server	IIS	Yes	No	Yes	Yes	Yes	Yes	No	No
	JBoss	No	No	No	No	No	No	Yes	No
	WebLogic	No	No	No	No	No	No	No	Yes
	Wildfly	No	Yes	No	No	No	No	No	No
Database	Oracle	No	No	No	Yes	No	No	Yes	Yes
	MySQL	No	Yes	Yes	No	No	No	No	No
	SqlServer	Yes	No	No	No	Yes	Yes	No	No

Figure 6.9: TEL updated with IExplorer on one Test Environment (TE)

Exercising the algorithm with the previously described inputs (TS, TEL and SM) and being executed the algorithm in the updated set of eight Test Environments, the Test Case List obtained from the test discovery algorithm is the presented in Figure 6.10.

Test Case	R91-SAN DBOX	R91F08A 0-M0506	R91I07C0 -M0506	R91I08C0 -O-RDS	R91I08C0 -S2012	R91I08C0 -S2014	R91J06A 0-O1200	R91W12A 0-O1100
Chrome;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Chrome;Java;MySQL	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Firefox;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Firefox;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled
IExplorer;.Net;MySQL	Disabled	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled
IExplorer;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled

Figure 6.10: $statusDiscovery$ of each Test Case (TC) exercising the DecimalInput Test Specification (TS) after adding IExplorer to one Test Environment (TE)

The Test Cases generated composing the Test Case list (designated by TCL) are different from the generated in the previous exercises, but they continue to fulfil the Relevant Factors (RF) description in the Test Specification (TS) and continues to represent the minimum number of generated Test Cases possible. With the simple update in the Test Environments, the test discovery algorithm was able to evaluate the new set of Test Environments and generate Test Cases that are all with $statusDiscovery = Enabled$ in one of the Test Environments.

The obtained results were only possible to achieve with the proposed combinatorial algorithm, described in Section 4.2, and implemented during this work as the combinatorial engine described in Section 5.6. Figure 6.11 represents the results obtained when evaluating the test discovery algorithm, using PICT instead of the implemented combinatorial engine.

Test Case	R91-SAN DBOX	R91F08A 0-M0506	R91I07C0 -M0506	R91I08C0 -O-RDS	R91I08C0 -S2012	R91I08C0 -S2014	R91J06A 0-O1200	R91W12A 0-O1100
Chrome;.Net;Oracle	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
Chrome;Java;MySQL	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Firefox;.Net;MySQL	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
Firefox;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled
IExplorer;.Net;SqlServer	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
IExplorer;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled

Figure 6.11: $statusDiscovery$ of each Test Case (TC) exercising the DecimalInput Test Specification (TS) with PICT

The results represented in the Figure 6.11 were the main motivation to propose one new combinatorial algorithm instead of using PICT. Our proposed algorithm is able to receive the available combinations in the Test Environments and generate the minimum number of required combinations, having the maximum number of combinations available in the the Test Environments.

6.6 Modifying the Relevant Factors description

The following validation was to assure that the test discovery algorithm was able to handle with changes in the Relevant Factors (RF) description.

The factors that influence one Test Specification (TS) evolve with time, one concrete example is the appearance of a new browser that needs to be supported, meaning that the previously described Test Specification (TS) will require to be exercised against this new browser. The most recent browser that appeared in the market was "Edge" from Microsoft and the goal of this section is to evaluate the results obtained when the Relevant Factors (RF) description is updated. The updated Relevant Factors (RF) description is the following:

$$
\begin{aligned}
RF = \quad & \{factors, T, subModels\} \\
factors = \{ & \\
& Browser \mapsto \{Chrome, Firefox, IExplorer, \underline{Edge}\}\ , \\
& PlatformStack \mapsto \{.Net, Java\}\ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} & \\
T = \quad & 1 \\
subModels = \{ & \\
& (T = 2, \{Browser, PlatformStack\}) \\
\} &
\end{aligned}
$$

The only difference in the previously presented Relevant Factors (RF) description is that "Edge" has been added as one required variation in the factor Browser. The value "Edge" has already present in the System Model (SM) description and it is identified by the test discovery algorithm as one valid value for the factor Browser.

Exercising the algorithm with the previously described inputs (TS, TEL and SM) and being executed the algorithm in the set of eight Test Environments, the Test Case List obtained from the test discovery algorithm is the presented in Figure 6.12.

Test Case	R91-SAN DBOX	R91F08A 0-M0506	R91I07C0 -M0506	R91I08C0 -O-RDS	R91I08C0 -S2012	R91I08C0 -S2014	R91J06A 0-O1200	R91W12A 0-O1100
Chrome;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Chrome;Java;MySQL	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Edge;.Net;SqlServer	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
Edge;Java;MySQL	Missing	Missing	Missing	Missing	Missing	Missing	Missing	Missing
Firefox;.Net;SqlServer	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled
Firefox;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled
IExplorer;.Net;MySQL	Disabled	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled
IExplorer;Java;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled

Figure 6.12: TCL with the Edge value added

Analysing the obtained results represented in Figure 6.12, the TCL generated represents the minimum number of Test Cases that are required to fulfil the updated Relevant Factors (RF)

description. There are two new Test Cases, exercising the new value ”Edge” against the two platform stack (.Net and Java) and all databases continue to appear at least once. The two new Test Cases will appear with $statusDiscovery = Missing$ in all Test Environments because there is not any Test Environment (TE) with the value Edge, meaning that there is not any Test Environment (TE) with all the values that the Test Case (TC) is requiring.

This is the expected behaviour of the system, those Test Cases with $statusDiscovery = Missing$, allow the developers to understand that they need to change their test environments to support the test requirement: being able to test the Edge browser. Currently, the existent test discovery at OutSystems, does not detect this new test requirement. It requires extra developer effort to always try to evaluate if the current test environments are fulfilling the needed requirements.

6.7 Evaluating different Test Specifications

The industrial context that was described in Chapter 2 and its complexity demands that the implemented prototype is able to deal with multiple different scenarios.

Assuring that the proposed solution works for any Test Specification (TS) with one valid Relevant Factors (RF) description is essential. With this goal in mind, more informal descriptions of rationale that should be used to distribute available Test Specifications at OutSystems were recovered and exercised.

The following Test Specification (TS) that was exercised was one Test Specification (TS) belonging to the group of ”SAP Client”. This group of Test Specifications validate that the connectivity with one external service (SAP) has the expected behavior. The Test Specifications belonging to this group requires to be exercised in the two platform stacks, the four application servers and in the three databases.

The previous informal requirement was described in the Relevant Factors description, with the three factors that must be exercised with the respective values, being required that each value appear at least once ($T = 1$), one criteria that was defined that the developer creating the test as defined. Following is the complete representation of the Relevant Factors (RF) example previously described:

$$
\begin{aligned}
RF = \quad & \{factors, T, subModels\} \\
factors = \{ \quad & \\
& PlatformStack \mapsto \{.Net, Java\} \ , \\
& ApplicationServer \mapsto \{IIS, JBoss, WebLogic, WildFly\} \ , \\
& Database \mapsto \{SqlServer, Oracle, MySQL\} \\
\} \quad & \\
T = \quad & 1 \\
subModels = \quad & \{\emptyset\}
\end{aligned}
$$

Exercising the algorithm with the previously described inputs (TS, TEL and SM) and being executed the algorithm in the set of eight Test Environments, the Test Case List obtained from the test discovery algorithm is the presented in Figure 6.13.

Generated	R91-SAN DBOX	R91F08A0 -M0506	R91I07C0- M0506	R91I08C0- O-RDS	R91I08C0- S2012	R91I08C0- S2014	R91J06A0 -O1200	R91W12A 0-O1100
.Net;IIS;SqlServer	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Java;JBoss;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled
Java;WebLogic;Oracle	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled
Java;Wildfly;MySQL	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled

Figure 6.13: TCL of the Test Specification (TS) contained in ”SAP Client”

The results represented in figure 6.13 enabled to validate that the Relevant Factors (RF) description allows to describe the factors that must be exercised for multiple Test Specifications.

The minimum number of Test Cases that fulfil the Relevant Factors (RF) description is four, the number obtained. Each of the Test Case (TC) is with $statusDiscovery = Enabled$ in only one Test Environment (TE), minimizing the number of test executions to the minimum required to fulfil the Relevant Factors (RF) description.

The same experiments have been done with others Test Specification (TS) available at OutSystems, where one informal description was collected from the person that created the Test Specification (TS) and the factors that influence the test execution. Following the remaining collected examples are briefly presented because they only show the same behavior that the previously described Test Specifications.

The following Test Specification (TS) that was exercised was one Test Specification (TS) belonging to the group ”PreCaching”. This group of Test Specifications validate the expected behavior of one generated application when the cache of the browser is in different states. The Test Specifications belonging to this group requires to be exercised in two browsers against two platform stacks. The two browsers selected were ”Chrome” and ”IExplorer”, excluding intentionally ”Firefox” because the Test Specification (TS) exercised one functionality in the browser that Mozilla Firefox do not support. Based on the previously described factors, the Relevant Factors (RF) description was done to exercise both factors and the expected result was obtained, represented in Figure 6.14.

Generated	R91-SAN DBOX	R91F08A0 -M0506	R91I07C0- M0506	R91I08C0- O-RDS	R91I08C0- S2012	R91I08C0- S2014	R91J06A0 -O1200	R91W12A 0-O1100
Chrome;.Net	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Chrome;Java	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
IExplorer;.Net	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
IExplorer;Java	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled

Figure 6.14: TCL of the Test Specification (TS) contained in ”PreCaching”

6.8 Conclusion

The goal of this chapter is to present a set of different validations that were conducted to verify that the implemented prototype is able to significantly reduce the human effort required to manage the test distribution by the multiple heterogeneous environments.

The implemented prototype showed how it was possible to apply the proposed solution to the OutSystems context without any major change to their test automation system, allowing the elimination of all human effort related to disabling tests in test environments.

Furthermore, the implemented prototype is able to retrieve the required information from the Test Specification (TS) so that each Test Case (TC) can be assigned to be executed in the Test Environments that fulfill the requirements of the test case (calculated based on the Relevant Factors (RF) described in the Test Specification (TS)).

This technique allows the developer that is creating the test to describe which factors are relevant to be exercised in the test execution. The test environments where the test should be executed will automatically be selected based on the previous factors and the available test environments to execute the test. It also showed us to support changes in the required factors, and help developers detect fast and effortless if the there are missing supported configurations in the available test environments

The proposed technique demonstrated to be able to select the correct test environments where the tests should execute, selecting the minimum number of test environments required to exercise the factors described for each test.

To conclude this chapter, the experimental results conducted to assess the functionality and value of the implemented prototype, providing evidence that the goals of the proposed technique were effectively and successfully achieved, as they were stated in Chapter 1.

Chapter 7

Conclusion

This chapter presents the conclusions that can be drawn from the work described in this dissertation. The contents of this chapter should give a summary of the proposed solution and make it clear that it fulfils the goals that were set on the problem statement. Furthermore, some directions for future work are briefly described.

7.1 Conclusion and Final Results

This dissertation's main motivation was to propose one automated test case distribution technique that is able to select the heterogeneous test environments where each test case should execute.

This dissertation was conducted in an industrial context, on the facilities of the OutSystems, more specifically inside the organization's R&D group. This context was of great interest since the software developed by OutSystems is indeed a highly configurable system. Furthermore, OutSystems, as many other companies these days, was suffering from the problem of having a large test suite, leading to challenges described below.

A large test suite requires huge human effort to keep the distribution of test cases over the multiple heterogeneous environments up-to-date. Addressing this problem would allow the present dissertation to give a practical contribution to a industrial context and also to evaluate that contribution in a real scenario.

In the related work study that is part of this dissertation, two different areas of research were presented with some research done to solve related problems. The first research area presented was the work done to distribute the test cases over multiple environments. None of the studied approaches enabled to distribute the same Test Specification (TS) over different factors (e.g.: operating system, web browser, browser screen resolution) that influence the behavior of the System Under Test (SUT). To achieve this goal, minimization techniques were studied and presented.

There has been considerable studies about minimizing the number of tests needed to fulfil a Test Specification (TS) required input values. These studies showed that there were open opportunities to innovate and to propose a new solution that would extend the capabilities of the existing tests minimization techniques, adapt them to the problem of minimizing the number of system configurations against a Test Specification should execute, creating a test distributions and minimization technique.

The technique that was proposed makes use of the combinatorial techniques to describe the

Relevant Factors (RF) to one Test Specification and then generates the minimum number of test cases required to fulfil the previously described Relevant Factors (RF). This technique allows that only the necessary test cases were generated and each test case was only selected to execute in one test environment that fulfils the test requirements. The number of test executions was significantly reduced, which implied significant costs reduction, and each test case only appears in the test environment that is able to execute the test case. Also, the error prone task of manually defining in what tests enviorments should a given tests execute was replaced by an automated process. This lead to much less human effort and configuration errors.

The solution that was proposed was also implemented in the context of OutSystems and as a side goal of this implementation, it was possible to evaluate its behavior.

Concerning the evaluation of the proposed technique, it was crucial to study whether it would effectively comprise with the set of requirements that were described in the beginning of this dissertation. It is important to remember the initial requirements that were set:

The main goal of this dissertation is to solve the problem of test case distribution on highly configurable environments. More concisely, the challenge tackled in this work should include solutions that meet the following requirements:

R1) It must deal with different relevant factors for each test.

R2) It must scale to large and continuously growing systems.

R3) It must deal with a constant change in factors that influence the system, test suite and test environments.

R4) It must deal with the lack of test environments that can fulfill the relevant factors, returning a test failure if there are not enough environments.

R5) It must apply to real environments, within real validation scenarios.

R6) It must be deterministic, given the same input must always return the same selection.

The validations that were conducted during this work aimed to assess the ability of the proposed technique to fulfil the previous requirements. During the exercises presented on Chapter 6, this requirements have been validated. The Test Specification (TS) was exercised with multiple relevant factors, which fulfills the requirement R1) that was presented above. Regarding R2) the technique was exercised with the addition of new values at some factors to the Relevant Factors (RF) description, validating this requirement.

All the exercises presented in Chapter 6 change the description of the System Model (SM), Test Environment (TE) or Test Specification (TS), proving to be capable of handling R3). Some exercises have been done there was not any Test Environment available that could fulfil that can fulfil the Relevant Factors (RF). The proposed technique generated the maximum number of Test Cases that was able to execute, and the remaining were identified with missing test environment, fulfilling R4).

During Chapter 6, all the validation scenarios where based on real Test Specifications already existing at OutSystems and the proposed technique was able to generate the expected results,

validating R5). Regarding R6), the proposed technique is by design deterministic, returning always the same generated Test Cases when provided the same inputs.

In summary, the work and the conclusions of its validations, all the initially defined goals and requirements were met. This Dissertation has presented and successfully validated, an automated technique that is capable of selecting the heterogeneous test environments where each Test Specification (TS) must be exercised.

7.2 Future Work

The proposed technique was implemented using one internally developed tool responsible to manage the test automation infrastructure, designated by Engineering Dashboard.

Currently, OutSystems is changing the quality assurance process and associated tools. One new internal tool was created to achieve that, designated by CINTIA (**C**ontinuous **INT**egration and **I**ntelligent **A**lert System).

The main difference between the two tools, it that CINTIA works on a continuous integration model, executing the tests incrementally by stages, only executing the following stage after the previous one is finished. Normally multiple stages are executing at the same time and each stage have their own set of test environments.

The implementation of the prototype needs to evolve, adapting to the new tool used to execute the automating tests.

Bibliography

[BG98] K. Beck and E. Gamma. Test infected: programmers love writing tests. Technical Report 3(7), 1998. 20

[BS04] J. Bach and P. Shroeder. Pairwise testing: A best practice that isn't. In *Proceedings of the 22nd Pacific Northwest Software Quality Conference*, pages 180--196, 2004. 25

[BY98] Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513. West, 1998. 25

[CB10] N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Comput. Netw.*, 54(5):862–876, April 2010. Available from: `http://dx.doi.org/10.1016/j.comnet.2009.10.017`. 20

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, July 1997. Available from: `http://dx.doi.org/10.1109/32.605761`. 25, 26

[Cla99] James Clark. Xsl transformations (xslt) version 1.0. World Wide Web Consortium, Recommendation REC-xslt-19991116, November 1999. 53

[Cze] Jacek Czerwonka. Pairwise testing in real world practical extensions to test case generators. xv, 26, 27, 30, 33, 34, 37, 38

[Dav12] Burns David. *Selenium 2 Testing Tools: Beginner's Guide*. Packt Publishing, 2012. 20, 22

[DCBM06] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. Gridunit: Software testing on the grid. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 779–782, New York, NY, USA, 2006. ACM. Available from: `http://doi.acm.org/10.1145/1134285.1134410`. xv, 20, 21, 22

[DES+97] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing: Experience report. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 205–215, New York, NY, USA, 1997. ACM. Available from: `http://doi.acm.org/10.1145/253228.253271`. 25

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 20

[GOA05] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005. 26

[Kap01] G. M. Kapfhammer. Automatically and Transparently Distributing the Execution of Regression Test Suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, 2001. 20, 21

[LML87] Michael J. Litzkow, Matt W. Mutka, and Miron Livny. CONDOR : a hunter of idle workstations. Technical Report TR 0730, University of Wisconsin (Madison, WI US), 1987. Available from: http://opac.inria.fr/record=b1017651. 20

[LT98] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=645432.652389. 25

[Man85] Robert Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, October 1985. Available from: http://doi.acm.org/10.1145/4372.4375. 25

[Mic15] Microsoft. Pairwise independent combinatorial testing. https://github.com/Microsoft/pict, 2015. 30, 54

[PCG+06] Andrew Pavlo, Peter Couvares, Rebekah Gietzel, Anatoly Karp, Ian D. Alderman, Miron Livny, and Charles Bacon. The NMI build & test laboratory: Continuous integration framework for distributed computing software. In *Proceedings of the 20th Conference on Systems Administration (LISA 2006), Washington, DC, USA, December 3-8, 2006*, pages 263–273, 2006. Available from: http://www.usenix.org/events/lisa06/tech/pavlo.html. xv, 20, 21

[Ric95] R. Richardson. *The Ultimate Batch File Book!* McGraw-Hill, 1995. Available from: https://books.google.pt/books?id=bDlYNgAACAAJ. 53

[RMR02] D. Richard, Kuhn Michael, and J. Reilly. Nasa goddard space flight center, 4-6 december, 2002. an investigation of the applicability of design of experiments to software testing, 2002. 25

[SFM00] Benjamin D. Smith, Martin S. Feather, and Nicola Muscettola. Challenges and methods in testing the remote agent planner. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, pages 254–263, 2000. Available from: http://www.aaai.org/Library/AIPS/2000/aips00-027.php. 25

[She94] G. Sherwood. Effective testing of factor combinations. *Proceedings of the Third International Conference on Software Testing, Analysis and Review*, pages 133–166, 1994. 26

[Tat87] K. Tatsumi. Test-case-design support system. In *Proceedings of the International Conference on Quality Control (ICQC)*, pages 615––620, Tokyo, 1987. 25, 26

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, February 2005. Available from: http://dx.doi.org/10.1002/cpe.v17:2/4. 20

[Wei07] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, December 2007. Available from: http://doi.acm.org/10.1145/1327512.1327513. 20

[WK01] Dolores R. Wallace and D. Richard Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. In *ACS/ IEEE International Conference on Computer Systems and Applications*, pages 301–311, 2001. 24