UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Test automation and code dependencies in highly complex environments

Diogo Manuel Raposo Oliveira

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

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

Covilhã, Outubro de 2015

Acknowledgements

First of all I would like to thank Professor João Paulo Fernandes and Professor Simão Melo de Sousa for coordinating this dissertation. I would like to thank you for all your constant support and feedback regarding my dissertation, and for proposing me a challenging and very interesting research subject. Also, I would like to thank you for spending part of your personal time to help me and support me.

Then, I would like to thank Miguel Costa Antunes and António Melo for coordinating this dissertation in the industrial context of OutSystems. Your support was simply amazing. I will not forget the personal time you spent to help me. Really, it was great to work with you!

I would also like to thank all the people whom I have worked with at OutSystems and that helped me during this time. Special thanks do Bruno Loureiro, Jandira Peres, Ricardo Martins, Carlos Almeida, João Miranda, João Rosado, Rui Eugénio, Sérgio Silva, Stoyan Garbatov, and any other person that helped me and that I might have forgotten to mention here.

Now in portuguese...

Gostaria de começar por agradecer às pessoas mais importantes da minha vida, que são os meus pais e o meu irmão. Aos meus pais, um muito obrigado por tudo o que sempre fizeram por mim e que me permitiu chegar a esta fase da minha vida. Devo-vos muito daquilo que sou como pessoa, e devo-vos todas as condições que sempre me deram para tudo!

Ao meu irmão, um agradecimento especial por todo o apoio que sempre me deu, pela preocupação que sempre teve comigo, e pelo incondicional apoio e ajuda em tudo o que precisei.

À Joana, a minha namorada, um obrigado por tudo. Mesmo em momentos mais complicados, a tua ajuda foi preciosa, e o teu apoio fez relamente a diferença.

A todas as pessoas com quem partilhei o último ano da minha vida mais de perto, em especial: o meu irmão, Jorge Oliveira, o Tiago Lograd, o José Braz, a Cátia Martins e a Margarida Rodrigues. Um obrigado por todo o vosso apoio em todas as alturas.

Um grande obrigado ao meu amigo Nuno Garcia, por tudo o que partilhámos na vida pessoal e no decorrer das nossas dissertações! Um grande obrigado ao Francisco Vigário pelas conversas diárias sempre bem humoradas.

Um grande obrigado a todos os meus amigos da minha Covilhã com quem fui partilhando os últimos tempos, em especial: Vitor Rolo, Pedro Santos, Vitor Pereira, André Reis, Rodrigo Simões, Pedro Fael, José Tourais, Bruno Aguilar, Samuel Casteleiro, Nélson Belo, Sérgio Pinto, Patrício Albuquerque, João Real, António Duarte, João Amaral e Nuno Martins.

Um obrigado à Ana Ramos, ao Carlos Silva e à Flávia Silva.

Obrigado à Ana Baía, a minha madrinha, por tudo o que passámos juntos ao longo destes anos,

não me esquecerei disso.

Um obrigado a todos aqueles que não tenha mencionado, mas que em alguma altura da minha vida estiveram ao meu lado e me apoiaram. O facto de ter escrito estes agradecimentos 5 minutos antes de imprimir a dissertação obrigou-me a pouco tempo para esforço de memória.

Obrigado a todos mais uma vez, depois pago um copo a quem me esqueci de agradecer!

Resumo

A sociedade moderna evolui nos dias de hoje a um ritmo nunca antes observado. A este respeito, pode ainda constatar-se que o desenvolvimento tecnológico tem sido um dos principais pilares que sustentam essa evolução. Neste contexto, os sistemas de *software* são essenciais, e é cada vez mais importante assegurar que esses sistemas chegam ao mercado em tempo útil e com garantias de qualidade. Na tentativa de assegurar estes objetivos, têm sido propostas novas metodologias e ambientes de desenvolvimento de *software*, bem como modernas ferramentas que permitem a sua evolução e integração contínua, e ainda estratégias cada vez mais evoluídas para verificação e validação de *software*.

No que diz respeito à verificação e validação de *software*, que de diferentes formas procura assegurar a sua qualidade, o desenho e execução de testes desempenha um papel central. Contudo, a constante evolução dos sistemas de *software* implica que os artefatos de teste evoluam ao mesmo ritmo. De facto, observa-se também para esses artefatos uma evolução rápida, e em particular um aumento muito significativo do número de casos de teste com que é necessário lidar.

Um grande desafio da engenharia de *software* moderna prende-se precisamente com a necessidade de desenvolver mecanismos que permitam que a fase de execução de testes de *software* implique menos custos a nível de tempo e dinheiro, mas também de esforço humano. E esta redução deverá necessariamente ser conseguida sem colocar em causa a qualidade desta mesma fase.

No presente trabalho é proposta uma técnica de seleção automática de casos de teste de *software*, que pretende, em cada iteração da execução de testes, apenas selecionar para execução testes cujo resultado possa ser influenciado pelas alterações que tenham sido feitas ao código do *software* a ser testado ou aos próprios testes. A técnica proposta na presente dissertação baseia-se na análise de dependências estáticas e dinâmicas entre os testes de *software* e o código do próprio *software*, bem como na identificação de alterações entre duas versões do mesmo *software*.

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 fins de experimentação e validação da técnica de seleção automática de casos de teste de *software*, e os resultados finais demonstram que a técnica proposta não colocou em causa a capacidade de deteção de falhas no *software* e apresentou benefícios significativos relativamente à redução do tempo dispendido na execução de testes.

Palavras-chave

Engenharia de *software*, testes de *software*, seleção de casos de teste, integração contínua

Resumo alargado

O desenho e execução de testes de *software* são amplamente utilizados para validar que os sistemas de *software* se comportam como esperado, de acordo com os seus requisitos. Efetivamente, os testes de *software* são a técnica mais utilizada para encontrar falhas nos sistemas de *software* antes de estes serem disponibilizados aos seus utilizadores finais.

Nos dias de hoje, observa-se uma evolução cada vez mais acentuada das metodologias, tecnologias e paradigmas utilizados no desenvolvimento de *software*. Esta evolução, aliada ao facto de diversos sistemas de *software* serem desenvolvidos e/ou melhorados ao longo de vários anos, fazem com que a tarefa de testar esses sistemas seja cada vez mais exigente a diversos níveis.

Em particular, têm surgido metodologias de desenvolvimento cada vez mais ágeis, e paradigmas de desenvolvimento que permitem a integração e monitorização contínua de *software*. Numa tentativa de melhorar a produtividade no seu desenvolvimento, estes avanços requerem o processo de desenvolvimento de *software* ocorra em iterações (de desenvolvimento e teste) cada vez mais rápidas.

O objetivo do trabalho abordado neste dissertação está relacionado com a otimização da fase de desenvolvimento de *software* no que diz respeito à execução de testes em ambientes complexos e de larga escala. Assim, pretende-se, sem comprometer a qualidade da fase de testes, reduzir o tempo dispendido na execução automática de testes de forma a que os programadores possam obter *feedback* sobre o código que desenvolveram.

Com este objetivo em mente, pretende-se ao longo do trabalho apresentado nesta dissertação propor uma solução que tenha a capacidade de selecionar apenas os casos de teste que estejam relacionados com as alterações que foram efetuadas ao código do *software* a ser testado. Uma solução deste género, se provada eficaz e segura, garantiria que apenas os testes estritamente úteis são executados em cada iteração do desenvolvimento de software, com o aumento de eficiência que daí advém.

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 fase de testes de *software*. Estes trabalhos foram cuidadosamente estudados no contexto da presente dissertação, com o objetivo de identificar oportunidades de investigação e de evolução para as soluções propostas até à data, bem como identificar potenciais lacunas ou pontos fracos que esses mesmos trabalhos apresentem. Se, por um lado, a maioria destes trabalhos aponta diversas bases e direções interessantes para uma investigação continuada e sustentada, por outro lado existem diversos pontos fracos que fazem com que os trabalhos estudados não sejam apropriados para utilização em contextos reais, de larga escala e com uma complexidade elevada, por possuírem significativas limitações ou assunções que dificilmente são espelhadas em cenários industriais.

A solução proposta nesta dissertação para resolver o problema da seleção de casos de teste de *software* em ambientes complexos é uma solução que utiliza mecanismos de análise estática e dinâmica de código, permitindo a criação de uma relação de dependência entre casos de

teste de *software* e diversas partes do código desse mesmo *software*. Este tipo de análise de dependências, aliado à identificação das alterações que foram feitas ao código do *software*, permitem que para cada iteração do processo de desenvolvimento de *software* seja selecionado um conjunto específico de testes cujo seu resultado possa ser influenciado devido às alterações efetuadas.

A solução proposta é, portanto, uma técnica de seleção de casos de teste de *software* que utiliza informação de dependências de código provenientes de vários tipos de análises de código diferentes, e que é pensada para aplicação em contextos reais e de forma automatizada, sem que seja necessária intervenção humana. A técnica proposta é uma técnica que tem a capacidade de se adaptar a sistemas complexos e de natureza distribuída, sendo também escalável e com capacidade de se adaptar ao constante crescimento dos sistemas de *software*. Para além disso, a técnica proposta não assume que sejam necessárias quaisquer tipo de adaptações aos ambientes dos sistemas de *software* em que venha a ser aplicada.

A técnica de seleção automática de casos de teste apresentada foi implementada em prática num ambiente real, e nomeadamente no contexto de uma empresa internacional de desenvolvimento de *software* que enfrenta todos os desafios de escala e complexidade que a indústria de desenvolvimento de *software* enfrenta em geral nos dias que correm. Esta mesma implementação foi utilizada para realizar a seleção de testes de *software* num contexto de desenvolvimento de *software* real, num conjunto de casos reais não escolhidos nem fabricados de forma artificial.

Na presente dissertação diversas formas de avaliação foram utilizadas para analisar a funcionalidade dos diversos passos da técnica de seleção de casos de teste proposta em particular, bem como a técnica na sua totalidade e a sua eficácia na seleção dos casos de teste que efetivamente deveriam ser selecionados. Entre outro tipo de validações, foi criado um mecanismo de validação automatizada que permite aplicar a técnica sempre que existam novas alterações do código e essas alterações sejam enviadas para o sistema de controlo de versões, e comparando os resultados da execução dos testes selecionados com os resultados da execução de todo o conjunto de testes existente.

Relativamente aos resultados obtidos, nos diversos cenários de validação a técnica proposta na presente dissertação apresentou o comportamento esperado, sendo capaz de selecionar casos de teste relacionados com alterçaões ao código, e ainda assim sem comprometer a capacidade de deteção de falhas. Vários casos foram observados a partir da metodologia de validações automatizadas referido acima, e em todos os casos todos os testes selecionados detetaram as mesmas falhas que a bateria de testes completa teria detetado.

Abstract

Modern society is nowadays evolving at a pace that has never been witnessed before. Regarding this evolution, it can be also observed that the technological evolution has been one of its main pillars. In this context, software systems have a crucial role and it is increasingly important to ensure that these systems reach the market on time and with quality guarantees. In order to ensure these goals, several new methodologies and software development environments have been proposed, together with modern tools that enable software systems evolution and continuous integration, as well as increasingly advanced strategies for software verification and validation.

Concerning software verification and validation, which in different ways aim to ensure software quality, design and execution of software tests play a key role. However, the constant evolution of software system implies that test artifacts are able to evolve at the same pace. In fact, these artifacts are indeed evolving fast as well, and this implies that it is needed to deal with test suites of constantly increasing size.

A major challenge of modern software engineering has to do precisely with the need to develop mechanisms that allow the software tests execution stage to imply less costs in terms of time and money, but also of human effort. These mechanisms must necessarily achieve a reduction in terms of costs without jeopardizing the quality of the software tests execution stage.

In this dissertation an automatic test case selection technique is proposed, which aims, in each iteration of software test executions, to only select for execution test cases whose results might be influenced by the changes that have been made to the code of the software under test. The proposed technique is based on static and dynamic dependency analysis between software tests and the software code (and the code itself), as well as on the identification of the differences between different software code versions.

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 final results demonstrated that the proposed technique is effective in terms of its software fault detection capabilities and also showed significant benefits in what concerns the time spent running software tests.

Keywords

Software engineering, software testing, test case selection, continuous integration

Contents

List of Figures

List of Tables

Acronyms

PaaS Platform as a Service

QA Quality Assurance

IDE Integrated Development Environment

DE Development Environment

PS Platform Server

ILP Integer Linear Programming

EFSM Extended Finite State Machine

CDG Control Dependence Graph

PDG Program Dependence Graph

SDG System Dependence Graph

CFG Control Flow Graph

ICFG Intraprocedural Control Flow Graph

JIG Java Interclass Graph

VCS Version Control System

SVN Apache Subversion

AOP Aspect-oriented Programming

API Application Programming Interface

ECMA European Computer Manufacturers Association

CIL Common Intermediate Language

SOAP Simple Object Access Protocol

DSL Domain Specific Language

Chapter 1

Introduction

Software testing is by far the most widely used technique to find defects in software and particularly in the context of industrial software development. It is also well accepted that the more a program is exercised in different ways and with different kinds of concrete tests, the more chances exist of encountering (and later removing) its faults [MS04, Sin11].

In the context of high-scale and complex systems that are being developed and evolving for several years, the need to thoroughly test those systems leads to the consumption of significant amounts of resources, such as human effort, time and money. In particular, the time taken to test a real software system is often an obstacle to achieve short release cycles, which are more and more frequently aimed by companies.

Apart from the scale and complexity of software systems, it is relevant to mention that code and tests are also frequently changed/added/removed due to new functional requirements, maintenance operations and even technological evolutions. Indeed, regular updates to code and tests, and complex dependencies between different distributed software components, dramatically increase the difficulty to know (and then select) which tests should be executed due to the changes that were done. This becomes even more relevant considering that modern software development methodologies include continuous integration practices, which demand frequent development cycles and a fast feedback on the impact of changes. This is nowadays a common and context independent scenario, since the industry is suffering heavily from this problem.

Continuous integration is a software development practice where members of a team integrate their work in a frequent way, at least once per day [DMG07]. Under a continuous integration model developers need to get fast feedback, ideally within the minutes time scale, every time they make a change in the code to check whether that change did not introduce a bug caught by tests (existing tests or newly created tests).

This dissertation elaborates on the subject of optimizing the software development feedback loop time, which is the time that a software developer has to wait before getting feedback about his latest changes to the source code of the software being developed (Quality Assurance (QA) feedback, specifically). This is crucial in order to provide software developers with a fast but at the same time valuable feedback about the impacts their changes have in the software behavior, and therefore to enhance developers productivity and achieving more frequent software releases.

In order to improve the feedback loop time a test case selection technique is proposed. The proposed technique is able to realize the static and dynamic dependencies between source code and test cases, so that the pieces of source code that are executed by each test case are always known. So, when the source code base is changed, the goal is to select for execution

only those test cases that exercise the changed code.

Furthermore, the proposed technique was implemented in a real software development environment. This environment also 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.

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[1], an international software house with its Research & Development group based in Portugal. Specifically, all the work that is related with this dissertation was conducted at OutSystems Research & Development group in the R&D Productivity Team, and was supported with a scholarship sponsored by OutSystems.

OutSystems develops a platform, which is called the OutSystems Platform, that is in fact a piece of software that is highly complex, of distributed nature, with a large test suite, and that has evolved through many years. This is, therefore, a scenario that fits the conditions mentioned above in nowadays general software systems. This renders OutSystems as a perfect context to solve the problem of fast feedback loops and, better than that, a perfect context to test the proposed solution that will be described in a real industrial software development context.

In OutSystems, Agile software development methodologies are used and in this context fast feedback is a cornerstone. Furthermore, continuous integration is also a reality in many Agile software development contexts and this demands practices such as software developers integrating their work in a frequent way. This continuous integration implies necessarily that a developer has a feedback that is both valuable and fast. Therefore, achieving a fast feedback loop is the main goal of this dissertation.

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.

In fact, industry often deals with scenarios where both software code and automated test cases are being developed over the years and by many different developers, many of which might not be available these days. Furthermore, 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 a test subset: even if the rationale for the creation of a particular test is still clear (which often is not), the number of tests in the suite invalidates any accurate manual or empirical selection. If this was the practice, important tests could be left out, which would not provide a valuable

[1] http://www.outsystems.com

feedback and entail great risks. So, the really valuable and effective feedback is only obtainable by executing the entire test suite, unless there is an automated mechanism to effectively select relevant test cases based on rational and proven techniques (which is the focus of the presented work).

In fact, and as mentioned before, there are already several papers and articles published coming from both academic and industrial research that aim to solve the problem of selecting which test cases shall be executed due to a set of code changes. The great majority of those approaches are either focused on standalone or simple programs without complex interactions or require the programmer to manually define dependency information. Furthermore, even the few that aim to scale to large systems use fabricated validation scenarios or a very few number of validation scenarios, and usually with some restrictions.

The main motivation for this work is, therefore, to propose an automated test case selection technique that is designed and able to select test cases related to code changes. Where this work brings innovation is in the fact that the proposed test selection technique a) aims to work on highly complex, distributed and dynamic systems and b) does not require any help or hint from the developers that create the code and tests. Furthermore, another goal is to demonstrate that the test selection technique proposed on this dissertation is able to work on real and unconstrained scenarios, presenting a significant number of validations using real data (real software and real test cases).

1.2 Problem Statement and Goals

The context and motivation described above clearly states that the software industry in general is heavily suffering from the fact that fast feedback and release loops are not easy to achieve in large systems that are being developed for several years. The code and test suites that intend to test that code grow with time, and software maintainability tends to become harder and implying more costs. Besides this, technical debt [SSS15] tends also to dramatically increase with time.

Therefore, it is important to develop techniques that aim to achieve fast feedback. The ability to select which test cases shall be executed due to some specific code changes is crucial towards this fast feedback, 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 selection, might be defined as follows:

Problem Statement. *(Test Case Selection Problem)*
Given: A program, μ, a modified version of μ, μ', and a test suite, τ, that was designed to test μ.

Problem: Find the smallest subset of τ, τ', with which to test μ', such that τ' includes all the tests that exercise the differences between μ' and μ.

This statement can be considered a simple adaptation of the test case selection problem as originally proposed in [RH96a].

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

R1) It must be suitable to a multi-process and multi-program environment, where there are several applications and services that interact with each other.

R2) It must be suitable to automated tests that execute code that crosses the boundaries of several processes.

R3) It must be suitable to be applied to an existing large system (code and tests) without the need to change its code and tests.

R4) It must scale to large and continuously growing systems.

R5) It needs to be efficient in what concerns execution performance in practice, which will allow its usage on a continuous integration environment.

R6) It must deal with a constantly changing code base and test suite.

R7) It must apply to real environments, within real, unconstrained, validation scenarios.

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. First, the described problem and existing related work were studied, and only then a solution was proposed, implemented, and validated.

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. Also, this step is crucial for preparing the research work that has to be done next.
2. Formulation and characterization of the problem that is to be solved during the work.
3. Survey on the academic and industrial publications that exist so far, so that a concise perspective of the fast feedback loop problem might be elaborated.
4. Proposal of a solution to the stated problem in an abstract and generic way, without any requirements that are too dependent to a specific industrial context.

5. Implementation of the proposed solution in a practice and in the context of a real software development context.

6. Evaluation of the behavior of the proposed and implemented solution, specifically demonstrating the theoretical feedback time reductions and measuring actual feedback time reductions, using real and not fabricated nor chosen scenarios from daily software developer activities.

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 on this section, yet without a very detailed description, which will be made on Chapter 4.

The OutSystems Platform is a complex software of distributed nature, and therefore it has several components that interact with each other. This means that the platform as a whole, while executing, has several system processes executing at the same time and communicating with each other.

In order to select in an automated way which test cases should be executed when a change in the source code is made, there must be some kind of relationship or dependency information between the source code of the platform and the test cases that are designed to test that same platform.

This context renders a solution that only relies on static code dependency analysis unsuitable to understand all these complex interactions, since this is not a context where a traditional self-contained system exists and enables that analysis to understand the full possible execution flows.

Then, recording the execution paths while tests are executing is also relevant, specifically because this kind of runtime dependency analysis has the ability to record the execution path of the platform between different processes and, therefore, between different platform components.

Furthermore, runtime dependency analysis records the actual execution path, while static dependency analysis produces as its dependency analysis all the possible execution paths that a program might follow when triggered by the execution of a specific test case. This renders the runtime dependency analysis more accurate. On the other hand, static dependency analysis has the ability to better deal with the introduction of new test cases or structural changes in the source code.

The solution that is proposed on this dissertation to solve the problem of fast feedback loops is a solution that relies on selecting a subset of the full test suite that shall be executed upon specific changes of code, tests, or both code and tests.

So, this means that, by using and combining both static dependencies and runtime dependencies of previous test executions between test cases and the source code, and by relating those dependencies with changes that were made in the source code, this work proposes a methodology for selecting test cases that are specifically related with the changes that are made in the source code and tests.

Both static and runtime dependency information are computed incrementally, which means that only changed parts of the source code or tests are computed on each execution of the proposed methodology.

In Figure 1.1 an overview of the proposed test selection methodology is illustrated, using a scenario where changes were performed to the previous code revision, originating a new code revision (the current one) that needs to be tested.

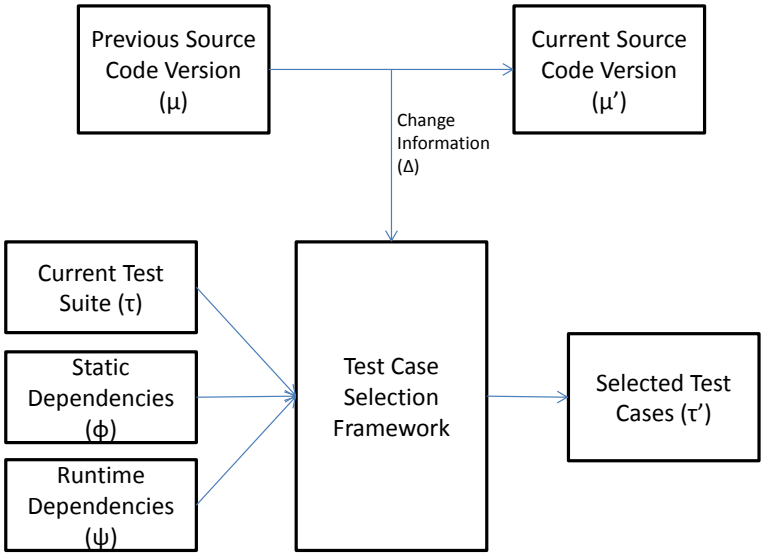

Figure 1.1: Test Selection Methodology Overview

The proposed solution to the problem of fast feedback has the ability to scale, is efficient and is safe in the sense that it revealed the ability to select the correct test cases to 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. Then, the main contribution of this dissertation is the following:

- Proposal of a test case's automated selection methodology, which is based on both static and runtime dependency information between test cases and software code (and between software code itself). The proposed methodology consists on identifying the changes between two different source code versions and on the ability of relating those changes with test cases. More specifically, the presented methodology relies on a static analysis of the dependencies between test cases and code, and also on runtime analysis of dependencies

between test cases and code (by recording the actual test case code execution paths), without requiring an explicit input from the programmer.

After having the source code changes information and test/code dependencies, the proposed methodology is then able to select a subset of the full test suite that, when related with the changes made, has the same coverage and fault-detection capability as executing the full test suite. This approach is suitable not only for regression testing, but also for continuous testing with continuous introduction of newly created test cases.

In order to achieve the main contribution of this work, previous contextualization and research were conducted and they led to the following contributions:

- 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.
- Wide survey on the subject of feedback loop optimizations and test optimizations, presenting related works to these main topics and the different approaches that were presented until these days, also with a critic and constructive perspective of those approaches' advantages and weaknesses.

As a by-product of the main contribution, two other relevant contributions emerged from this work:

- Proposal of an automated validation methodology that uses data from real software development scenarios.
- Elaboration of a scientific paper that has been submitted to the Software Engineering In Practice (SEIP) Track of the International Conference on Software Engineering (ICSE).

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 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 fast feedback, which is a test case selection methodology that uses information from both static and runtime code analysis. 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 and not selected nor fabricated data. Then, validation results are presented with several relevant measures, along with actual feedback time reductions.
- **Chapter 7** includes some final remarks and summarization of 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 the presented 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 fast feedback, define a specific field of intervention and set the relevant research areas that will be further studied in Chapter 3.

2.1 Introduction

The challenge of being able to release software on a frequent pace has become crucial for most software development companies. Indeed, only with fast release cycles it is possible to satisfy the continuously evolving needs of customers: this permanent evolution is in itself a consequence of the dynamic nature of modern businesses, and a nature that software needs to support.

While the need for practices, methodologies and tools that speed up the software development life cycle can also be witnessed at OutSystems, this is by no means a need that is specific of that company. In the particular case of OutSystems, this need led to the creation of a Productivity Engineering Team, which focuses precisely on continuously improving tasks that relate to software development, and therefore aim at increasing the productivity of developers.

In this chapter, the industrial context of OutSystems is described. While this context is particular of that company, it is believed that the characteristics that are relevant for this dissertation are common to most software development companies that have adopted modern development methodologies with continuous integration. 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, and also its technological setting. Furthermore, some detail on its infrastructure is given, together with the status of the current most relevant challenges it faces.

2.2 The Company and its main Product

OutSystems is an international software house that develops a specific product, which is the OutSystems Platform. This software is provided as a Platform as a Service (PaaS) solution that provides customers with an IDE which relies on designing 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 hundreds or thousands of users.

Applications developed with the OutSystems Platform allow customers to immediately leverage the benefits of visually generating (*e.g.*, using drag and drop capabilities) standard and optimized *Java* or *C#* code, developing only once for all devices (cross platform), and with different stack combinations (different combinations of application servers, operating systems, and databases).

Figure 2.1 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.

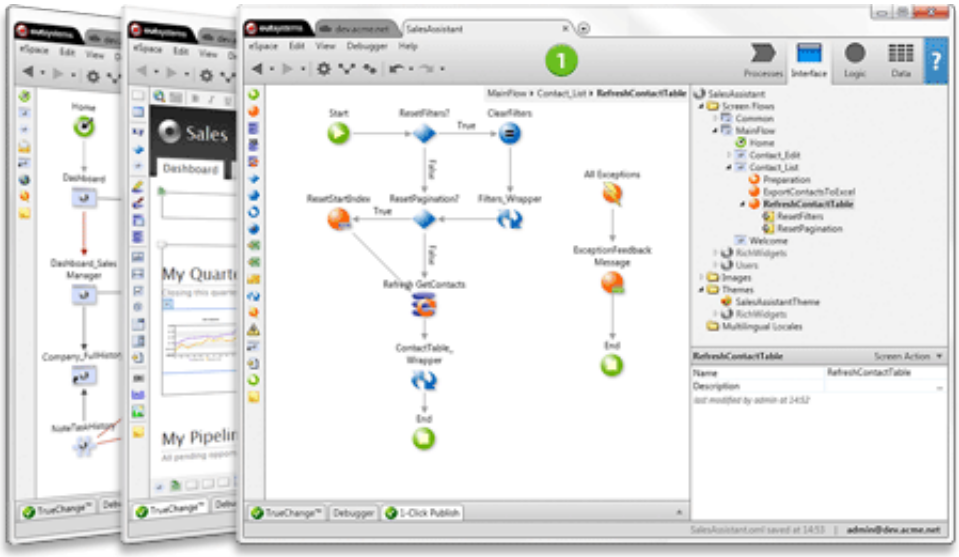

Figure 2.1: The OutSystems Platform - IDE Samples

The application lifecycle management allows users to have a full control over deployed applications, including monitoring and troubleshooting capabilities, and also managing applications across different environments.

In what concerns the platform architecture, its components can be divided in two big groups - the Development Environment (DE) and the Platform Server (PS).

The DE is composed of two major components - Service Studio and Integration Studio. The first one is the desktop environment already illustrated in Figure 2.1, and that allows customers to use visual models in order to create mobile and web applications, that are then compiled and deployed to several devices at once by the PS. The second one is also a desktop environment that allows users to integrate external libraries, services and databases in developed applications, extending the OutSystems Platform with additional functionality.

The PS 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. Also, it has two main browser based tools - Service Center and LifeTime - from where customers can manage, monitor and troubleshoot the OutSystems Platform application's portfolio, as well as managing it across different environments, define security polices and manage the complete lifecycle of applications,

from development to production.

In Figure 2.2 an overview of the OutSystems Platform components and their interactions is presented.

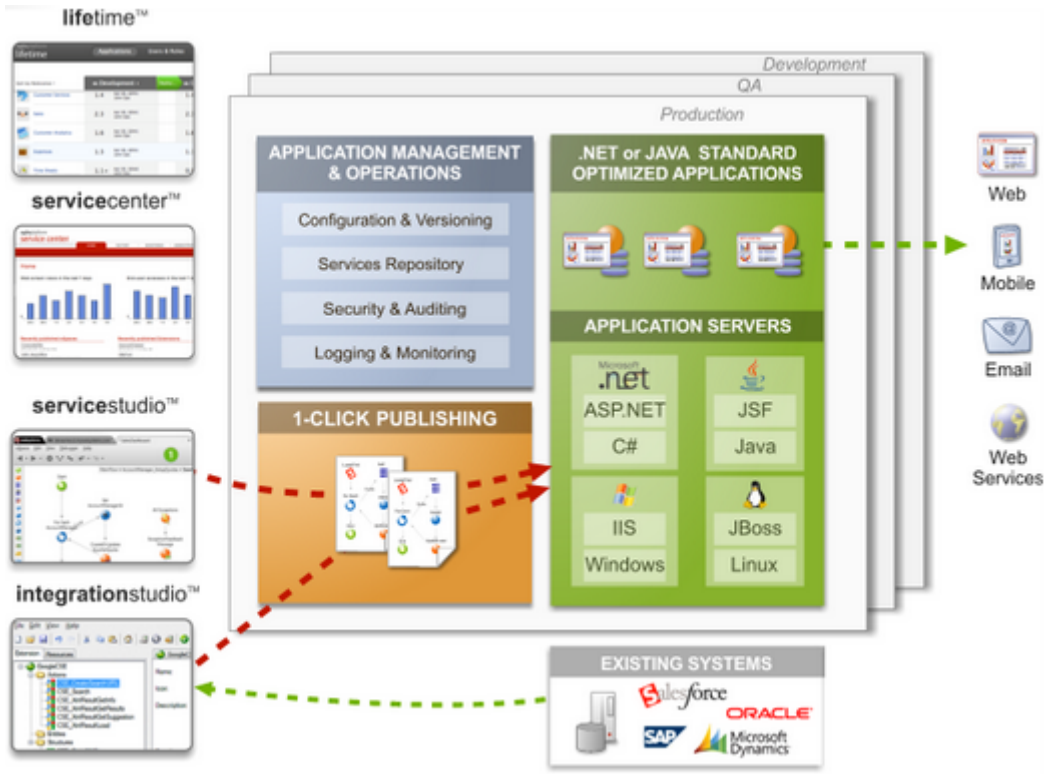

Figure 2.2: The Components and Interactions of the OutSystems Platform

In order to provide more detail about the platform's architecture and complexity, in Figure 2.3 an overview of the platform architecture is presented.

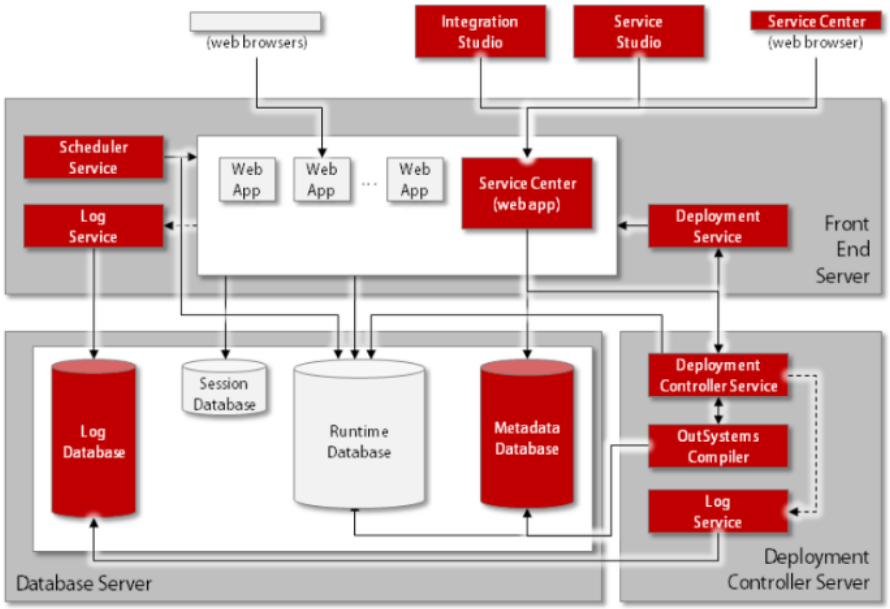

Figure 2.3: The OutSystems Platform Architecture Overview

It is concluded from the presented description and architecture diagram that the OutSystems Platform is indeed a large and very complex piece of software, with different components, developed in different technologies and designed to work on top of different technologies, and all interacting with each other. In the next sections more details will be given in what concerns the platform development and its actual feedback flow.

2.3 The Development Team, Methodology and Paradigms

OutSystems has dozens of collaborators organized in different teams working actively and exclusively on the development of the OutSystems Platform on a daily basis. This means that dozens of collaborators are developing new code, changing existing code and creating test cases daily, several times each day. Collaborators are spread among several teams that have specific tasks and specific goals, and usually each of those teams works on more specific parts of the Platform's code.

The code that contributes to a given release is therefore developed by several different project teams. In some releases, each project team works on its own code branch that is periodically reintegrated to the common release branch. In other releases, several teams work directly on the release branch. Whatever is the case, all test cases are executed so that the software developers might 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. In fact, in a book that compares two different flavors of Agile software development [Kni10], Kniberg concludes that *"generally speaking you want as short a feedback loop as possible, so you can adapt your process quickly"*.

Also, continuous integration is an interesting practice in the context of Agile software development, and its foundations help to reach the already mentioned fast feedback loops and fast integration and release cycles. Currently, there are efforts being conducted in order to achieve really fast continuous integration environments where developers can be notified very soon about the impact of their changes in the code, and this is precisely where this work becomes necessary. In Section 2.4 more details will be given in what concerns the current feedback flow of OutSystems, how is the platform developed in general, how software quality is achieved and where possible optimization points are found.

2.4 Platform Development, Feedback and Quality Assurance

The OutSystems Platform's code is in its majority written in *.NET*, specifically using the *C#* programming language. Some other languages are also used, such as *JavaScript* and *TypeScript*. These languages are used to develop different modules that make part of the OutSystems Platform, such as desktop environments and web services. Since the platform is available for both *.NET* and *Java*, an internal tool is also responsible for translating the *.NET* code to *Java* (including the translation of test code).

In what concerns the current feedback loop, after a developer performs changes in the code of the platform, there are three main steps to assure that the platform is still behaving as expected: a) build the platform code, b) install the platform code in different environments with different configurations and c) execute a battery of automated tests. This loop will provide a constant feedback about the impact of changes in the platform to the developer. Figure 2.4 contains a visual representation of this feedback loop.

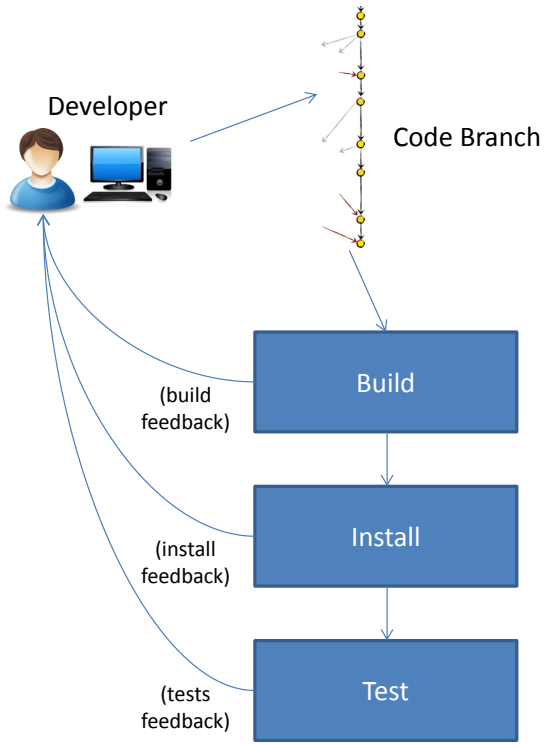

Figure 2.4: OutSystems Development Feedback Loop

Describing those steps more specifically, the build stage builds the platform code for a given code revision of a given branch, producing the build artifacts that are necessary for the platform to be installed. This stage itself provides feedback already, since if the build fails the developers get immediate feedback of the cause of failure.

Then, and since the OutSystems Platform supports a lot of different stack combinations, the platform's built code is installed in a myriad of testing machines with different configurations. Also, if the installation fails the developers get the information that something went wrong.

If both build and install steps are successful, then a battery of automated tests is executed in every testing machine where the platform was installed. This is the step where a full battery of automated tests that include unit, integration and system tests are executed in order to guarantee a stable state of the Platform's code.

To perform all these feedback loop stages, developers also have access to a tool that allows them, among other things, to a) request builds for a given branch, b) request the installation of a given build in a myriad of testing machines with different configurations and c) request the execution of tests on the testing machines and monitor the results of that execution (check which tests are in queue, which succeeded, which failed).

The OutSystems Platform is being developed for several years already, and therefore its code base and test suite have grown with time. At the moment, OutSystems test suite has more than 10.000 tests being executed to validate its software and all of these tests are automated.

Executing all the automated unit, integration and system tests currently takes several hours. This is, in fact, the biggest motivation for this work: being able to have automated systems that reduce the feedback loops even when software is large and complex, and guaranteeing that is has been as thoroughly tested as possible.

2.5 Specific Problem Formulation

In Chapter 1, and specifically in Section 1.2, the generic problem was already defined. Considering the context of the current chapter, and specifically in Section 2.4, the problem might be defined in a more concise and specific way. So, what is desired to be achieved is a solution to the problem described below.

Specific Problem Statement. *(Tests Feedback Loop Optimization)*
Improve the way that tests are orchestrated so that software developers might be notified as soon as possible about the results of tests executions. This fast feedback is to be obtained by only executing all tests in the suite when this is strictly necessary.

Specifically, define a methodology that is able to relate specific test cases with specific code changes, so that a partial execution of the full test suite might suffice to guarantee software quality.

2.6 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 software quality is guaranteed.

Understanding this context was crucial to understand where feedback loop optimizations could take place, and it became clear that optimizations should emerge from one of the QA steps, which is test cases execution.

For a developer to benefit from feedback it must be fast, which means that test cases execution must be fast enough for a developer to get feedback as soon as possible.

The description of the OutSystems context that was made in this chapter also helps to clarify why this research has focused on optimizing test orchestration and feedback. This is the main reason why this chapter comes before the State of the Art (Chapter 3), 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 fast feedback loops. To be more concise, research areas that are related with optimizing the time that test suites take to execute are interesting for the context of the problem that is being solved. It is of a great interest to understand what has been done so far in both academic and industrial works in order to optimize software tests execution and orchestration.

3.1 Introduction

In software engineering, the concept of feedback loop can be defined as a sequence of steps of the development process that permit continuous software corrections and improvements. Executing a battery of tests that intend to test the software code logic, integrations between different components, and even simulate software behavior as a whole is a very important step in this loop.

Improving the feedback loop time is very important to improve the overall productivity of software development. Indeed, the faster the feedback loop, the higher the number of corrections and improvements that are possible per time unit.

The problem of achieving fast software development feedback loops has been extensively studied in the past. In fact, several works that can be found in the literature have focused in solving this problem.

These works have been developed both in academic and industrial contexts, and they might be divided in several categories in what concerns how they approach the problem. In fact, there are different approaches that focus on different parts of the software development steps. Some approaches are focused on infrastructure improvements while other approaches are focused on tests improvement.

The different main topics found during this research work that address the final goal of achieving fast feedback loops might be divided in the following categories:

- Infrastructure improvements and Test Distribution, which are described in section 3.2;
- Test suite optimization techniques (test suite minimization, selection and prioritization), which are described in section 3.3.

3.2 Infrastructure Improvements and Test Distribution

Several works in the area of fast feedback involve infrastructure improvements or a better usage of available infrastructure resources.

In these lines, one natural infrastructure improvement is achieved by increasing the resources that are available for processing tests, which means, increasing machine power available (by increasing the number of existing machines or their hardware capabilities). This solution is able of improving the feedback loop time until a bottleneck is reached, which occurs when the available budget ends. Furthermore, as software and test suites grow, it is simply not affordable to continuously invest on hardware or machines.

Another approach that is related with infrastructure is allowing a better usage of the available infrastructure resources, and this is a field of research that is related with test distribution. This means, in practice, using techniques such as parallelization to execute a battery of tests. If the ability to split tests across different machines exists, or the ability to run several tests at the same time in the same machine, the overall time spent in executing a full test suite will be reduced.

Nowadays it is easier and cheaper to allocate computational resources than it was some years ago, mainly due to the proliferation of Virtualization [VMW], Grid [FK03] and Cloud [Wei07] technologies. This means that tests distribution does not imply as many costs and effort as it did some years ago.

During the year of 2001, Kapfhammer published a description of the conceptual foundations of tests distribution, along with a design and implementation of an approach that is able to distribute the execution of test suites across multiple machines [Kap01]. This idea relies on the principle that the workload needed to process a full test suite might be split across different machines, which will reduce time costs of testing activities, depending on the number of machines available. This specific work included the proposal of a tool, Joshua, that is an extension of the well known JUnit testing framework.

Later, a framework called Metronome was introduced for grid environments, prividing automated software build and test services for multi-platform systems that are based on grid technologies [PCG+06].

Yet another framework was developed, which is called GridUnit. This is a framework for distributing the execution of JUnit-based test suites [DCBM06].

In the past, some work was already conducted at OutSystems regarding tests distribution, parallelization and capabilities of executing several test cases at the same time. Therefore, this will not be the way to go on this dissertation work. Furthermore, tests distribution will not solve the problem of always executing the full test suite, it is too dependent on hardware resources and system growth will always imply more costs and human effort.

3.3 Test Suite Optimization Techniques

Several test suite optimization techniques were proposed in order to achieve more efficient feedback loops, and specifically to optimize the time in which test cases execute or optimize their fault-revealing capabilities. Some of the main aspects that are crucial for any of those

techniques to be suitable in practice are its efficiency and safety. Efficiency is the ability of the techniques to save significant costs when compared to a full test suite execution. Safety is the ability of the techniques to guarantee that all test cases that are related to code modifications are executed, and that no tests that are relevant are excluded from the test suite.

In this section three techniques and works related with tese techniques will be presented: test suite minimization, test case prioritization and test case selection.

3.3.1 Test Suite Minimization

Test suite minimization is a technique that aims to remove redundant test cases from a test suite. There are several definitions of redundancy, but in the techniques that are presented next a test case is redundant if a) it is no longer meaningful for the program under test; b) it has the exact same testing coverage than any other test case.

The concept behind test suite minimization is the idea of permanently removing test cases that are considered redundant.

Harrold *et al.* propose an approach to address the test suite minimization problem that aims to identify redundant and obsolete test cases based on tests relationship with testing requirements [HGS93]. These authors define the problem of test suite minimization assuming that each test case has a relationship with some kind of requirement - software requirement or test coverage requirement, as follows [HGS93]:

Definition 1. *(Test Suite Minimization Problem)*
Given: A test suite T, a specific set of testing requirements r_1,...,r_n that must be satisfied in order to provide the desired testing coverage of the program under test, and subsets of T, T_1,...,T_n, each one associated with each of the r_i's such that any one of the test cases t_j that belong to T_i can be used to test r_i's.

Problem: Find a representative set, T', of test cases from T that satisfies all r_i's.

Tests are classified as obsolete if they are not testing any test requirement anymore. A specific test is classified as redundant if there is any other test that tests the same testing requirement as the first one. This approach requires a strong and regularly updated set of testing requirements as well as a mapping between test requirements and test cases. Empirical studies were carried on a very small test suite (around 20 tests).

Offut *et al.* approach the problem as a minimal set cover problem [OPV95], using mutation testing. Mutation testing is a testing technique that involves modifying a program in specific small parts. Each of these modifications is called a mutant. Test cases shall detect these mutations, and if they do, this is called *killing* the mutants. The authors apply mutation testing techniques and then identify the minimal set that covers all the mutants (which means, that is able to *kill* all the mutants). Tests are first executed in different orders and the minimal set that obtains the maximum mutation score is selected. Empirical studies were made with a set of around 40 tests and 48 lines of code, reducing sizes of test suites by over 30%.

Chen *et al.* propose an approach that also assumes from the beginning a correspondence between each test and a set of test requirements [CL96]. A test suite is defined, in this context, as a set of test cases that can collectively satisfy the whole set of test requirements. The authors also define the concept of an essential case test, which is a test that will make a requirement uncovered it if removed. In the proposed approach the authors start by selecting all essential test cases and then make use of greedy algorithms to select further test cases that satisfy the maximum number of unsatisfied test requirements, until all the requirements are satisfied. On an alternative approach, they start by removing redundant test cases and then apply the first mentioned heuristic. The authors only present the results of simulations, but not any empirical results on real data.

Marré and Bertolino published a technique for coverage testing using spanning sets [MB03]. Their work is not specifically focused on test suite minimization, but it might be used as a direction to achieve that minimization, since tests might be selected (or removed) according to a specific coverage criteria. This approach is an alternative to the already existent approaches focused on requirements, focusing instead on representing the structure of the system under test using a decision-to-decision graph (ddgraph), which has a single entry and a single exit point. Marré and Bertolino's work is focused in unit tests. The authors mention, according to empirical studies they performed, that their technique is not totally effective, although it is more accurate than some previously existent approaches. The proposed approach is therefore based in finding a minimal set of tests that traverse all possible paths of the ddgraph. This still a weak assumption since ddgraphs alone are not enough to guarantee safety.

Tallam and Gupta presented an approach with a new greedy heuristic that aims to select a minimal subset of a test suite that covers the testing requirements that are also covered by the original suite [TG05]. This approach improves some redundancy present on previously presented greedy heuristics, which also improves the reduction efficiency.

Black *et al.* extended extended previous work of Chen *et al.* [CL96] by combining two testing criteria [BMK04], and both taken in account for test selection. This can be referred as a bi-criteria approach that, in this case, takes in account def-use (definition and usage) coverage criterion and fault-detection history for each test case.

Jeffrey and Gupta also extended previous work of Chen *et al.* [CL96] by applying a second set of test requirements (another criterion) after the first (but not simultaneously) [JG07], including then some tests that were before marked as redundant but that are not redundant for this second set of test requirements. This helped to improve the fault-detection capabilities of the test suite, making it larger on the other hand. The authors call this technique a selective redundancy technique. It is relevant to mention that a test case is only analyzed with the second criterion when it is marked as being redundant by the first criterion.

Hsu and Orso developed a framework (MINTS) for multi-criteria test suite minimization based on user input (the user assigns a priority to each of the given criteria) [HO09]. The authors mention possible criteria such as coverage data, execution cost and fault-detection capabilities (or expectancy). Then, an Integer Linear Programming (ILP) formulation is made in order to solve the addressed problem, with an weighted-sum approach, prioritized optimization or an hybrid approach, using defined criteria as constraints to the ILP formulation. This approach proved

to be more effective than previous single-criterion approaches in what concerns reducing the cardinality of a test suite (especially when it is large), but it is still not able to be considered as a safe technique.

McMaster and Memon propose an approach based on call-stack coverage generated by the test suite [MM05, MM08], which is a dynamic approach (since call-stack information is gathered during runtime) and has a wider applicability than other techniques, because it is language independent. Anyway, although the authors claim that their approach is able to produce favorable trade-offs between the reduction in test suite size and reduction in fault detection effectiveness, the technique is still not a safe technique.

Harder *et al.* presented an approach to test suite minimization using operational abstraction [HME03], which is a different approach from the ones already presented until now that share the fact of being based on some kind of coverage criteria. Operational abstraction is a formal mathematical description of a program's behavior, which has scaling costs that are hard to predict. This operational abstraction is obtained from program executions, so it is a dynamic approach. The basic idea is constructing a new test suite from scratch, where test cases are executed (in some order) and they are added to the test suite if they improve the existent operational abstraction. If they do not change anything in the operational abstraction, they are not added to the test suite. Still, as happens with many minimization techniques, this technique does not ensure that a removed test case was not relevant to detect a specific fault that other tests are not able to detect. Still, empirical studies demonstrated that more faults are detected when compared to branch coverage-based approaches. Also, this work can be used as a coverage criterion for previous approaches.

Schroeder and Korel propose an approach that is focused on black-box testing [SK00]. The authors identify possible redundancies by verifying the impact that test inputs have in the outputs of the program being tested. Test inputs that produce the same outputs might have redundancies, so it does not make sense, in the authors' opinion, to execute them all. This approach makes sense in a context where the system under test is composed of unmanaged code, which means, when the code of the system under test is not available.

Vaysburg *et al.* propose an approach that makes use of Extended Finite State Machine (EFSM) system models [VTK02], capturing dependencies (static and dynamic) between EFSM elements in order to reduce test suite by identifying tests that exhibit the same pattern of interactions. Anyway, this approach assumes that tests are requirement-based and there is a model-based system representation. Korel *et al.* and Chen *et al.* also proposed some extensions to this approach [KTV02, CPU07], with automatic identification of changes in the model and a more complex model of changes, respectively.

Kaminski and Ammann propose an approach using a logic criterion to reduce test suites cardinality [KA09]. Furthermore, this approach also provides guarantees in what concerns fault detection capabilities in testing predicates over Boolean variables. Still, this approach assumes that there is a formal description of a program.

In general, test suite minimization techniques aim to reduce the size of a test suite, but those techniques represent a one time improvement. Large test suites that are often found in large

and complex systems, even when minimized, will still be too slow to solve the problem of fast feedback. Furthermore, redundancy is a very vague definition in the majority of these techniques.

3.3.2 Test Case Prioritization

Test case prioritization is a technique that aims to prioritize some test cases of a test suite in what concerns their execution order. The main goal behind test case prioritization is the ability to detect faults as soon as possible. In the optimal scenario, all the test cases that have a greater probability of revealing failures shall be executed first. This means that test cases that have a higher probability of finding software defects shall be executed first. On the other hand, test cases that have a lower probability of finding defects according to the latest code changes shall be executed at the end.

Anyway, test case prioritization techniques might also be used to achieve other goals than early fault detection, although this one is the most common. Several heuristics are generally used to prioritize test cases execution, and the used heuristics are the main difference vector between different proposed techniques.

Rothermel and Harrold formalized the definition of test case prioritization as follows:

Definition 2. *(Test Case Prioritization Problem)*
Given: A test suite T, the set of permutations of T, PT, and a function from PT to real numbers, f : PT $\rightarrow \mathbb{R}$.

Problem: Find T' such that $\forall$ *T" ∈ PT, T" ≠ T'* $\implies$ *f(T') ≥ f(T").*

Rothermel *et al.* published a work where their approach is to test different heuristics for test case prioritization, evaluating the impact that each of those has in early software fault detection [RUC01]. Specifically, the approach proposed by the authors takes in account nine different test case prioritization techniques that vary on what concerns the used heuristics.

The base scenario is executing tests with no prioritization at all, that serves as a baseline for reference. Then, the authors performed some experiences using the following ordering criteria:

- Randomized ordering,
- rate of fault detection optimization ordering,
- early statement coverage ordering,
- early statement coverage ordering taking in account previous tests statement coverage,
- early branch coverage ordering,
- early branch coverage ordering taking in account previous tests branch coverage,
- early fault detection probability ordering, or
- early fault detection probability ordering taking in account previous tests.

The authors then performed some experiences to evaluate what heuristics would perform better and in what situations. They came to the conclusion that optimizing the rate of fault detection of tests would be the best scenario in the great majority of the conducted tests. Furthermore, the authors also mentioned that in some situations random ordering of test cases significantly improved the early fault detection of the test suite.

Rothermel and Kinneer also performed some experiences with the *JUnit* unit testing framework [DRK04]. One of the main conclusions of this work is that it is difficult to know which tests expose a fault, and therefore prioritization techniques will depend on surrogates and heuristics to prioritize test cases.

Leon and Podgurski presented an empirical comparison of three different test suite prioritization techniques [LP03]. Specifically, these were additional coverage prioritization, cluster filtering with one-per-cluster sampling and cluster filtering with failure pursuit sampling. Additional coverage prioritization is based on the idea that tests shall be prioritized by an order in which they maximize code coverage as early as possible. Cluster filtering uses groups of profiles of tests with specific characteristics and picks one or more tests from each group first. The presented paper concluded that these distribution-based prioritization techniques usually have better results than additional coverage prioritization.

Kim and Porter proposed an approach that used history-based heuristics to prioritize test cases [KP02]. The main idea behind this prioritization approach is that failure history of test cases is available, in order to prioritize test cases that failed more times or more recently.

Mirarab and Tahvildari proposed an approach to test case prioritization that is based on probability theory and Bayesian Networks [MT07]. Source code changes, software fault-proneness and test coverage data are incorporated into a unified model so that a probabilistic analysis might be performed to prioritize test cases.

In general, test case prioritization techniques aim to maximize early fault detections, but they do not have any ability of reducing the test suite size. Even if failures are detected sooner, the full test suite still needs to be executed. Furthermore, since test case prioritization techniques are based on heuristics, there is not any guarantee that a fault revealing test will execute before all the others.

3.3.3 Test Case Selection

Many researches have already been conducted in order to study the problem of test case selection, proposing several approaches for different types of software and system scales. Some of those research works stated the problem formally and started to define some solutions, the great majority of those for procedural programs [RH96a].

Rothermel and Harrold formally defined the problem of test case selection [RH96a], as follows:

Definition 3. *(Test Case Selection Problem)*
Given: A program, μ, the modified version of μ, μ', and a test suite, τ.

Problem: Find a subset of τ, τ', with which to test μ', such that all the tests that exercise μ' are selected.

Although the presented work is related to modern and large multi-modular and distributed systems, some older research works propose some techniques that served as an inspiration, thus some of those deserve a reference in this section.

First of all, almost all the proposed test case selection techniques consist on two major steps: identifying the affected parts of a program after changes and selecting test cases that have the potential to detect errors detected on account of that changes [BMSS11].

Several authors formulate the problem of test case selection as an ILP problem. Fischer *et al*. propose an approach for procedural languages, assuming sequential execution of block statements. Their technique also relies on matrices that represent the relationship between test cases and program segments [Fis77, Fis81]. The proposed work is aimed at procedural languages and is dependent on the control-flow structure of the software under test, which renders this technique unsuitable for the presented context and for today's software systems in general.

Harrold and Soffa propose an approach that is based on data-flow analysis and using program slicing, representing a program as a graph and using definition-use pairs to select test cases that exercise changed definition-use pairs [Har88]. This technique is not sensible to changes that are not related to data-flow changes, which renders it as a weak and unsafe technique.

Yau and Kishmoto propose an approach with symbolic program execution, using constraint solvers to optimize the problem. Changes are detected by analyzing control-flow graphs that represent a program [Yau87]. This approach assumes that test cases are designed to cover different input partitions and has an high algorithmic complexity, which makes this approach hard to scale for large test suites and large systems.

Some more techniques based on graph-walk approaches were proposed by different authors. Agrawal *et al*. propose an approach based on control-flow graphs and dynamic program slicing [AH90] with different slicing phases to reduce the execution complexity [AH93].

Rothermel, Harrold and other authors presented a set of different techniques based on graphs, starting with Control Dependence Graphs (CDGs) [RH93] that were not appropriated to capture inter-procedural dependencies. They also proposed Program Dependence Graphs (PDGs) that contain data dependency information, and also System Dependence Graphs (SDGs) that aimed to solve the problem of inter-procedural dependencies [RH94]. The same authors yet proposed a technique based on depth-first traversal of Control Flow Graphs (CFGs) [RH97], that they later extended to fit object oriented languages with the Intraprocedural Control Flow Graphs (ICFGs), which models object oriented languages specific features [RHD99]. Yet in the field of object oriented languages, also a technique with a graph specifically designed to work with *Java* language was proposed, called the Java Interclass Graph (JIG) [HJL+01]. This approach is yet too precise in what concerns graph representation of programs, and therefore it will have issues with scaling and program growth. Also, constructions such as late binding are excluded from the scope of this work.

Volkolos and Frankl propose an approach that is based on the textual difference between two different versions of the system under test [VF97]. Their approach is focused on *C* language, and they use preprocessing to remove redundant changes, such as blank spaces.

Chen *et al*. propose a technique that is modification-based, in the sense that the program is partitioned in different entities, and then test cases execution is monitored to create a relation to that entities [CRV94]. Entities are then compared, and those that changed between the original and the current version of the program will have the test cases that traverse them selected.

White and Leung proposed a different approach, but that has some similarities with this last one. They propose a technique which they call a firewall approach [WL92]. Basically, they split different program modules and treat them as atomic entities. Then, the technique checks for dependencies between different modules and creates a boundary (the firewall) around all modified modules and modules that interact with those that were modified. This approach is very conservative, thus, far from being optimal in what concerns precision. Kung *et al*. extended this approach for object-oriented languages [KGH$^+$93].

After this overview about general research works on the test case selection problem, some works have goals and scenarios that are more related or similar the work presented in this dissertation, which are next described.

Orso *et al*. propose an approach that aims to address the scalability problem of the graph-walk approach, and address that problem in large software systems [OSH04]. This approach is a two-phased technique, which assumes a first phase of partitioning and a second phase that is effectively the test selection, based on changes. The technique has been tested in different systems, real systems and some of them large systems, although test suite sizes are never over near 700 test cases. Their technique also assumes, for safety, that the code does not have reflection nor *late binding* and is very dependent on Java constructs.

Skoglund and Runeson applied the already mentioned firewall approach to a real large-scale distributed system in a bank in Sweden [SR05]. These authors also select modification traversing test cases, and they also use a graph based approach, but the firewall is created around classes and dependencies between different classes. This approach might be safe enough, although it is not precise. Though, the approach is efficient.

Buchgeher and Lusser proposed a technique for test case selection in a real software development environment, at Omicron Electronics GmbH. Their approach does not automate the process of selecting test cases, instead it helps developers select them with suggestions. This approach uses information of dependencies between test cases and code, system modifications and a graph representation of those dependencies, which is stored in a graph database [BERL13]. Then, developers are given a list of suggested test cases with which they might or might not agree, including more test cases or removing some that were suggested. Modifications are actually considered at file level, because the authors claim limitations of the version control system to get more information about code changes. In the work that is proposed on this dissertation this limitation was surpassed by syntactically analyzing the changed files to get the code meaning and being then able to identify methods and classes, among other needed constructs.

Elbaum *et al.* propose an approach that is designed to work in continuous integration development environments [ERP14]. But, instead of basing their technique on changes and relationships between test cases and code, the authors use some heuristics to select test cases. Test cases that failed recently, test cases that are not executed for some (specified) time or new test cases are then selected to execute first. This technique is weak in the sense that there is not any relationship between test cases and code, and test cases that reveal faults might not even be selected.

3.4 Conclusion

The current chapter presented a survey on several research areas that are related to the subject of fast feedback.

There are some works that focus on test distribution, which aim to enable the execution of several test cases at the same time. These approaches require significant costs in what concerns hardware infrastructure that will tend to grow with time. Furthermore, and this is often not the case, tests must be designed in a way that there is not any possibility that one test case execution might interfere with other test case's execution.

In what concerns test suite optimizations there were mentioned three main technique families: test suite minimization, test case prioritization and test case selection.

Test suite minimization aims to remove redundant test cases from a test suite. This kind of techniques is limited since redundancy is a very vague term in this context: if two test cases exercise the same code but with different inputs, are they redundant? If not, what are the criteria? This was unclear on the great majority of the presented techniques. Moreover, test suite minimization is a one time optimization and it will not decrease the execution time of a test suite that takes hours to the minutes scale.

Test suite prioritization techniques aim to change the order in which test cases execute. The main goal of this ordering is to execute first tests that, for some reason/heuristic, might have a greater potential to reveal software faults. This will not permit that a fast feedback is achieved, though. Although failures might be detected earlier, the full test suite shall be executed because heuristics are not able to guarantee the safety of these techniques when using only a subset of the full test suite.

Test case selection techniques aim to select a subset of the full test cases to be executed. This subset is generally computed using informations like the relationship between test cases and code. Test case selection techniques are interesting to the work that is being pursued in this dissertation because with these techniques a full test case execution is not required (not always), and in normal circumstances it would decrease the time spent in automated software test executions. The presented works on this field have, however, some bottlenecks that render them unsuitable to real scenarios. Specifically, the major bottlenecks found in these approaches were that these approaches were either:

- designed for standalone, self-contained and small systems without the characteristics that are found on modern software systems;
- unsafe even if applicable to large systems, without enough guarantees that the test cases that had the potential to reveal faults would be always selected; or
- developed for large systems but without promising validations nor fully automations (human intervention required).

Therefore, there is a clear need to improve these techniques in order to render them suitable in real scenarios, and it is precisely what this dissertation aims to to.

Chapter 4

Proposed Solution

This chapter presents the proposed solution (technique) to solve the problem of automated test case selection that was defined on Chapters 1 and 3, with the specificities of the problem in what concerns the characterization of 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 selection problem. It is also a test case selection problem whose solution must conform to the context of highly complex software systems, of distributed nature and developed with modern software development practices. These generic requirements arise from the need for a solution that is efficient, scalable and safe in the sense that the selected test cases always reveal the same failures as the full test suite, when related to code changes.

The distributed nature of many software systems and the usage of several different technologies require that the proposed test case selection 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 analyze the dependencies between test cases and software code. Apart from that, all the characteristics of the proposed test selection technique render that solution as suitable to a myriad of different software development contexts of large scale and complexity.

Regarding the related work that exists in both academic and industrial fields, described in Chapter 3, the present technique brings novelty when compared to the majority of those due to the fact that it is not designed with assumptions that are dependent to specific software development contexts. It was tested in unconstrained environments, on the contrary of the majority of the techniques presented before. The main goal of the current chapter is therefore to present the solution for the test case selection problem that aims to solve a real generic industrial problem and not only a problem that is specific of a particular company.

4.2 Test case selection technique

The test selection technique that is proposed in this chapter is a method that relies on dependencies between test cases and software code. The main goal of the proposed test case selection technique is to, for each set of code changes, select for execution only the test cases that might have been affected by code changes (or test cases that were newly introduced). This

selection is of great importance since executing all the test cases would lead to unnecessary spent time in test execution tasks and would be incompatible with a fast feedback.

When designing the proposed technique, main concerns were that it must be efficient and scalable as the software system grows, and safe (in the sense that all the test cases, from the full test set that would expose failures when executed, are always selected for execution). A test case selection technique with these characteristics is already a challenge by itself. But the target scenario of complex and distributed systems really increases the challenge to a new level.

In order to address the challenges mentioned in the last paragraph, the method proposed on this dissertation might be named as a *mixed static and runtime dependency based test selection technique*. The presented technique uses, indeed, information from both static and runtime dependency analysis to evaluate the relationships between test cases and software code, in order to select which test cases shall be executed due to some specific set of code changes.

The decision of evaluating dependencies between tests and software code statically and during their execution (runtime) was made because those two kinds of dependency analysis provide different data that is useful in order to achieve a safe test case selection technique. The static dependency analysis provides information about newly introduced tests and structural code changes, for instance. On the other hand, the runtime dependency analysis provides information about multi-process relationships that can't be obtained by static analysis (such as *late binding* code execution).

In what concerns the efficiency and scalability of the proposed test selection framework, the proposed technique performs an incremental static and runtime dependency analysis. This means that each iteration of the test selection technique only recomputes static dependency information if that information has changed, and runtime dependency information is only recomputed for the tests that are selected to execute. The incremental approach avoids unnecessary computation costs of dependency analysis between test cases and code. Furthermore, as a test suite and software code grow, the need to always calculate the whole dependency information would render any test case selection technique unable to scale efficiently.

Any technique or methodology that relies on incremental computations needs to compute the information from scratch at least once. This means that, before the proposed technique is executed, there is no static nor runtime dependency information available, so that information needs to be created. Furthermore, since runtime dependency analysis is performed, as its name suggests, during runtime, before the first execution of the proposed test selection technique the full test suite has to be executed in order to generate all the current dependencies between test cases and code. This initial scenario is named as the initial setup of the proposed test selection technique. Algorithm 1 presents a description of the required test selection technique setup.

Algorithm 1 Test Case Selection Setup Algorithm

Let: μ built code; τ full test suite; ϕ, ψ (test $\Rightarrow$ code) dependencies; ι execution textual information — be

1 $\phi \leftarrow$ Compute Static Dependencies (μ, τ);
2 $\iota \leftarrow$ Execute Tests(μ, τ);
3 $\psi \leftarrow$ Compute Runtime Dependencies(ι);

In Line 1 of the Algorithm 1, static dependency analysis is performed in order to generate the first static dependency information between test cases and software code (and code itself) and store that information in an appropriate data structure.

Line 2 corresponds to the execution of the full test suite. To perform this setup stage, each test execution is instrumented so that all the methods that it exercises during that same execution are recorded. On the presented technique, it is proposed that this code instrumentation is made in a non-intrusive manner. Obviously it is not expected that code instrumentation is included in production code, because that might for instance slow down software performance. As such, it is proposed that tests and code are only instrumented in a post-build step, in a way that the instrumentation is transparent to the developers and will not be included in the source code repository.

In the Line 3 textual information of tests execution, which contains for each test a list of exercised methods, is then organized and also stored in an appropriate data structure.

At this point, the setup of the proposed test selection technique is completed and the technique can now be used incrementally and cyclically.

From this point and forwards, the proposed technique relies on the ability to identify the changes that were performed to the software code and tests.

The usage scenario of the current test case selection technique is the generic scenario of software development: code/tests are changed, and stored to some code base (usually Version Control Systems (VCSs)). Test case selection aims to select test cases that shall execute after these changes, comparing the differences between the previous and the current versions of tests/code.

Algorithm 2 presents a description of the *mixed static and runtime dependency based test case selection technique*.

Algorithm 2 Test Case Selection Algorithm

Let:	μ, μ'		built code
	τ, τ', γ		test suites
	ϕ, ψ	be	(test $\Rightarrow$ code) dependencies
	Δ		a list of differences
	ι		execution textual information

Data: ϕ, ψ calculated in Algorithm 1

```
1  (μ, τ) ← Build Code and Tests();
2  while true do
3  |  (μ', τ') ← Build Code and Tests();
4  |  if ((μ ≠ μ') ∨ (τ ≠ τ')) then
5  |  |  Δ ← Identify Changes((μ, μ'), (τ, τ'));
6  |  |  φ ← Update Static Dependencies(φ, μ', τ', Δ);
7  |  |  γ ← Select Tests(φ, ψ, Δ);
8  |  |  ι ← Execute Tests(μ', γ);
9  |  |  ψ ← Update Runtime Dependencies(ι);
10 |  | (μ, τ)← (μ', τ');
11 |  end
12 end
```

The proposed algorithm is cyclic and iterative. Obviously, tests execution is required when changes are performed either in software code or test cases.

Each iteration of the proposed technique starts when a new code/tests version is available. There are many ways of identifying when there is a new available code/test version. For instance, and as will be detailed in Chapter 5, VCSs are widely used in the industry to store source code and source tests versions. The creation of new version control revisions might be used to trigger the execution of a test selection technique iteration. Also, specific schedules might also be a valid option. The main idea is that there must be a way to periodically build the code/tests and verify whether it has suffered any changes since the last build, which corresponds to the Lines 3 and 4 of the Algorithm 2.

If changes between the last and the current code versions (μ and μ', respectively) or changes between the last and the current test suite (τ and τ', respectively) exist, then it is necessary to compute the specific changes in order to co-relate them with the test cases (which is the step present in the Line 5 of the Algorithm). Different granularities can be used to describe the code changes. Those different granularities might be file level, class level (for object oriented languages), method level, or even statement level. Relating dependencies between test cases and source code at statement level will be a fine-grained approach, although it will be less efficient. On the contrary, relating dependencies between test cases and source code at file level will be a coarse-grained approach, but it will be more efficient. The idea is to find a good trade-off between efficiency and precision. At the end of this step, a list of changes (files, classes, methods or statements) must be produced, so that further analysis will be able to relate the changes with test cases.

After the changes have been identified, incremental static dependency analysis is performed. As explained in Algorithm 1, the dependency information is already available from the initial setup or from the last test case selection algorithm iteration. The incremental approach here

is that the dependency information that exists is updated (computed), but only for what has changed since the last iteration. This means that the new dependencies between test cases and code are computed again for changed code/tests. This step corresponds to the Line 6 of the Algorithm 2.

At this point, static dependencies between test cases and code are available and updated, and the runtime dependencies between test cases and code are also available (these dependencies are updated each time a test executes, apart from the initial setup). Then, using the data about the changes (Δ) and the static and runtime dependencies (ϕ and ψ, respectively), test selection is performed computing a test suite (γ) that is the list of selected test cases that shall be executed related to the changes that were performed to the code/tests.

After the test selection is performed, the selected test cases shall be executed, using any available testing framework or runner, and that corresponds to the Line 8 of the Algorithm. This step of test cases execution has a simultaneous crucial task: as test cases are executed, and since their code and the software code have been instrumented, so that runtime dependency information might always be up-to-date. The goal is that each time a test case is executed its execution path is recorded to be used for the next iteration of the test selection technique.

Test cases have now been executed and test cases execution path recorded. Provided with that information, the last step of the algorithm (Line 9) comprises an incremental update of the available runtime dependency information between test cases and code.

Figure 4.1 presents an iteration of the test selection technique with chronological order and separating the several framework steps.

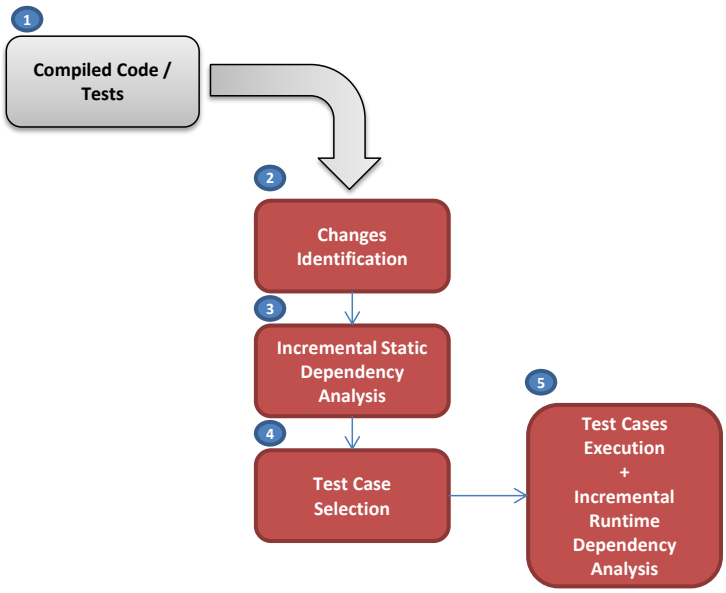

Figure 4.1: Flow of an iteration of the test selection technique

In the presented overview of the test selection technique the steps colored in red are steps for which a more detailed description is relevant for the correct understanding and reproduction of the proposed technique.

Therefore, the next subsections aim to describe with more details the stages presented in Figure 4.1, and specifically the stages of (2) changes identification, (3) incremental static dependency analysis, (4) test case selection and (5) test cases execution and incremental runtime dependency analysis.

4.2.1 Changes Identification

The changes identification stage aims to compute the specific differences between two code versions (μ and μ') and two test suites (τ and τ').

In what concerns the two different code versions, three main kinds of changes might occur: added code, removed code and edited code. In the case of the two test suites, the same kind of changes might occur but related with the test cases: test cases might be added, removed or edited.

One specific relevant point of changes identification is related with the granularity in which the changes identification is performed. Code structures and nomenclatures might be different between different language paradigms. Object oriented languages, for instance, usually have their code structured in files, that contain classes, that contain methods and properties, that contain statements. Functional languages, for instance, are usually structured in files that contain functions. The idea here is that different language paradigms contain different code structures, and therefore different options in what concerns the granularity of dependency analysis. In the proposed technique, and specifically because of the goal of achieving fast feedback, it is very important that it has an efficient implementation. Therefore, there shall exist a trade-off where efficiency is benefited (still without loosing much precision). Thus, coarser-grained approaches might bring more value in the context of the test selection technique that is proposed on this dissertation. On the other hand, precision is still important, so once again the mentioned trade-off must be analyzed wisely.

It is, however, relevant to mention that the majority of VCSs that are widely used for software development only provide information about changes at the file level. At each code version, those VCSs inform the developer of the files that were added, removed or modified in a specific code revision. Apart from that, those VCSs have the ability to visually show the differences between two file versions (or three file versions, taking in account a common base file). Usually those differences are however textual and do not provide by themselves any syntactic meaning of code changes. The proposed test selection technique suggests that, if code changes are to be identified at class, method or statement level, a syntactic analysis is required in order to clearly identify the different code structures (in an objected oriented language this would be classes, methods, properties, statements).

Algorithm 3 presents a description of the changes identification stage of the proposed test selection technique, which aims to identify all the changes between two different code versions and test versions.

Algorithm 3 Changes Identification Algorithm

Let: μ, μ' be source code; τ, τ' be full test suite; λ, λ' be language syntax tree; Δ be a list of differences

```
1 Δ ← [];
2 foreach changed file f between (μ, τ) and (μ', τ') do
3 |   λ  ← Compute File Language Syntax Tree (μ, τ);
4 |   λ' ← Compute File Language Syntax Tree (μ', τ');
5 |   Δ  ← Δ + Find Differences (λ, λ');
6 end
```

As already mentioned before, the proposed technique is based on syntactic analysis of code files to identify specific code structures and therefore have the ability to detect code differences at different granularity levels. This syntactic analysis is presented at Lines 3 and 4 of the Algorithm, and it is performed for each changed code/tests file.

The proposal is that this kind of syntactic analysis is performed using abstract syntax trees, called language syntax trees in this specific context, since they contain some language specific constructs. Those syntactic constructs aim to provide information about what pieces of code are method declarations, statements, properties, or any other code structures that depend on programming languages specificities.

After both syntax trees are computed (for old and new code/tests versions), those trees are compared in order to identify the specific differences that are found in the new code/tests version when compared to the previous one (Line 5 of the Algorithm).

At the end of the execution of the algorithm, a list of differences (Δ) is the output, and it will be used on further steps of the test case selection technique to perform the needed tasks.

4.2.2 Incremental Static Dependency Analysis

Static dependency analysis is the technique stage where dependencies between test cases and code (and between code itself) are analyzed statically, which means, without executing the tests or built code, and only considering the source code and the dependencies that can be extracted from that source code analysis.

In the first iteration of the static dependency analysis, which corresponds to the initial setup, a full analysis is done starting from each test case and gathering all the code that could in theory be triggered directly or indirectly due to each test case execution.

In Section 4.2 of the current chapter it was already mentioned that both static and runtime dependency analysis are performed, and some reasons were already pointed out to justify the need of two different kinds of dependency analysis. Still it is relevant to detail the motivations for performing a static dependency analysis. Those are, specifically:

- Ability to identify new tests or introduced/removed code paths in the last tests/code changes.

On the other hand, static analysis has some weaknesses, and that is another reason why the proposed technique is a mixed one. Specifically, the main disadvantages of static dependency analysis if it is used alone are:

- Inability to identify multi-process tests/code changes in general;

- Lower precision, since static dependency analysis has limitations on identifying the actual paths of execution (that depend on inputs), and as such typically identify more paths than what will happen in runtime.

Despite of the presented weaknesses of this kind of dependency analysis, it is still crucial to identify newly introduced tests and general structural code changes that the runtime dependency analysis would not be capable of identifying.

Once again, the static dependency analysis between test cases and code might also be performed at different granularity levels. Anyway, the granularity that is chosen to perform static dependency analysis must be the same one that was used to identify the tests/code changes, obviously, to maintain the coherence of information and give the presented technique the ability to effectively complete the needed tasks.

The initial setup case was already explained, but the general case also needs to be described. Indeed, the general case is what allows the proposed static dependency analysis to be an incremental static analysis.

In practice, every time a change in tests/code is performed and the list of changes is clearly identified, that list will be used to re-compute only parts of the static analysis information. For instance, if only a specific test case was introduced or changed, it is only necessary to calculate the code dependencies for that specific test case, and not for all the unchanged code. This assumption of an incremental capability is crucial for the presented technique to be efficient and able to scale to large, complex and continuously growing software systems.

Algorithm 4 presents a description of the incremental static dependency analysis.

Algorithm 4 Incremental Static Dependency Analysis Algorithm

Let: μ', τ', Δ, ϕ be: source code; full test suite; a list of differences; static (test ⇒ code) dependencies

```
1  if initial setup execution then
2      foreach t ∈ τ' do
3          φ ← Compute All Calls Recursively (μ', τ');
4      end
5  end
6  else
7      foreach (x ∈ Δ) do
8          if ((x ∈ μ') ∨ (x ∈ τ')) then
9              φ ← Find x and Update All Calls Recursively (x, μ', τ', φ);
10         end
11         else
12             φ ← Compute All Calls Recursively (x, μ', τ', φ);
13         end
14     end
15 end
```

In the presented Algorithm, Line 3 corresponds to the initial setup of the test selection technique, as described in the Algorithm 1. In this case, no differential information is available as well as any static dependency information. Therefore, the full static dependency analysis is performed for all existing tests at this time. Recursively, all code (*e.g.*, methods or classes) that might be reached by each test case is computed and that information is stored in an efficient and appropriate data structure (*e.g.*, a graph).

In the regular execution of the test case selection technique (which is, every case after the initial setup), previous static dependency information and a list of differences are already available. Therefore, if the changed test/code already exists in the static dependencies, it dependencies are recomputed (Line 9). If the test/code is new/removed, its dependencies are computed and added to the existing static dependencies information (Line 12).

At the end of the incremental static dependency analysis stage, static dependencies between each test case and code are available to be used for the next stages of the test case selection technique.

4.2.3 Test Case Selection

Test case selection step is where the test cases that shall be executed due to code changes are selected. This selection is performed using static dependency information and runtime dependency information between test cases and the software code.

Static and runtime dependency information are stored in efficient and appropriated data structures, so that search operations might be efficient for this stage to be fast.

In the mentioned data structures it is crucial that test cases and software code are easily identified, so that test case selection might be as straightforward as possible.

The main idea behind test case selection is that, knowing the list of changes (Δ), for each of those changes the static and runtime dependency data structures are traversed in order to identify all the test cases that are, directly or indirectly, related to the code that was subject to changes.

Algorithm 5 presents a description of the test case selection stage.

Algorithm 5 Test Case Selection Algorithm

Let: γ be selected tests suite; ϕ be static (test $\Rightarrow$ code) dependencies; ψ be runtime (test $\Rightarrow$ code) dependencies; Δ be a list of differences

```
1 γ ← [];
2 foreach change c ∈ Δ do
3 |   γ ← γ + Find Related Test Cases (ψ, φ, c);
4 end
```

The test case selection is the stage of the presented technique where test cases are selected according to static and runtime runtime dependency information (Line 3).

At the end of the test case selection stage a test suite of selected test cases (γ) will be produced, so that they may be executed in the next stage.

4.2.4 Test Cases Execution and Incremental Runtime Dependency Analysis

The last stage of the presented test case selection technique is one where the selected test cases are finally executed. This stage is crucial in the presented technique because the execution of these test cases will contribute to an incremental runtime dependency analysis.

Static dependency analysis is evaluated over source or compiled code, without the need to execute test cases. In runtime dependency analysis, code execution is necessary, and therefore the execution of test cases will be responsible for providing that information for an incremental runtime dependency analysis.

In the first iteration of the technique a full test suite execution is required in order to generate all the dependency information (which is clarified in Algorithm 1). On the following iterations of the technique, only the selected test cases will be executed. It is relevant to mention that runtime dependency information will only be generated for test cases that do not fail. This is crucial because tests that fail might have a different code path than what they should.

Once again, in Section 4.2, it was mentioned that both static and runtime analysis are performed, and that only both analysis are able to guarantee that all the relevant test cases will be selected. Still, it is important to clearly state what are the motivations for performing a runtime dependency analysis. Those are, specifically:

- Ability to identify dependencies between tests and code that span multi-process execution;

- Higher precision, since runtime analysis records the actual execution path of a test case.

On the other hand, runtime dependency analysis also has some weaknesses, which renders a *mixed static and runtime dependency bases test case selection* a necessary choice. Specifically, the main disadvantages of runtime dependency analysis if it was used alone were:

- Inability to identify newly created test cases, which is important because new tests shall always execute after creation (they were created for some purpose);
- Inability to identify newly created code that is not exercised by any test case, which is important because that code dependencies have to known even before executing test cases.

Runtime dependency information is crucial for the presented technique since it provides the ability of tracing test cases execution through several processes and software modules, and be precise at the same time.

In the presented technique, every test case that is selected to execute and the software code will be instrumented so the execution path triggered by test case will be recorded. This instrumentation shall be made as a post-build step, without affecting the original source code.

In runtime dependency analysis, an incremental approach is also crucial for the technique to be scalable, and that is guaranteed by the fact that only the selected test cases are executed.

Algorithm 6 presents a description of the incremental runtime dependency analysis stage.

Algorithm 6 Incremental Runtime Dependency Analysis Algorithm

```
      μ'        source code
      γ         selected tests suite
Let:  τ'  be    full test suite
      ψ         runtime (test ⇒ code) dependencies
      ι         tests execution record information
1 if initial setup execution then
2 |   ι  ←  Execute All Tests (μ', τ')
3 |   ψ  ←  Create Runtime Dependency Information (ι);
4 end
5 else
6 |   ι  ←  Execute Selected Tests (μ', γ)
7 |   ψ  ←  Incrementally Update Runtime Dependency Information (ι);
8 end
```

The initial setup scenario was already described above. Still, the algorithm Lines 2 and 3 correspond to this initial setup case. All tests are executed, producing information of each test execution path. That information will then be stored in an efficient data structure.

In what concerns the general case (which means, not the initial setup), only the selected test cases are executed. Still, the execution information is produced for each executed test case and that information will be used to update the existing runtime dependency information (Lines 6 and 7).

This is the last stage of the test case selection algorithm, where selected test cases are executed and their runtime dependency information is updated for the next iterations of the presented technique.

4.3 Conclusion

This chapter presented a test case selection technique based on both static and runtime dependencies information between test cases and code.

The main goal of the proposed solution is that only test cases that are related to code changes are executed, therefore achieving a faster and also valuable feedback to software developers.

All the stages of this test case selection technique were presented in a way that they may fit different industrial contexts, as long as there is a way to record dependencies between test cases and code.

Summarizing the different steps of the test case selection technique, any implementation of the proposed technique must implement/use tools to perform the following tasks:

- Changes identification;
- Static dependency analysis;
- Test case selection;
- Runtime dependency analysis.

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 details of an implementation of the solution presented in the Chapter 4. The main goal of the implementation work was to demonstrate the applicability of the proposed technique in practice and also to perform validations and assess the solution behavior. Furthermore, it is desired that the implementation described in this chapter allows the reader to implement the proposed solution by himself.

5.1 Introduction

The foundations of the proposed test case selection technique were already described in Chapter 4, in a very abstract and generic way, so that it might be applicable to a myriad of different contexts.

Still, some details on how to implement the proposed solution in practice are relevant to make all the necessary steps more clear and reproducible.

Following the conclusions of Chapter 4 and analyzing the several steps of the test case selection technique, a real implementation context must provide several tools to perform the different steps of the test case selection technique. Specifically, the following tools might be implemented or used if already available:

- A code changes identification tool, that is responsible for identifying the differences between two code versions;
- A static code analyzer, that is responsible for computing the static dependencies between tests and code (and code itself);
- A test case selection tool, that is responsible for selecting test cases according to changes and dependencies between tests and code;
- A tool to instrument test cases execution, that is responsible for recording tests execution path and compute runtime dependencies between tests and code;

In the present chapter, the toolset and programming languages used for this implementation are described, followed by a specific description of the implementation of each of the technique steps. The presented implementation was designed to fit the specific needs of the industrial context of OutSystems, although it is generalized enough to work in other industrial contexts.

5.2 Toolset

For the implementation of the presented test case selection technique several programming languages, tools and libraries were used. Therefore, before actually describing the imple-

mentation details of the proposed solution, such toolset is first described. In the remaining subsections the used programming languages, tools and libraries will be briefly described, with details on what is their role in the test case selection technique implementation.

5.2.1 Programming Languages

In the implementation of the proposed solution some high-level, general purpose, programming languages such as *C#* and *Python* were used, as well as scripting languages such as *Batch* and *Powershell*.

C# language is an object oriented programming language and was the main language used to develop the several tools that were implemented during this work. The main reason for using this language is that it is also the main language used to the develop the OutSystems Platform, and it makes it easier to perform code analysis.

In what concerns *Python*, *Batch* and *Powershell*, those languages were specially used to automate all the steps of the presented implementation. Specifically, they were used to write several scripts that perform necessary tasks that were needed to automate the test case selection technique.

There would be no point in describing any further the languages mentioned here, since those are well-known languages and a further description of those would not bring any value to the present dissertation.

5.2.2 Tools

In order to implement the presented solution, tools to instrument source code and that support unit testing design and execution were also used. Specifically, those tools are NUnit and PostSharp, which are next described.

5.2.2.1 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.

In the context of this dissertation, NUnit was used to execute the test cases. This software provides different test runners, including a console runner and a GUI runner.

Furthermore, all the subject test cases at OutSystems context were already written to conform with NUnit specifications. 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.

5.2.2.2 PostSharp

PostSharp is an open-source extension to *.NET* languages that provides mechanisms for injecting code over compiled *.NET* assemblies.

This extension relies on the Aspect-oriented Programming (AOP) [Kic96] paradigm that aims to increase the modularity of software and the separation of cross-cutting concerns. It allows that code is separated from the main software code and only injected in it during post-build step, without affecting the original source code contents.

In the specific context of this dissertation, PostSharp was used to instrument the test cases and software code, so that test cases execution path might be recorded.

5.2.3 Libraries

In the presented implementation, several software libraries were also used, and namely *.NET* libraries. Specifically, the used libraries were: SharpSVN, Mono.Cecil, *.NET* Compiler Platform ("Roslyn") and QuickGraph.

The following subsections will further describe each of these libraries and their role in the test case selection technique implementation.

5.2.3.1 SharpSVN

SharpSVN is an open-source *.NET* library which works as a binding of the Apache Subversion (SVN) Client Application Programming Interface (API) for *.NET* (from versions 2.0 to 4.0+).

Basically, SharpSVN encapsulates the whole SVN Client API in a way that Subversion features might be used inside *C#* code.

In the context of this dissertation, SharpSVN was used to create a tool that has the ability to identify the changes between two SVN code revisions.

5.2.3.2 Mono.Cecil

Mono.Cecil is an open-source *.NET* library that aims to provide functionalities to generate and inspect programs and libraries in the European Computer Manufacturers Association (ECMA) Common Intermediate Language (CIL) format, which is an open specification developed by Microsoft.

This library allows the loading and inspection of managed *.NET* assemblies inside a *C#* written application.

Mono.Cecil has the ability of interpreting CIL language and transform that CIL language into *C#* code.

In the context of this work, this library was used to perform the static code analysis, and specifically to create a map of dependencies between test cases and software code without actually executing that test cases or code.

5.2.3.3 *.NET* Compiler Platform ("Roslyn")

Roslyn is the codename of an open-source compiler platform developed by Microsoft, which contains *C#* and *Visual Basic* compilers together with code analysis APIs.

With this library, a developer has the ability to create rich code analysis tools using the same APIs that are used by *Visual Studio* IDE to perform several code analysis tasks. The code analysis APIs contained in Roslyn have several data structures that are specifically designed for *.NET* languages.

In the context of this dissertation, Roslyn was used for the changes identification tool, specifically to create *C#* language syntax trees for the different code versions. This language syntax trees were crucial to identify differences between two code versions.

5.2.3.4 QuickGraph

QuickGraph is an open-source *.NET* library that provides generic directed and undirected graph data structures and algorithms for use in *.NET* languages.

With this library, a developer has the ability to create graph data structures with efficient storage and efficient graph traversing algorithms.

QuickGraph library was used to store static dependency information between test cases and code in a persistent way, and also to continuously update that information.

5.3 Implementation Details

The toolset that was used to implement the proposed test case selection technique was already described. This section aims to describe the implementation details of each of the test case selection technique steps, were already presented in Chapter 4 and that were also enumerated in the section 5.1 of the present chapter.

Prior to the description of the several implementation steps, some relevant implementation decisions will be first described and justified. This is necessary in order to answer the more abstract implementation decisions that were left open by the description of the test case selection technique that was made earlier.

5.3.1 Implementation Decisions

In the abstract solution proposal that was described in Chapter 4, two main aspects of the solution were left open as implementation choices, specifically:

a) changes identification granularity;

b) data structures used for dependency data storage.

The main reason why those two aspects were left open as implementation choices is because different contexts might require different choices.

Regarding changes granularity, the goal of any implementation is to achieve a good trade-off between precision and efficiency. That trade-off might only be evaluated with some experiments or some previous context studies. Also in what concerns data structures used to store dependency data, that might also vary according to the way that dependency analysis is performed and the general infrastructure characteristics.

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

a) Changes Identification Granularity

To implement the presented test case selection technique, it was decided that changes should be identified at method level. This decision has two main reasons:

- In the presented industrial context, identifying changes at class level would be too imprecise (a lot of test cases would be selected). On the other hand, identifying changes at statement level would be too inefficient (too many lines of code);
- test cases are also code methods, so it is more coherent to relate methods to methods.

Therefore, method-level changes identification has the most favorable trade-off and is also more coherent to the presented scenario characteristics.

b) Data Structures used for Dependency Data Storage

In the context of code dependency analysis, the main goal is to relate specific code parts with each other. This goal requires a data structure that is able to store relationships between different elements contained in that same data structure. Furthermore, that data structure has to avoid repeated information and has to be efficient when search operations need to be performed.

Two different data structures were used in this implementation: a graph to store static dependency data, and a dictionary to store runtime dependency data.

The reasons to use a graph in static dependency information storage were the following:

- A graph contains no repeated data;
- a graph provides efficient search operations;
- static code analysis is performed in a chronological order, so it is known which test method or code method eventually calls which code method(s).

The reasons for the usage of a dictionary to store runtime dependency information were the following:

- Runtime code analysis data is not available in a chronological way, because of the way that the code path of test cases that traverse multiple processes was recorded. Each time a new process is reached the execution path inside that process will be recorded. When that process returns the execution control to the previous one the dependencies information is written. This means that it is known which test cases exercise which methods, but there is no information of relationships between different code methods (this will be more detailed in Section 5.3.2.4). Still, this step could have been done with code analysis data recorded in a chronological way, but that would add a significant overhead to the test cases execution;
- a dictionary provides efficient search operations if the dictionary keys are appropriated (those are code methods, in this case).

These implementation decisions will be taken in account when describing further implementation details, in the remaining sections of the current chapter.

5.3.2 Implementation Steps

The present section aims to describe the specific implementation steps of the test case selection technique.

The technique implementation steps will be further described in this section by the following order:

1. Changes Identification;
2. Static Dependency Analysis;
3. Test Case Selection;
4. Code Instrumentation and Runtime Dependency Analysis;

5.3.2.1 Changes Identification

To perform changes identification, a tool was implemented in order to perform this specific task. This tool implements the Algorithm 3 of Chapter 4. It is relevant to mention that Subversion is used as the version control system in this implementation context.

To implement a changes identification tool a *C#* application was developed, using the libraries SharpSVN and *.NET* Compiler Platform ("Roslyn").

Regarding the input information of the changes identification tool, it receives the full address of the working SVN branch, and an interval of revisions from where code changes might be evaluated. In a context of continuous integration, ideally this interval shall be composed of the actual code revision and the previous code revision.

The first task of the changes identification tool is to identify all the files that were changed between the defined interval of revisions. Specifically, it will detect the three kinds of file-level changes that might occur between different code revisions:

- Code files were added,
- code files were removed,
- code files were modified.

After identifying the different changes performed over code files, those files will be analyzed in order to identify specific changes at method-level.

The first step to compare changes to files at method level, is to create a *C#* Syntax Tree for each file, which is a syntax tree provided by the Roslyn library. This syntax tree provides decorations that allow the identification of code methods.

Provided with the syntax trees for added, removed and modified files, the following evaluations are performed to identify which methods were changed from one code revision to another code revision:

- If a file was added/removed, all methods contained in it are new/removed methods, and therefore are considered as changed methods,
- If a file was changed, the first evaluation consists on identifying added or removed methods between the old and the new file versions. All those methods are considered as changed methods. For the methods that are present on both file versions, method bodies will be compared, for each method. If there are any differences in the body of two versions of the same code method, then the method is considered as changed. This verification is sensible to all kinds of changes like including the introduction of blank spaces in a method body. Furthermore, this approach is efficient, and a deeper analysis over method bodies could compromise that efficiency without significant benefits.

At the end of these evaluations, the changes identification tool will produce a text file that contains a list of changed methods and changes test cases, one per line.

5.3.2.2 Static Dependency Analysis

A static dependency analysis tool is needed to compute the dependencies between test cases and code methods, and also the dependencies between methods themselves. This implementation corresponds to the Algorithm 4 of Chapter 4.

A static code analyzer was implemented using the programming language *C#* together with the library Mono.Cecil to perform assembly inspection. To store static dependencies information, QuickGraph library was used to store that information in a graph data structure.

As inputs for this tool, compiled software code and tests assemblies shall be provided. Additionally, and in order to enable incremental graph reconstruction, the file containing the list of changes is received, as well as the current version of the static dependencies graph.

There are two distinct scenarios for static code analysis:

1. Static dependencies graph does not exist yet (initial setup scenario);
2. static dependencies graph already exists (incremental scenario).

In the case of the first scenario (initial setup), the static dependency graph is created from scratch.

Since the goal of static code analysis is to identify which code methods are exercised by which test cases, the analysis will always begin from each test case, and evaluating all the possible code methods that the program execution might eventually reach when triggered by each test case. This evaluation is recursive in the sense that subsequent method calls will be recursively evaluated until all the existing calls are to methods that were already visited.

At the same time that test methods and code methods are visited, the information is simultaneously stored in a graph data structure, including the method names and their relationships.

In this graph, there is an important detail concerning the separation between test methods and software code methods: since the analysis always begins from each test case, all the graph root nodes (which are nodes that have no incoming edges) are test methods. All the non-root nodes are code methods.

In the case of the second scenario (incremental graph reconstruction), instead of performing a static analysis over the whole code, only changed code parts are reconstructed.

As a final output of the static code analysis stage, static dependency information is persisted into a file.

5.3.2.3 Test Case Selection

The test case selection tool is responsible for performing the actual test case selection, given that static and runtime dependency data is already available, as well as a list of changes. This implementation corresponds to the Algorithm 5 of Chapter 4. It is also written in *C#* language, and makes use of the QuickGraph library that, apart from providing graph data structures, also provides mechanisms to build search algorithms.

Test selection is performed in two phases:

1. Test case selection according to runtime dependencies;
2. test case selection according to static dependencies.

The first test case selection is performed by evaluating the runtime dependency information and the list of code changes.

For each change that exists in the list of code changes, the existing runtime dependency dictionary is searched to find the index of the current change. When the method corresponding to each change is identified, all the test cases that are related to that method are selected to be executed.

At the end of this phase, a list of test cases selected by runtime dependency analysis is created. This list will be written to a file, that will contain the full test names of each test case, one per line.

The second test case selection is performed by evaluating the static dependency information and also the list of code changes.

For each change that exists in the list of code changes, the static dependency graph will be traversed in order to find the method node corresponding to that change. After each method node is identified, the graph is traversed backwards in order to find all the root nodes that lead directly or indirectly to the current node.

All the root nodes that are reached for each change are then added to a list of test cases, which is the list of test cases selected by static dependency analysis.

On this second phase all the test cases that are already present on the runtime test case run list will not be selected for the static test case run list, so that test cases will not be repeated on both lists.

Once again, this second list of test cases will be written to a file, that will contain the full test names of each test case, one per line.

The decision for creating two lists in separate has to do with the fact that a developer might be able stop test cases execution directly after the first test case execution phase. This provides an opportunity for a even faster feedback loop. Alternatively developers may choose to see the results of both test list to maximize the number of detected failures.

5.3.2.4 Code Instrumentation and Runtime Dependency Analysis

Runtime dependency analysis, as its name suggests, uses information that is recorded during test cases execution and that is later analyzed and stored as dependency data. This implementation corresponds to the Algorithm 6 of Chapter 4.

In order to perform this kind of dependency analysis, it is necessary to record/trace all code execution paths during test execution. To perform this instrumentation, PostSharp was used to instrument all the code related to test cases or software code.

PostSharp provides capabilities for instrumenting software code by creating separate code that is not included in the original software source code (separation of concerns) but that is later weaved.

Using PostSharp, a library was created using its capabilities to implement the ability to instrument code so that several events might be recorded in specific situations. Specifically, the implemented library has the ability to:

- Trigger events when a test case starts executing and when it ends,

- Trigger events when a code method is invoked,
- Trigger events when a method in a different process starts executing and when that process is left.

In order to create an instrumentation mechanism that is able to trace the code execution path between several processes, it is crucial that the different kinds of communications that might exist between different processes can be correlated to create a single execution path.

In the specific industrial context where this technique was implemented, there are three main kinds of communications between different processes:

- Remoting calls,
- web service calls.

Having these scenarios identified, the general logic used to record test cases execution path was the following:

1. When a test case method starts executing, a text file is created with the name of that test case and a time stamp to uniquely identify the current execution and a variable is set to the name of this file, which is a call context variable that is carried with the execution code path (between different processes),
2. Each time a code method is invoked
   (a) If the method is in the same process as the test case, that method is added to a list data structure corresponding to the call context test variable (to allow multiple test cases executing at the same time),
   (b) If the method is in a different process and invoked with a remoting call, the call context variable goes along with the call to the method so that the remote process will know which test is executing, and then it creates a list inside that process and will add to that list the subsequent method calls. When the remote process ends, the current list of exercised methods is written to the test case corresponding file,
   (c) If the method is in a different process and invoked with a web service, the Simple Object Access Protocol (SOAP) header request is manipulated automatically to include the call context variable content. Then, the logic is the same as for remoting calls.
3. When a test case method ends, the list of methods executing in the same process is written to the file contained in the referenced variable (because method calls to other processes were already written when those processes returned their execution to the current one).

At the end of test cases execution there will exist a set of text files, one for each test case, containing a list of exercised methods during the execution of a specific test case.

Still, this kind of data storage is not efficient for search operations. Furthermore, in order to identify which test cases are related to which methods, it is more efficient that a data structure that relates each code method to a set of test cases is available, instead of the contrary.

Therefore, the information that is contained in the text files that contain the execution path record for each test case is parsed using an application developed in *C#* and stored in a dictionary, which is a data structure available natively by the *C#* language. That data structure information is then serialized and stored in a binary file, so that it is accessible for needed search operations or updates.

The analysis of runtime dependencies information will also work incrementally, similar to the static code dependency analysis. This means that for each test cases execution, the whole runtime dependency information data structure will not be totally re-computed, but instead only the parts of it that were changed.

5.4 Conclusion

The current Chapter aimed to describe an implementation in practice of the test case selection 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 real scenarios. Furthermore, this implementation was crucial to perform several validations and experiments (which will be further detailed in Chapter 6).

Summarizing the current chapter, the following tools were implemented:

- A changes identification tool;
- a static code dependency analysis tool;
- a test case selection tool;
- a tool that has the ability to instrument executing source code and storing that information in a dictionary data structure.

All these tools were described in a way that they might be reproducible in different industrial contexts, although this specific implementation was conducted in the OutSystems industrial context.

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

Chapter 6

Results and Discussion

In this chapter the solution that was presented in this dissertation is assessed in practice. The work described in this chapter was to validate 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 that was developed was subject to a set of concrete experiments that are next described in detail. Furthermore, the proposed technique was also validated relying on well-known test selection evaluation techniques based on different metrics.

The results that were observed are very promising and indicate that our solution fully accomplishes the objectives that were set for it in the first place.

6.1 Introduction

In the context of this chapter, a fault-revealing test case *t* is one that tests a program *P* and whose execution fails, therefore revealing failures in program *P*.

Always executing the full test suite might lead to a huge amount of resources spent in automated testing processes, on the other hand this full test suite execution guarantees that all the existent fault-revealing tests (as well as all other tests) are executed.

Unfortunately, there does not exist an effective procedure to select only the tests that are fault--revealing [Rot96]. In fact, if it were possible to know in advance which test cases would fail before their execution, there would be no point in executing tests. Therefore, our technique and the generality of test selection techniques aim to select modification-related test cases.

A modification-related test case is a test *t* that is used to test a program *P'*, that is a modified version of an original program *P* where code changes C_1,...,C_n were made, such that the result of *t* might be affected because of a change C_i.

While focusing on selecting all modification-related test cases for *P'*, actually it is desired to select (at least) all fault-revealing tests for *P'*. Indeed, if a portion of code in *P* remained unchanged (and is unaffected by changes) in *P'*, there exists no test in the entire suite that can reveal a new fault for such portion. In fact, either: a) that portion had already been shown faulty by at least one test in the (full) suite with which *P* was tested; and in this case that fault remains until the portion is corrected; b) that portion was already not faulty, and still is.

In the case of test selection techniques, the full suite is not expected to be executed often. [1]

[1] In the presented industrial case, as seen before, an initial run of the suite is required in order to

This arises relevant questions in what concerns the ability of test selection techniques to select all the fault-revealing test cases, even when only a subset of the full test suite is executed.

Test selection only brings value in practice if it is able to reveal the same failures that a full test suite would reveal. Otherwise, there would be no point in selecting subsets of tests to be executed if that would not provide enough confidence.

Sometimes it is hard to evaluate test selection techniques, specially in an high complex context like the industrial context that was presented in Chapter 2. It is just not feasible to select test cases from a test suite with thousands of tests that test millions of lines of code and manually validate that each test case is modification-related. It is not feasible because the amount of data that needs to be manually analyzed would make it impossible to evaluate a significant number of scenarios in time.

Therefore, the goal of this chapter is to present a set of different validations that were conducted to verify that the proposed solution and the implemented prototype is able to always select all the fault-revealing test cases, and hence proven to allow to effectively reduce the feedback loop which is the main goal of this work.

Four different types of validations were conducted, each of which with specific and different goals. For every type of validation that was conducted, there is included in the remainder of this chapter one section that describes it in detail.

6.2 Laboratory Validations

The first kind of validations that was performed is referred as laboratory validations. As the name suggests, these are validations where a sample or simulated scenario is created in order to specifically verify some property or functionality. Chronologically, this was the first validation that was conducted in order to analyze the correctness of the presented prototype, and it was performed before moving on to validations within real environments.

The industrial context that was described in Chapter 2 and its complexity demands that a test selection technique is able to deal with some different scenarios. Specifically in what concerns the nature of test cases and the way they interact with the different platform components, all the different interactions found in the software must be covered by the presented technique. To be more specific, the implemented prototype must be able to detect dependencies between test cases and code in the following scenarios:

- test cases that test code in the same process / module as the test;
- test cases that test code between different processes (via remoting[2], WCF services[3] or web services);

record dependencies between tests and code.

[2] An approach to interprocess communication

[3] Framework for building service-oriented applications

- iterative test cases, which are an OutSystems custom NUnit extension used to automatically execute the same test over different scenarios. Each concrete test defines the list of test scenarios a test should execute by specifying an iterator (which is an object that enables the traversing of a specific list) of the different scenarios.

Knowing which scenarios must be covered, a sample laboratory application was created to verify that runtime dependency analysis was capable of dealing correctly with all the above mentioned scenarios. The application that was created for this purpose contains:

- A main client application,
- a WCF Service,
- a Web Service,
- test cases that traverse all boundaries across different modules,
- iterative test cases.

Having implemented a sample application, the approach that was followed to validate the presented technique was the following:

1. For each test case, manually check all the code methods that are exercised during its execution;
2. execute all the test cases with runtime dependency analysis;
3. validate, for each test case, that the obtained list of exercised methods is the same that the one manually checked.

It was expected that, by the end of this validation, the manually checked dependencies between test cases and code were exactly the same that were obtained via runtime dependency analysis. Indeed, in all the scenarios that were tested, this laboratory validation revealed that the implemented runtime were correctly dealing with all the previously defined scenarios (after the process of adjusting the implemented solution).

Table 6.1 summarizes the results of this laboratory validation.

Test	Scenario	Dependencies as expected?
Test 1	Tests main client methods	YES
Test 2	Tests main client methods and WCF methods	YES
Test 3	Tests main client methods and Web Service methods	YES
Test 4	Iterative test case that tests main client methods	YES
Test 5	Iterative test case that tests main client methods and WCF methods	YES
Test 6	Iterative test case that tests main client methods and Web Service methods	YES

Table 6.1: Results of laboratory validations

After validating the implemented prototype in a laboratory scenario, more real validations were needed to verify that the proposed technique is able to work in real scenarios. Therefore, Section 6.3 will present validations that are more realistic, using real code and real test cases.

6.3 Validations in real scenarios with fabricated failures

In this section, the second kind of validations that was considered is described. Validations with induced failures are validation scenarios where real code and real tests are used to validate the functionality of the proposed solution, although failing tests are in a controlled environment. In fact, failures are manually introduced in code that otherwise passes all tests, so that the failing tests might be controlled and predicted.

The main goal of performing validations in real scenarios with induced failures is to verify that the functionality of both runtime and static dependency analysis is preserved in a real context, after the success of laboratory validations. On this specific validation scenario, runtime dependencies are further evaluated (they were not on Section 6.2).

In these validations the implemented prototype is not validated as a whole, only static and runtime dependency analysis are subject of validation. The reason for this decision is that dependency analysis is crucial for the proposed technique, so further end-to-end validations shall only be done when dependency analysis is considered to be fully working as expected and without any known issues.

The validation approach that was considered was the following:

1. Select a subset τ' of the full test suite τ such that all tests in τ' are passing[4],
2. change the code in order to introduce a bug that affects the result of some known tests in τ',
3. identify the failing tests after the introduction of the bug,
4. use static and dependency analysis and perform test case selection,
5. validate that all the failing tests (among others) are selected.

With these validations the main interest was in checking whether all the failing test cases were selected to be executed. As an indication, there was also an interest in observing in practice that executing all tests in τ' takes less time than executing all tests in τ.

Two different validation scenarios were created: a) one where only tests that test code in the same process as the test are selected; and b) another one where tests that traverse the boundaries of several processes/modules are selected.

In the scenario of a), tests baseline includes 483 tests, executing in 1 hour and 8 minutes. Then, a bug was introduced (specifically, an exception was raised on purpose), and 7 test cases started to fail. The main idea was to compare, if only tests related with code changes were selected, how many test cases would have been executed, how much time would it take, and if the same failures would be detected.

[4]This subset is actually the largest subset of non-failing tests, from which also some tests that for technical reasons were blocking the tests execution were removed.

Table 6.2 summarizes the results of this validation.

	Number of Tests	Tests Execution Time	Number of Failing Tests
Tests Baseline	483	1 hour and 8 minutes	7
Selected Tests	7	6 seconds	7

Table 6.2: Comparison of execution time and detected failures of both full tests baseline and selected tests after introducing the bug on scenario a)

Comparing the execution of the full tests baseline and the selected tests, only 1.45% of the total number of tests were selected for execution and they detected the exact same failures as the entire baseline.

In the scenario of b), tests that traverse different processes and modules of the software code were also evaluated. This includes tests that test web services, WCF services, or calls to remote processes using *.NET* Remoting.

More tests were included in this validation scenario (specifically, multi-process tests), with a baseline of 800 tests. Those 800 tests executed in 2 hours and 38 minutes. Following the same procedure as before, a bug was introduced in the code, which caused 14 test cases to fail.

Table 6.3 summarizes the results of this validation.

	Number of Tests	Tests Execution Time	Number of Failing Tests
Tests Baseline	800	2 hours and 38 minutes	14
Selected Tests	26	1 minute and 2 seconds	14

Table 6.3: Comparison of execution time and detected failures of both full tests baseline and selected tests after introducing the bug on scenario b)

Once again, comparing the execution of the full tests baseline and the selected tests, only 3.25% of the total number of tests were selected to be executed and they detected the exact same failures as the entire baseline. The tests that were selected were not all failing in this case, but that was not the goal of the presented technique anyway (the goal was to detect all the test cases related to code changes, whether they were failing or not).

In both validations the proposed technique detected all the failing test cases and revealed a good precision in what concerns the total number of selected test cases. These validation were the first approach for assessing the behavior the proposed technique in a real scenario, although failures were introduced on purpose. These validations increased the confidence in the proposed solution, so it was decided that further validations should be performed on real scenarios. On Section 6.4 is presented a validation scenario without fabricated failures with real data.

6.4 Reintegrate Validations

In the context of software development supported by modern VCSs, it is often the case that source code changes occur in branches other than the main branch. These branches are created

so that only when changes are consolidated, they are merged into the main branch again. This merging steps are called reintegrates.

In this section, the third kind of validations that was considered is described. Reintegrate validations, as they were named, are validations that were conducted in a real reintegrate context.

A reintegrate usually implies merging a significant number of changes in the code, because it is usually made when one or more features, milestones or refactoring tasks are finished, for example. Furthermore, this is also a scenario where test failures might occur after reintegrating the changed code to the main branch, because a lot of code is being merged and a lot of conflicts might occur.

These characteristics of reintegrates make them a good scenario to validate the test selection technique that is proposed in this dissertation, because a lot of changes are involved and the possibility of failing tests exist.

To start with this validations, a real reintegrate scenario, conducted by a given OutSystems R&D team, where some tests had failed when executed was recreated, but using the prototype of test case selection instead of executing the full test suite. Specifically, several reintegrate scenarios were analyzed and one where errors did occur was chosen for this validation purpose.

At this point, the goal was to validate the implementation as a whole, including the change list identification, static dependency analysis, test run list creation and runtime dependency analysis. Previous validations aimed to validate specific parts of the framework, because before analyzing the framework as a whole it was crucial to validate its parts. Also, now it was important to clarify the execution time of each of the framework steps since the time spent in test selection plus the time executing selected tests must always be less than the time spent executing the whole test suite.

Furthermore, the used reintegrate scenario represents an unconstrained scenario in what concerns the failing tests. While in Section 6.3 the failures in the code were fabricated, in this specific scenario all the used data was real.

Once again, a subset of the full test suite was used in order to perform validations, specially because it was wished to deal with a more concise and faster execution sample. Nevertheless, such scenario will be considered later. Anyway, the subset of test cases that was used was composed by tests from different parts of the OutSystems Platform and without excluding some specific tests with some specific characteristics.

The baseline of tests used for reintegrate validations was of 2253 test cases that executed in 3 hours and 23 minutes, and of those 91 were failing.

The main goal was, therefore, to evaluate whether by selecting test cases with the proposed test selection technique, the failing detection capabilities of the selected group of tests would be the same as those of the baseline that was considered. A complementary goal was to evaluate whether adding the time of running the technique to select the tests and the time of executing the selected tests would be faster than executing the full baseline test suite.

To start, the time that was needed to perform the different steps of the test selection was evaluated. Table 6.4 shows the execution times of the different steps of the test selection technique and the execution time of the selected tests.

	Runtime	Static
Dependency Analysis	0	7 minutes
Change List Identification	15 minutes	
Test Run List Creation	8 minutes	
Selected Tests Execution	1 hour and 30 minutes	16 minutes
TOTAL TIME	**2 hours and 26 minutes**	
TOTAL TIME without the technique	**3 hours and 23 minutes**	

Table 6.4: Execution times of the different steps of the test selection technique

Comparing the time spent in selecting the test cases that should execute plus executing them took 2 hours and 26 minutes. If the full test suite was executed it would imply 3 hours and 23 minutes to execute tests. This means that, using the test selection framework, 57 minutes of tests execution were saved.

Still, it was needed to verify that the fault detection capabilities of the selected tests were preserved when compared to executing the full test suite.

Table 6.5 shows the tests that were selected using runtime dependency information, static dependency information and information from both.

	Failing Tests Detected	Percentage of Total Tests Selected
Tests selected by runtime analysis	89	98%
Tests selected by static analysis	2 [2]	2% [2]
Sum of tests selected by both analysis	91	100%

[2] Does not include tests that were already selected by runtime analysis;

Table 6.5: Tests selected by static and runtime kinds of dependency analysis

In this specific situation, runtime dependency information allowed the selection the great majority of the failing tests. Anyway, static dependency information also allowed the selection of some tests that runtime dependency information would have not selected.

In this validation scenario, all the test cases that revealed failures were selected to execute, and still a significant amount of time executing tests was saved. Still, the steps of change list identification and test run list creation are too dependent on the size of the changes (number and complexity of changes). This means that for each change that exists the files and code methods need to be compared, and this comparison requires some computation time.

Anyway, even if this scenario of a reintegrate context demonstrated that the test selection technique proposed in this dissertation was able to reduce feedback loop time and bring value to the developer, at this scale a fast feedback was still very hard to achieve.

Furthermore, the time spent in any validation in a reintegrate context was very significant. This was the real reason not to pursuit the reintegrates as the validation approach.

Therefore, these numbers suggest that a scenario where code changes are fewer but more frequent, the proposed test selection technique might be more useful and provide real fast feedback, and would also imply less efforts to reproduce and further validate the technique. This is the main motivation for the next validations described in Section 6.5.

6.5 Continuous Integration Validations

This section presents continuous integration validations. Continuous integration validations were validations conducted in a continuous integration environment, where frequent and small commits were made and analyzed during several days. The main goal for this validations was to evaluate the proposed test selection technique as a whole and in a context where its work is fully automated and has absolutely no human intervention.

As already mentioned in Chapter 1, continuous integration demands that developers commit their changes frequently to the source code repository. Continuous integration is highly dependent on fast feedback, because many developers are working on the same branch doing frequent code commits and code updates, and one developer may impact the work of many developers - and needs to act fast when he/she breaks something.

The previously obtained results (from the validations mentioned in previous sections) pointed the need of an analysis in a continuous integration environment, to verify whether that environment would be more favorable for the presented technique. This is actually aligned with OutSystems' interests, since a new initiative inside the company where a project was being developed using continuous integration practices actively was in course at the moment this validations were performed. This context was used to perform constant and automated validations.

The continuous integration initiative that was mentioned contained a test suite that was a subset of the full test suite. The criteria for selecting tests for that subset was based on their execution time (only the fastest-running tests are included). Moreover, this test suite was growing every day, as more tests were created and new code was created. In the validations described in this section, this subset was used as baseline. Although continuous integration implies smaller commits and therefore more validation scenarios are evaluated, one complementary goal of continuous integration validations was also to create an automated validation mechanism where no human intervention was needed.

Therefore, the approach for performing validations in this context was the following:

1. Make use of an existing pipeline[5] that includes build, installation and tests execution with the full tests baseline always executing;
2. create a pipeline similar to the previous one (that includes build, installation and tests execution) but where test case selection is perform to select which tests to execute;
3. constantly evaluate commits in an automated way to validate the behavior of the implemented test selection technique.

[5]Which is a chain of connected stages, executed sequentially, where the output of one stage is the input of the next one

The idea is that both described pipelines are executed in parallel, so that they might be compared and evaluated. Each pipeline instance was triggered by commits to the version control system, which means that each commit would trigger a build and the subsequent stages. This validation configuration was totally automated using continuous integration software, which was *Go*[6], from *ThoughtWorks*.

In this validation approach, several validation scenarios were automatically generated every day and more than once per day, based on real data without any specific restriction on the nature of commits, tests or code. This means that in the validations period every commit was taken into account without any exception.

After each parallel execution of tests and selected tests, it was evaluated whether the test selection technique was able to always include all the tests that were failing, which with a high number of evaluated scenarios would give a good confidence on the functionality of the proposed test selection technique.

Validations were conducted during 2 weeks, with 142 evaluated and taken in account.

Figure 6.1 presents a chart comparing the number of tests that were executed using the full tests baseline execution and the number of tests that were executed using the selected test cases for each commit.

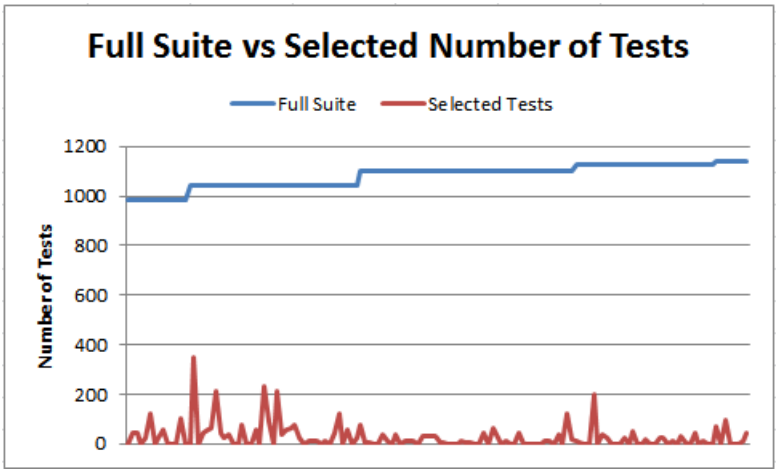

Figure 6.1: The number of tests executed in the full baseline tests vs selected tests

In all of the evaluated cases, all the failing tests were always selected with no exception, and the test suite size and execution times were significantly reduced. Evaluating the test selection results, there is an average test suite reduction of more than 90%. This means that, on average, only half of the test suite is selected to be executed, while fault detection capabilities are still preserved.

Figure 6.2 presents a chart comparing the execution time of tests that were executed using the full tests baseline and the execution time of selected tests.

[6]http://www.go.cd/

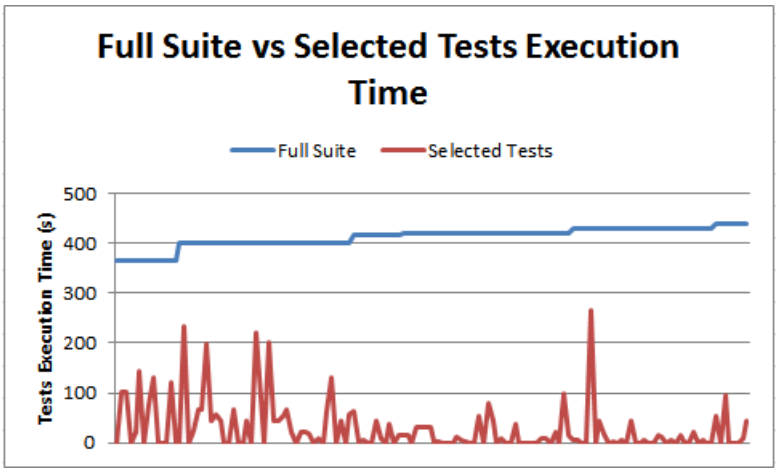

Figure 6.2: The execution time of tests executed in the full baseline tests vs selected tests

Similarly to the number of executed test cases, the time spent executing tests was also suffered a significant reduction towards fast feedback.

In the current scale of the evaluated context, the effective time reductions are not that significant because the full tests baseline by itself is already fast. Anyway, it is evident by the data present in the charts that while the full test suite baseline tends to grow linearly (blue line), the number of selected test cases is not related with the number of test cases existent in the test suite. This clearly shows that the proposed technique does not only provide value and less time spent executing tests, but it also has the ability to scale and adapt as the software code and tests grow. In fact, the full test suite is indeed growing and the feedback loop is getting slower, which requires some solution to allow the project to continue adding more tests without compromising the feedback time, which is what the presented test case selection technique aimed to do.

Furthermore, as the test suite grows (and it is growing every day) the benefit of using the proposed test selection technique will be even more significant and even essential.

6.6 Evaluation Metrics

Apart from the validations that were presented until now, there is a well-known test case selection evaluation framework proposed by Rothermel and Harrold [RH96b]. Although the presented validations seem more relevant to bring evidence of the value of the proposed technique, it would be of interest fo validate the presented technique also using Rothermel and Harrold's work.

The authors define four main metrics to evaluate test case selection techniques: inclusiveness, precision, efficiency and generality.

In what concerns inclusiveness, it is defined as the ability of a test case selection technique to select all the test cases that are related with code modifications. By the validations presented on this chapter, it is clear that every failing test case was always selected. There is enough evidence that enables the claim that the presented test selection technique is 100% inclusive,

and therefore it is considered as a safe test case selection technique.

Precision is the ability of only selecting test cases that are related with code modifications. Also looking at the validation results of this Chapter and the proposed technique, the technique itself is specifically designed to only select test cases that exercise modified pieces of the source code. Therefore, the presented technique is 100% precise in what concerns the relationship between test cases and code.

While, on one hand, inclusiveness and precision are based on quantitative metrics, efficiency and generality are more qualitative and ambiguous.

The main concern about efficiency of any test selection technique is that the cost of selecting test cases plus executing those selected test cases must always be lower than executing the full test suite. On all the different validation scenarios that were described, executing always the full test suite was always slower than selecting test cases and executing them. There is no human intervention needed for the proposed technique to work, so there is also no time spent by developers in test case selection activities. Furthermore, according to the definition of efficiency from Rothermel and Harrold, the presented technique's space requirements are not exponential. Therefore, the proposed test selection technique is efficient.

Generality of test cases selection techniques is the ability of that techniques to work in different contexts without specific or context related assumptions. The only assumption that the presented technique has is that there must be a way to compute the dependencies between test cases and software code. Apart from that, the proposed technique is designed to work without any further restrictions and was validated in an industrial scenario that shares a lot of aspects that are found in the software industry in general, even with some specificities that add even more complexity to it.

6.7 Conclusion

The main goal of the technique proposed in Chapter 4 is to provide developers with fast and valuable feedback. Therefore, all the conducted evaluations were focused on finding clear evidences that the implemented prototype of the proposed test selection technique is able to effectively select the correct test cases (it is able to include, and as such select, all the test cases that reveal failures) and represents a significant feedback time reduction.

In all kinds of validations that were presented in this chapter it was evident that every test case that exposed bugs in the software code was always selected to be executed. The fact that analyzed scenarios were not constrained, controlled and neither homogeneous gives a good confidence that this accuracy will remain in future.

Furthermore, it is clear that the proposed technique might be useful in different scenarios, from a reintegrate to a continuous integration context. If, on one hand, on scenarios with more frequent and smaller commits fast feedback is effectively achieved, on more complex scenarios with larger commits the time spent in testing activities is also significantly reduced.

To conclude this chapter, the validation results conducted to assess the functionality and value of the proposed test selection technique demonstrated 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. It is desired that this chapter clarifies whether the proposed solution comprises with the initial goals set for it. Furthermore, some directions for future work are briefly described.

7.1 Conclusion and Final Results

The main goal of this dissertation was to propose an automated test case selection technique that would be efficient, safe and scalable to large and complex systems.

A specific industrial context, the one of OutSystems, was the context where all the practical work made during this dissertation was conducted. This context was of great interest since the software developed by OutSystems is indeed a complex and large system. Furthermore, OutSystems, as many other companies these days, was suffering from the problem of having a large test suite that takes a significant amount of time to execute. This would allow the present dissertation to give a practical contribution to a practical context and also to evaluate that contribution in a real scenario.

Test case selection and test suite optimizations in general are well established research areas. In the related work study that made part of this dissertation, it was however concluded that the great majority of the already existing works was either: a) designed for standalone and small systems; b) unsafe even if applicable to large systems; or c) developed for large systems but without full automation (human intervention always needed). These studies showed that there were open opportunities to innovate and to create a new solution that would fully automate the test case selection and do it in a safe and efficient way.

The technique that was proposed makes use of both static and runtime dependency analysis to relate code changes to specific test cases. This direct co-relation allowed that only the necessary test cases were selected for execution, instead of the full test suite, which implied significant costs reduction.

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. That implementation was described and it was desired to do it in a way that it might be reproducible.

Concerning the evaluation/validation of the proposed test selection 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 conclude this work and conclude what was its final result, to remember the initial requirements that were set for it were that:

R1) It must be suitable to a multi-process and multi-program environment, where there are several applications and services that interact with each other.

R2) It must be suitable to automated tests that execute code that crosses the boundaries of several processes.

R3) It must be suitable to be applied to an existing large system (code and tests) without the need to change its code and tests.

R4) It must scale to large and continuously growing systems.

R5) It needs to be efficient in what concerns execution performance in practice, which will allow its usage on a continuous integration environment.

R6) It must deal with a constantly changing code base and test suite.

R7) It must apply to real environments, within real, unconstrained, validation scenarios.

The first validations that were conducted during this work aimed to assess the ability of the proposed technique to record code execution paths of automated tests and dependencies between different processes (multi-process environments). Those validations showed that code execution paths between different processes was indeed being recorded successfully, which fulfills the requirements R1) and R2) that were above presented.

Any of the presented validation scenarios dealt with a large system, except for the first one (sample application scenario). The proposed test case selection technique was indeed implemented in a way that it values this point and also avoids unnecessary and persistent changes to be propagated the original source code and source tests, which fulfills the requirement R3).

Automated validations were also conducted in a continuous integration environment with constant code changes and with a growing test suite, requirements R4) and R6), still preserving the efficiency of the presented technique, which did not degrade with time or system growth, in line with requirement R5). Furthermore, these automated scenarios were not selected nor chosen, which means that real scenarios were subject of validations without restrictions, which fulfills the requirement R7).

In summary, the work and the conclusions of its validations, all the initially defined goals and requirements were met, and so an automated test case selection technique that is efficient, scalable and safe was effectively proposed and evaluated.

7.2 Future Work

The proposed test case selection technique is a technique that, as already mentioned during this chapter, uses dependency information between test cases and code to select test cases. This requires that, at least, there is some kind of mechanism to relate tests to software code.

This assumption might point a bottleneck of the purposed technique in what concerns its generalization: there must always be a way to relate tests code with software code. In some specific

types of tests, for instance tests that are defined in a Domain Specific Language (DSL), the relationship between tests definition and software code might not be available or hidden by a mechanism that works behind the curtain.

Tests that are not defined in a way that they can be co-related with software code were left out of the scope of this dissertation. Still, this is an interesting direction for future work: how to select test cases which are defined in non standard ways or using definitions that are specific to a particular domain.

Bibliography

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990. Available from: http://doi.acm.org/10.1145/93548.93576. 24

[AH93] Krauser EW London SA Agrawal H, Horgan JR. Incremental regression testing. *Proceedings of the International Conference on Software Maintenance (ICSM 1993)*, pages 348–357, 1993. 24

[BERL13] Georg Buchgeher, Christian Ernstbrunner, Rudolf Ramler, and Michael Lusser. Towards tool-support for test case selection in manual regression testing. pages 74–79, 2013. Available from: http://dblp.uni-trier.de/db/conf/icst/icstw2013.html#BuchgeherERL13. 25

[BMK04] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=998675.999417. 20

[BMSS11] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011. Available from: http://dblp.uni-trier.de/db/journals/informaticaSI/informaticaSI35.html#BiswasMSS11. 24

[CL96] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Inf. Process. Lett.*, 60(3):135–141, November 1996. Available from: http://dx.doi.org/10.1016/S0020-0190(96)00135-4. 20

[CPU07] Yanping Chen, Robert L. Probert, and Hasan Ural. Regression test suite reduction using extended dependence analysis. In *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SOQUA '07, pages 62–69, New York, NY, USA, 2007. ACM. Available from: http://doi.acm.org/10.1145/1295074.1295086. 21

[CRV94] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. pages 211–220, 1994. Available from: http://dl.acm.org/citation.cfm?id=257734.257769. 25

[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. 18

[DMG07] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition, 2007. 1

[DRK04] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Proceedings of the 15th International*

Symposium on Software Reliability Engineering, ISSRE '04, pages 113–124, Washington, DC, USA, 2004. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ISSRE.2004.18. 23

[ERP14] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. pages 235–245, 2014. Available from: http://doi.acm.org/10.1145/2635868.2635910. 26

[Fis77] K.F. Fischer. A test case selection method for the validation of software maintenance modifications. *In Proceedings of COMPSAC '77*, pages 421–426, 1977. 24

[Fis81] Raji F. Chruscicki A. Fischer, K.F. A methodology for retesting modified software. *In Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6, 1981. 24

[FK03] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. 18

[Har88] Soffa M.L. Harrold, M.J. An incremental approach to unit testing during maintenance. *In Proceedings of the Conference on Software Maintenance*, pages 362–367, 19888. 24

[HGS93] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993. Available from: http://doi.acm.org/10.1145/152388.152391. 19

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. *SIGPLAN Not.*, 36(11):312–326, October 2001. Available from: http://doi.acm.org/10.1145/504311.504305. 24

[HME03] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 60–71, Washington, DC, USA, 2003. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=776816.776824. 21

[HO09] Hwa-You Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 419–429, May 2009. 20

[JG07] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Trans. Softw. Eng.*, 33(2):108–123, February 2007. Available from: http://dx.doi.org/10.1109/TSE.2007.18. 20

[KA09] G. Kaminski and P. Ammann. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 356–365, April 2009. 21

[Kap01] Gregory M Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, 2001. 18

[KGH+93] David C. Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. 1993. 25

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es), December 1996. Available from: http://doi.acm.org/10.1145/242224.242420. 43

[Kni10] Henrik Kniberg. *Kanban and Scrum - Making the Most of Both*. Lulu.com, 2010. 12

[KP02] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 119–129, New York, NY, USA, 2002. ACM. Available from: http://doi.acm.org/10.1145/581339.581357. 23

[KTV02] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 214–223, 2002. 21

[LP03] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 442–, Washington, DC, USA, 2003. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=951952.952367. 23

[MB03] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Softw. Eng.*, 29(11):974–984, November 2003. Available from: http://dx.doi.org/10.1109/TSE.2003.1245299. 20

[MM05] S. McMaster and A.M. Memon. Call stack coverage for test suite reduction. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 539–548, Sept 2005. 21

[MM08] Scott McMaster and Atif Memon. Call-stack coverage for gui test suite reduction. *IEEE Trans. Softw. Eng.*, 34(1):99–115, January 2008. Available from: http://dx.doi.org/10.1109/TSE.2007.70756. 21

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. 1

[MT07] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 276–290, Berlin, Heidelberg, 2007. Springer-Verlag. Available from: http://dl.acm.org/citation.cfm?id=1759394.1759424. 23

[OPV95] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pages 111–123, 1995. 19

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. *SIGSOFT Softw. Eng. Notes*, 29(6):241–251, October 2004. Available from: http://doi.acm.org/10.1145/1041685.1029928. 25

[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 Large Installation System Administration*, LISA '06, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1267793.1267814. 18

[RH93] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. pages 358–367, 1993. 24

[RH94] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. pages 169–184, 1994. Available from: http://doi.acm.org/10.1145/186258.187171. 24

[RH96a] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22, 1996. 4, 23

[RH96b] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996. Available from: http://dx.doi.org/10.1109/32.536955. 62

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997. Available from: http://doi.acm.org/10.1145/248233.248262. 24

[RHD99] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for c++ software. *Software Testing, Verification and Reliability*, 10:2000, 1999. 24

[Rot96] Gregg Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. PhD thesis, Clemson, SC, USA, 1996. AAI9703440. 53

[RUC01] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, October 2001. Available from: http://dx.doi.org/10.1109/32.962562. 22

[Sin11] Y. Singh. *Software Testing*. Cambridge University Press, 2011. Available from: https://books.google.pt/books?id=NjQvKQEACAAJ. 1

[SK00] Patrick J. Schroeder and Bogdan Korel. Black-box test reduction using input-output analysis. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 173–177, New York, NY, USA, 2000. ACM. Available from: http://doi.acm.org/10.1145/347324.349042. 21

[SR05] Mats Skoglund and Per Runeson. A case study of the class firewall regression test selection technique on a large scale distributed software system. pages 74–83, 2005. Available from: http://dblp.uni-trier.de/db/conf/isese/isese2005.html#SkoglundR05. 25

[SSS15] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. Chapter 1 - Technical Debt. In Girish Suryanarayana and Ganesh SamarthyamTushar Sharma, editors, *Refactoring for Software Design Smells*, pages 1 – 7. Morgan Kaufmann, Boston, 2015. Available from: http://www.sciencedirect.com/science/article/pii/B9780128013977000011. 3

[TG05] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Softw. Eng. Notes*, 31(1):35–42, September 2005. Available from: http://doi.acm.org/10.1145/1108768.1108802. 20

[VF97] F. I. Vokolos and P. G. Frankl. Pythia: A regression test selection tool based on textual differencing. 1997. 25

[VMW] Understanding full virtualization, paravirtualization, and hardware assist. *VMware White Paper*. 18

[VTK02] Boris Vaysburg, Luay H. Tahat, and Bogdan Korel. Dependence analysis in reduction of requirement based test suites. *SIGSOFT Softw. Eng. Notes*, 27(4):107–111, July 2002. Available from: http://doi.acm.org/10.1145/566171.566188. 21

[Wei07] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, December 2007. Available from: http://doi.acm.org/10.1145/1327512.1327513. 18

[WL92] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. September 1992. 25

[Yau87] Kishimoto Z. Yau, S.S. A method for revalidating modified programs in the maintenance phase. *In Proceedings of COMPSAC '87*, pages 272–277, 1987. 24