



UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Rafaela Gonçalves de Almeida

**Sound Test Case Generation for Concurrent Features Combining Test Cases for
Individual Features**

Recife

Rafaela Gonçalves de Almeida

Sound Test Case Generation for Concurrent Features Combining Test Cases for Individual Features

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Doutora em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Prof. Dr. Augusto Cezar Alves Sampaio

Coorientador (a): Prof. Dr. Sidney de Carvalho Nogueira

Recife

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Almeida, Rafaela Gonçalves de.

Sound test case generation for concurrent features combining
test cases for individual features / Rafaela Gonçalves de
Almeida. - Recife, 2024.

128f.: il.

Inclui referências.

Tese (Doutorado) - Universidade Federal de Pernambuco Centro
de Informática, Programa de Pós-graduação em Ciência da
Computação.

Orientação: Augusto Cezar Alves Sampaio.

Coorientação: Sidney de Carvalho Nogueira.

1. Teste concorrente; 2. Teste baseado em modelo; 3.
Quiescência. I. Sampaio, Augusto Cezar Alves. II. Nogueira,
Sidney de Carvalho. III. Título.

UFPE-Biblioteca Central

Rafaela Gonçalves de Almeida

“Sound Test Case Generation for Concurrent Features Combining Test Cases for Individual Features”

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Doutora em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 19/08/2024.

Orientador (a): Prof. Dr. Augusto Cezar Alves Sampaio

BANCA EXAMINADORA

Prof. Dr. Alexandre Cabral Mota
Centro de Informática / UFPE

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática / UFPE

Prof. Dra. Simone do Rocio Senger de Souza
Departamento de Sistemas de Computação / ICMC-USP

Prof. Dr. Eduardo Henrique da Silva Aranha
Departamento de Informática e Matemática Aplicada / UFRN

To all those whose paths intersected with mine during this pursuit, your contributions (be they a brilliant idea, a comforting word, or simply sharing a moment over a drink) have left an indelible mark on this work.

ACKNOWLEDGEMENTS

I am immensely grateful to my family for molding the person I am today. My mother, in particular, instilled in me the belief that education is the key to unlocking a fulfilling life. Her foresight has been a guiding light on my journey. I also appreciate my baby brother, who has matured into a man throughout this journey, for always being there to listen, share a laugh, and eventually drag me into some rather questionable adventures to ease my sorrows. Their support and encouragement have been priceless on my path.

I am deeply thankful to my advisors, Augusto Sampaio and Sidney Nogueira, for their knowledge, kindness, and understanding. Their mentorship has been instrumental in shaping my academic journey, and I am immensely grateful for their guidance without which this thesis would not have been possible. Beyond their academic expertise, their genuine kindness and encouragement have created a supportive environment that fostered my intellectual growth. I am truly grateful for their belief in my potential and their willingness to share their wisdom, which have been essential to my success.

I would like to thank my colleagues. Your ideas were always inspiring and working together made this experience truly memorable. You made even the toughest challenges feel like a breeze with your support and good humor.

A special thanks to CIn-Trust research group for the opportunity to expose the advances of my research and all the contributions.

Many thanks to CIn/UFPE-Motorola Mobility (a Lenovo company), particularly to Motorola's Regression Testing Team (Vinicius Siqueira, Vitor Silva, Dayvison Silva, Arthur Silva, Pedro Cruz e Daniel Oliveira), Audir Paiva and Viviana Toledo for providing feedback about this work and the opportunity to carry out an empirical evaluation.

Lastly, I acknowledge the grace of God, whose divine guidance and blessings have sustained me through challenges and triumphs alike.

"The cost of a thing is the amount of what I will call life which is required to be exchanged for it, immediately or in the long run." (THOREAU, 1854)

RESUMO

No atual cenário de desenvolvimento de software, onde as aplicações estão se tornando cada vez mais complexas e projetadas para lidar com múltiplas tarefas, simultaneamente, torna-se imperativo validar a confiabilidade em condições concorrentes. Os sistemas podem apresentar uma ampla variedade de interações e comportamentos difíceis de serem reproduzidos, tornando muito desafiador a elaboração de estratégias de teste eficazes. Adicionalmente, a geração de casos de teste para sistemas concorrentes é desafiadora devido à falta de descrições explícitas de seu comportamento concorrente nos requisitos tipicamente capturados em linguagem natural. Nossa principal contribuição é uma abordagem para a geração de testes consistentes baseados em requisitos escritos em linguagem natural, focando, particularmente, em aplicativos de dispositivos móveis. Essa abordagem é enriquecida com uma estratégia de análise de dependência que garante uma ordem de execução consistente dos passos de teste, eliminando, assim, casos de teste com configurações incompletas ou que não podem ser executados devido a pré-condições não atendidas. Além disso, abordamos a consistência (*soundness*) da abordagem proposta a partir da definição de uma nova relação de conformidade, $cspio_q$. Essa nova relação lida efetivamente com a ausência de saídas (quiescência), visando garantir que o sistema possa lidar adequadamente com cenários em que não são esperadas mais saídas ou eventos (característica comum em sistemas concorrentes). Quando os requisitos não estão disponíveis, o que ocorre frequentemente num contexto industrial, um processo de engenharia reversa é necessário para gerar requisitos a partir de casos de teste existentes a fim de garantir a consistência da geração de casos de teste a partir de requisitos. Exploramos essa abordagem para definir uma estratégia consistente (*sound*) de geração de casos de teste que considere a quiescência. No entanto, do ponto de vista prático (de implementação), este processo é bastante oneroso. Como alternativa, propomos uma estratégia de geração de testes otimizada por meio da combinação de passos de teste, denominados de átomos. Esta estratégia visa simplificar o processo, extraindo diretamente novos casos de teste a partir dos existentes sem a necessidade de uma engenharia reversa que tende a ser complexa. Também abordamos a consistência (*soundness*) da abordagem otimizada mostrando a conexão com a abordagem original baseada em engenharia reversa. Implementamos suporte ferramental e conduzimos uma avaliação empírica da eficácia dos testes gerados. Analisamos a cobertura dos testes e o número de falhas durante a execução dos testes criados pelos engenheiros de nosso parceiro industrial, Motorola Mobility (uma empresa da Lenovo). As métricas adotadas foram

então comparadas com aquelas obtidas dos testes gerados usando a abordagem proposta. Os resultados revelam que o conjunto de testes produzido por nossa abordagem apresenta uma cobertura significativamente maior e tem o potencial de identificar mais *bugs* em comparação com o conjunto criado pelos engenheiros da Motorola.

Palavras-chaves: Teste concorrente. Teste baseado em modelo. Quiescência.

ABSTRACT

In the current landscape of software development, where applications are becoming increasingly intricate and designed to handle multiple tasks simultaneously, it is essential to validate reliability under concurrent conditions. Systems can exhibit a wide range of interactions and behaviours that are difficult to replicate, making the creation of effective testing strategies extremely challenging. Additionally, generating test cases for concurrent systems is demanding due to the lack of explicit descriptions of their concurrent behaviour in the typically captured natural language requirements. Our primary contribution involves an approach for generating consistent tests based on requirements expressed in natural language, with a particular focus on mobile device applications. This approach is enhanced with a dependency analysis strategy that ensures a consistent order of test steps execution, thereby eliminating incomplete test cases or those that cannot be executed due to unmet preconditions. Furthermore, we address the soundness of the proposed approach through the introduction of a new conformance relation, denoted as csp_{io_q} . This new relation effectively handles the absence of outputs (quiescence), aiming to ensure that the system can adequately handle scenarios where no further outputs or events are expected, which is a common characteristic in concurrent systems. When updated requirements are not available, which is often the case in an industrial context, a reverse engineering process is necessary to generate requirements from existing test cases, in order to allow the proof of soundness of test case generation from requirements. We explore this approach to define a sound test case generation strategy that considers quiescence. Nevertheless, from a practical (implementation) point of view, this process is rather burdensome. As an alternative, we propose an optimised test generation strategy through the permutation of test steps, referred to as atoms. This strategy aims to simplify the process by directly extracting new test cases from existing ones without the need for a complex reverse engineering process. We also address the soundness of the optimised approach by demonstrating its connection with the original approach based on reverse engineering. We fully implemented tool support and conducted an empirical evaluation of the generated test effectiveness. We analysed test coverage and the number of bugs during the execution of tests created by engineers from our industrial partner, Motorola Mobility (a Lenovo company). The adopted metrics were then compared with those obtained from tests generated using the proposed approach. The results reveal that the test set produced by our approach exhibits significantly greater coverage and has the potential to identify more bugs compared to the set created by Motorola engineers.

Keywords: Concurrent Testing. Model-based testing. Quiescence.

LIST OF FIGURES

Figure 1 – The Process of Model-Based Testing.	24
Figure 2 – TaRGeT generation process.	26
Figure 3 – TaRGeT Test Case for My_Phonebook Application.	27
Figure 4 – Data definition for Feature F1.	28
Figure 5 – Use case 1 (Email).	29
Figure 6 – Active use case.	30
Figure 7 – Two applications running in parallel.	32
Figure 8 – Use case 1 (Video).	32
Figure 9 – Test case from the sixth iteration of TC generation for Feature F1	48
Figure 10 – Test case from $t1_F1_F2$	48
Figure 11 – Generation process with dependency analysis and optimised approach.	49
Figure 12 – 3G-SIM1 Use Case	59
Figure 13 – Example interaction of TC1 with SUT.	61
Figure 14 – Overview of Atom Granularity Levels	63
Figure 15 – Atom Granularity (Function Level)	63
Figure 16 – Original test cases sample, mca_01 and mca_02.	64
Figure 17 – Interleaving of mca_01 and mca_02.	64
Figure 18 – Atom Granularity, highlighted in blue (Action Block Level)	64
Figure 19 – Atom Extraction	66
Figure 20 – Force Controller for Moto Settings Feature	67
Figure 21 – Sanity Automation Package (Uneti Structure)	69
Figure 22 – Interleaving of atoms	70
Figure 23 – Tool's Architecture	81
Figure 24 – Tool's usage workflow	82
Figure 25 – Screen 1 - Test Cases	83
Figure 26 – Screen 2 - Dependency Analysis	83
Figure 27 – Screen 3 - Generated Test Cases	84
Figure 28 – Screen 4 - Execution Results	84
Figure 29 – Dependency Analysis for <i>Performing calls</i> and <i>Managing contact</i> Test Cases.	86
Figure 30 – Fragments of the original Motorola test cases, MOT_001 and MOT_002.	91

Figure 31 – iTC_001 generated from MOT_001 and MOT_002.	91
Figure 32 – BUG-018.	92
Figure 33 – Selected Test Cases Hierarchical Model	97
Figure 34 – Generated Permutation from the interleaving of Product Test use cases (UC_01: Streaming UC_02: Airplane Mode)	97

LIST OF TABLES

Table 1 – CSP model for data elements of Feature F1.	38
Table 2 – CSP model for data elements of Feature F1.	38
Table 3 – CSP model for Use Case UC01 of Feature F1.	39
Table 4 – Formalisation of Feature Composition	43
Table 5 – CSP model for Use Case UC02 from Feature F1	43
Table 6 – CSP model for Feature F1.	44
Table 7 – CSP process that ensures critical region.	45
Table 8 – TC01.	65
Table 9 – TC02.	65
Table 10 – Managing contact [Automation Traceability]	67
Table 11 – Performing Calls [Automation Traceability]	68
Table 12 – Research Questions	88
Table 14 – Sample of Motorola’s Test Teams and areas	93
Table 16 – Selected test cases automation status	94
Table 17 – Selected test cases execution status	95
Table 18 – Selected test cases execution and log capture Status	95
Table 13 – Compilation of uncovered bugs.	101
Table 15 – Overview of the assessment applied to Motorola’s Test Suites	102
Table 19 – TC-MCA-542: Streaming	102
Table 20 – TC-MCA-703: Airplane Mode	102
Table 21 – Use case UC_02:Airplane mode derived from TC-MCA-703.	102
Table 22 – Scenario 1: Comparison results of two different variations resetting the smart- phone only in the first run	103

CONTENTS

	Contents	14
1	INTRODUCTION	16
1.1	PROBLEM STATEMENT	18
1.2	OBJECTIVES AND CONTRIBUTION	19
1.3	THESIS STRUCTURE	21
2	BACKGROUND	23
2.1	MODEL BASED TESTING	23
2.2	MODELLING CONCURRENT FEATURES	25
2.2.1	TaRGeT	25
2.2.2	Modelling Active Use Case	28
2.2.3	Modelling Concurrent Features	31
2.3	TEST CASE GENERATION VIA CSP REFINEMENT	33
2.3.1	CSP operators	33
2.3.2	CSP Semantics for Concurrent Features	37
2.3.3	Test Case Generation Strategy	46
3	A SOUND STRATEGY TO GENERATE TEST CASES FOR CON-	
	CURRENT FEATURES	49
3.1	CONFORMANCE RELATIONS AND CONCURRENCY	50
3.1.1	The $cspio$ Relation	52
3.2	THE $cspio_q$: EXTENDING $cspio$ TO DEAL WITH QUIESCENCE	53
3.2.1	The $cspio_q$ relation	54
3.2.2	Constructing Sound Test Cases	55
3.2.3	Test Suite Generation	58
4	OPTIMISED TEST GENERATION STRATEGY	62
4.1	TEST GENERATION VIA INTERLEAVING OF ATOMS	65
4.2	SOUNDNESS OF THE OPTIMISED APPROACH	70
4.3	OPTIMISED TEST GENERATION COST ANALYSIS	75
5	TOOL SUPPORT	79
5.1	TOOL'S OVERVIEW AND DEVELOPMENT STACK	79
5.2	TOOL'S ARCHITECTURE	80

5.3	TOOLS USAGE WORKFLOW	81
5.4	DEPENDENCY ANALYSIS: KAKI	84
5.5	LIMITATIONS	86
6	EVALUATION	87
6.1	CONTEXT AND MOTIVATION	87
6.2	PLANNING	87
6.2.1	Context Selection	87
6.2.2	Participants	87
6.2.3	Variables	88
6.2.4	Research Questions	88
6.2.5	Metrics	88
6.2.6	Design	89
6.3	RESULTS AND DISCUSSIONS	90
6.3.1	Number of Uncovered bugs	90
6.3.2	Coverage Rate	92
6.3.2.1	<i>Keyword Coverage Scope</i>	93
6.3.2.2	<i>Keyword Coverage Execution and Results</i>	96
6.3.3	Threats to Validity	99
7	RELATED WORK	104
7.1	TEST OF CONCURRENT SYSTEMS	104
7.2	TEST GENERATION FROM NATURAL LANGUAGE MODELS	108
7.3	TEST GENERATION FROM DIAGRAMMATIC AND FORMAL NOTATIONS	109
7.4	TEST GENERATION FROM GENERATIVE AI AND LARGE LANGUAGE MODELS	113
7.5	CONCLUDING REMARKS	114
8	CONCLUSION	116
8.1	STUDY LIMITATIONS	117
8.2	FUTURE WORK	119
	References	122

1 INTRODUCTION

Software testing is a crucial phase within the software development process that entails the methodical validation and verification of a software application or system to assess its functionality, precision, and general quality Burnstein [2006]. Ideally, the objective is to identify and rectify issues prior to the software release to end users or its deployment in a production environment.

The testing procedure includes different levels of granularity and abstraction, such as unit testing, integration testing, system testing, and acceptance testing, to thoroughly assess the software at different stages of its development. Particularly, system testing focuses on the system already integrated.

As software becomes increasingly more complex, system testing requires complex interactions between different components, Application Programming Interface (API)s, databases, and external services. The concurrent execution of multiple threads or processes may result in unforeseeable interactions, posing challenges in guaranteeing the accuracy and dependability of the software. It is, therefore, extremely hard, if not impossible, to anticipate all possible situations of failure. For instance, consider the interaction between two instances—such as a calling function and a display configuration—within split-screen mode, where two applications or windows are used concurrently, displayed side by side on a single screen. While on a call, the user adjusts the screen size, which may lead to minimal effects, such as a partial obstruction of the call button, or potentially result in more significant issues, like rendering the call button unclickable.

When testing reactive systems such as smartphones, the focus is on evaluating how the system responds to stimuli or alterations in their environment, processing events as they arise and modifying their behaviour in real-time. This capacity for dynamic responsiveness is crucial for applications where prompt and precise responses to user interactions, notifications, or other inputs are vital for enhancing user experience and ensuring functionality. Smartphones are highly interactive devices that constantly receive input from various sources like user interactions, network events, sensor data, and system notifications. Concurrent testing of reactive systems can be challenging due to the non-deterministic nature of concurrent execution and the complex interactions between components Andrews and Schneider [1983].

Moreover, in real-life systems, one can encounter various types of interactions and be-

haviours that may require different approaches when designing test cases. While some systems exhibit a straightforward input-output relationship, others may have more complex behaviours that involve multiple sequential inputs or outputs. It's important to note that designing test cases for systems with complex behaviours may require additional testing techniques, such as model-based testing, state machine-based testing, or scenario-based testing.

Performing manual testing is a critical aspect that requires a significant amount of human effort and time to execute test cases and validate the functionality of software Rafi et al. [2012]. As the complexity of the application increases, the number of test cases also increases, resulting in more time and resources being consumed. Additionally, manual testing is susceptible to human errors, such as overlooking defects or inconsistencies, which can impact the overall effectiveness of the testing process. In terms of coverage, due to time limitations and the large number of test cases, it may not be possible to achieve complete test coverage through manual testing alone. Manual tests may also be difficult to repeat, especially if the test cases are poorly documented or if testers fail to consistently follow specific steps. This can make it challenging to accurately reproduce and investigate reported issues. Finally, as the project grows in size and complexity, scaling up manual testing becomes burdensome and may not be feasible in terms of time and resources. Thus, automation has been progressively adopted to balance speed and quality in software releases.

Model-based testing (MBT) is a software testing technique that uses models to represent the behaviour of the system being tested. These models can be used to automate the generation and execution of test cases Broy et al. [2005]. The process typically involves model creation, test generation, test execution, verdict, and analysis. MBT is most effective when the system's behaviour can be accurately represented in models and when the benefits of testing outweigh the cost of creating and managing these models. However, relying on MBT can be a barrier because formal models are the main input for test generation.

In conventional software engineering practices, informal or semi-formal notations, such as Unified Modeling Language (UML) use case diagrams, activity diagrams, or natural language descriptions, are commonly used to document requirements and design Sommerville [2011]. These notations are favoured because they are more easily understood by stakeholders, including non-technical team members. Formal notations contrast with the ones adopted by traditional software engineering approaches, such as use case models. To facilitate the adoption of MBT and make it more accessible, researchers have proposed various approaches de Carvalho [2011], Ferreira et al. [2010b], Nogueira et al. [2014] that use controlled natural language no-

tations to specify input models. These methodologies employ an automated and transparent process to generate test cases from formal models derived from natural language input models. For example, the work in Nogueira et al. [2016] presents an approach implemented by a tool called TaRGeT, which automatically generates black-box tests for mobile applications. In this research, a use case template written in a controlled natural language is used as input model. These templates are then automatically translated into Communicating Sequential Processes (CSP) models Roscoe [2011, 1998], from which test cases are generated and subsequently translated back into natural language. These generated test cases can be executed manually or used as a basis for further automation. This is an attempt to bridge the gap between the formal notations typically used in MBT and the more familiar natural language used in traditional software engineering approaches.

1.1 PROBLEM STATEMENT

For concurrent systems, capturing behaviour using natural language models is considerably challenging. Concurrency involves the simultaneous interaction of multiple entities that is difficult to describe in natural language. For instance, in Carvalho et al. [2015], a strategy is proposed to automatically extract Data-Flow Reactive Systems (DFRS) formal models from requirements written in a controlled natural language, aiming at generating sound test cases for reactive systems. Although various approaches deal with the automatic generation of test cases, a common limitation of these existing works is that it is not possible to explicitly describe the concurrent behaviour of the system under test using natural language models.

An extra challenge for testing concurrent systems is the treatment of inputs and outputs. In some formal relations for testing sequential systems, like the input-output conformance (*ioco*) relation Tretmans [1999], the notion of *quiescence* is used to capture system states that do not generate any output response unless a new input stimulus is provided. In this context, a system is considered to be in a quiescent state when it is idle or stable, having completed all ongoing processes and not actively producing outputs. Concurrent systems often involve multiple processes or threads that interact, culminating in traces involving multiple sequential inputs or outputs. While, for instance, the system requirements can state that an input sequence does not result in any output, in other situations, the lack of expected outputs may indicate a potential problem. Therefore, a theory for testing concurrent systems should be able to differentiate between these two scenarios.

A frequent challenge encountered in the use of MBT is that test cases can often be more up-to-date than the corresponding requirement artefacts in many practical scenarios. Additionally, access to requirement artifacts might be limited or unavailable. In such situations, test cases become a valuable and sometimes the only available source for crafting test models, which are crucial inputs for MBT approaches.

In summary, we emphasise some challenges related to testing concurrent systems.

- Concurrent execution may involve arbitrary and complex interleavings of control flow sequences that are difficult to cover and reproduce by a test strategy.
- Due to the simultaneous flows of execution, these systems may involve elaborate relations between input and output sequences.
- Absence of output (*quiescence*) can be interpreted in different ways; it might be a relevant undesired output lock in a context, but can also be a valid behaviour in another context.
- When adopting natural language for describing system requirements, there is no clear proper mechanism to address concurrency.

1.2 OBJECTIVES AND CONTRIBUTION

In this thesis, we propose an approach for the automatic generation of test cases specifically designed for concurrent features. Our main objective related to testing concurrent systems is the automatic generation of (sequential) test cases that exercise the interaction of parallel components. Our application domain is black box testing of functional requirements of mobile device applications, particularly involving concurrent features. It is not within our scope to verify classical concurrent properties like deadlock, livelock, or (non)determinism. Although our focus is exclusively on mobile applications, we assert that the approach may be applicable to other contexts, provided it adheres to the theoretical constraints outlined. We delineate the main contributions of this research as follows.

- The main contribution of this thesis is a sound test generation strategy from requirements written in natural language, in the context of mobile device applications. This is inspired by the approach developed in Almeida [2019], Nogueira [2012], but extended to consider

concurrent behaviour and a dependency analysis strategy that ensures a meaningful (consistent) execution order of test steps. This rules out, for instance, test cases with incomplete setups or whose steps cannot be executed because some preconditions are not satisfied. A simple example of inconsistency is a step that involves sending a message in a state without having previously setup an internet connection.

- As requirements are not always available, in order to use the above mentioned test generation strategy, a reverse engineering process might be required to generate requirements from existing test cases. As an alternative, we propose an optimised test generation strategy via combination of test cases. This approach aims to simplify the process by directly extracting relevant information without the need for complex reverse engineering.
- We develop a formal proof demonstrating that the optimised approach generates identical test suites for any possible input, mirroring the behaviour of the original approach. In this way we establish a link between the extended and the optimised approaches and discuss the preservation of soundness by the optimised approach.
- We propose a strategy to effectively handle the absence of outputs, commonly referred to as quiescence, in concurrent systems. Our strategy aims to ensure that the system can appropriately deal with scenarios where no further outputs or events are expected to occur.
- To assess the effectiveness of the approach, an empirical evaluation is carried out to analyse its coverage and the number of uncovered bugs. The test cases generated using our approach are executed and compared to the test cases developed by Motorola engineers for the same features. The results reveal that the test suite produced by our approach exhibits significantly higher coverage and has the potential to uncover more bugs compared to the suite created by Motorola engineers.
- To support the previous task we considered multiple test suites and automatic test cases. This allows the reproducibility and dependability of the case study. Also, the metric for the coverage assessment is based on the use of a filter mechanism based on keywords, enabling the retention of a precise log data for coverage evaluation.

- Our focus on testing multiple mobile features resulted in the uncovering of a significant number of bugs. We present a compilation of all bugs along with their corresponding descriptions and a detailed example of how the interleavings led to a reported defect.
- We fully implemented tool support and comprehensively describe each stage of the proposed strategy. This includes the tool architecture, interface, and the integration with a dependency analysis tool.

1.3 THESIS STRUCTURE

Chapter 2 overviews a previous strategy we developed that takes as input existing test cases and outputs a test model with concurrent flows (reverse engineering approach) Almeida [2019], supported by the TaRGeT tool Ferreira et al. [2010a]. Moreover, we present the extension of the use case templates that allows the specification of intra-feature and inter-feature concurrency, along with the CSP semantics that is obtained automatically from the use case templates. Finally, the automatic generation of the test cases from the CSP models using the FDR ? model checker is presented.

Chapter 3 presents the primary contribution of this thesis, which is a sound test generation strategy for mobile device applications. The strategy is based on requirements written in natural language and is inspired by Almeida [2019], Nogueira [2012], and expanded to incorporate concurrent behaviour and a dependent analysis strategy. This analysis strategy ensures that test steps are executed in a meaningful and consistent order, eliminating test cases with incomplete setups or steps that cannot be executed due to unsatisfied preconditions. Furthermore, we provide the theoretical background necessary to substantiate the soundness of the proposed approach.

To optimise the test generation process presented in the previous chapter, in Chapter 4, we propose a strategy that involves the interleaving of atoms (test case actions of varying granularity, as detailed later). By leveraging this approach, we aim to provide a more efficient alternative that aligns closely with the automation and execution environment of our industrial partner. The key idea behind our proposed strategy is to generate test cases by interleaving atoms, which are the fundamental units of the system being tested. By interleaving these atoms, we can explore different combinations and configurations, leading to a more comprehensive coverage of the system's behaviour. We also address soundness of the optimised

approach by showing its connection with the formal strategy presented in Chapter 3.

We present the specifics of implementing a tool to support test case generation in Chapter 5.

In Chapter 6, we delve into the thorough examination conducted in an industrial setting, encompassing the preparation, operation, and outcomes of an empirical evaluation. This evaluation was specifically designed to assess the effectiveness of the proposed approach in terms of coverage and uncovering bugs. The chapter provides detailed insights into the methodology, execution, and findings of the evaluation, shedding light on the practical applicability and impact of the approach in an industrial context.

Following that, Chapter 7 explores the body of related work that pertains to the three major areas addressed in this thesis: test generation from natural language descriptions, testing concurrent applications, and the application of formal methods in test generation. This section provides an overview of existing research, methodologies, and techniques relevant to each of these areas. By examining related work, we gain a comprehensive understanding of the existing knowledge landscape and identify the contributions and distinguishing features of our approach within the broader research context.

In Chapter 8, we summarise our contributions and implications of the study. We reflect on the accomplishments and limitations of the proposed approach and highlight the significance of the research in addressing the identified challenges. We also identify potential research areas for further exploration to advance the field.

2 BACKGROUND

In this chapter we provide the foundations of our research including an overview of the previous strategy we have developed extending the work in Nogueira [2012]. The referred approach takes existing test cases as input and produces a test model with concurrent flows through a reverse engineering process Almeida [2019]. In this approach, the automatic generation of test for concurrency scenarios is supported by the TaRGeT tool Ferreira et al. [2010a].

Additionally, we present the extension of the use case templates to accommodate the specification of two concurrent levels: intra-feature (models the concurrent execution of use cases that belong to the same feature) and inter-feature concurrency (concurrent behaviour that arises from the interaction of applications in different features). The extension is conservative in nature, as it incorporates new elements while retaining all the elements from the previous template. This ensures that the extension enables the modelling and generation of test cases for both non-concurrent and concurrent features.

In the next sections we provide a concise introduction to MBT, the TaRGeT tool and detail each of the previously mentioned forms of concurrency using as running example features that run on mobile devices. The subsequent sections describe the CSP semantics for features with concurrent behaviour, how to generate tests automatically from the CSP model, and the update performed in TaRGeT to input the extended template.

2.1 MODEL BASED TESTING

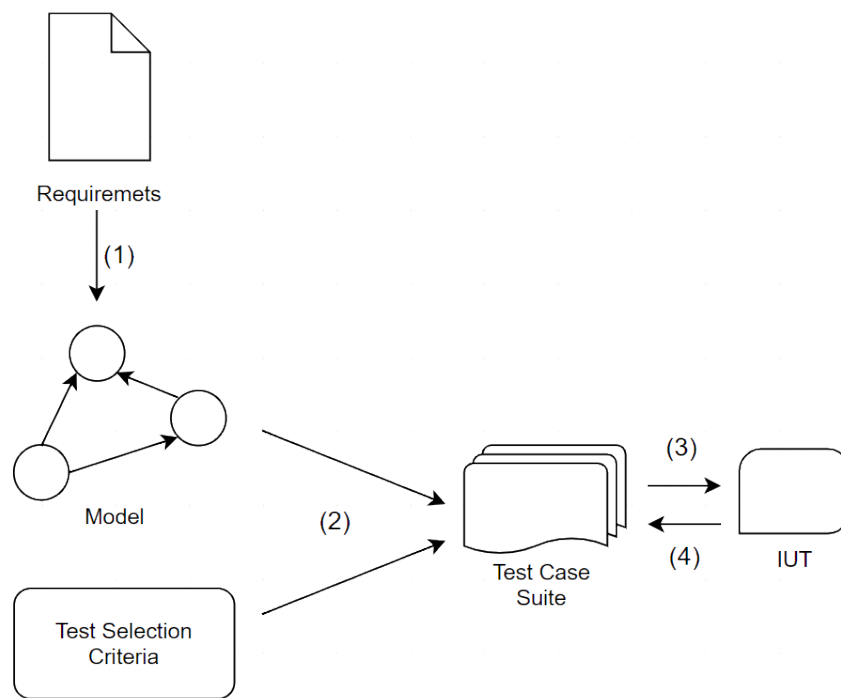
In the realm of software testing, it is a common practice to create and execute test cases manually. The manual testing process is frequently regarded as costly and time-intensive, as it necessitates testers to generate test cases, carry out the tests, and assess the outcomes. In addition, manual testing may prove to be ineffective or impractical, particularly in cases involving the monitoring of non-visible information, such as the interactions occurring between internal components of a system. In order to make the process more effective, automation is used to improve testing. Automating tests offers multiple advantages and contribute to the overall quality of software. Automation can reduce costs, minimise human error, and improve the efficiency of regression testing Sommerville [2004].

By effectively using test automation, organisations can improve the testing process and

enhance the quality of delivered software. The use of formal models, accompanied by mathematical notations, is of utmost importance in facilitating the test automation process. These models offer a methodical and rigorous framework for specifying the input domain and the intended functionality of the system.

Models are used in different domains to comprehend, define, and create systems. They can be used in various stages of the product life-cycle, such as enhancing specification quality, generating code, analysing reliability, and generating tests Apfelbaum and Doyle [1997]. MBT is a method employed to generate test cases automatically Dalal et al. [1999] from behavioural models, such as requirements, that depict certain aspects of the System Under Test (SUT). Some MBT approaches focus on automating software testing by taking an input model (represented as number 1 in Figure 1) and producing a test suite automatically (represented as number 2 in Figure 1).

Figure 1 – The Process of Model-Based Testing.



Source: The author (2018)

The process of MBT involves using a machine-readable input model, such as LTS, to generate test cases. Due to the potentially large or infinite number of test cases, a test selection criteria is applied to choose a feasible subset of tests. The selected tests cover specific parts of the input model or are limited to a maximum number of tests. If the input model is too abstract, the test cases are translated into a more concrete representation for execution

against the IUT. During test execution, the test inputs are provided to the IUT (number 3 in Figure 1), and the responses are compared with the expected outcomes described in the test cases (number 4 in Figure 1). To improve the efficiency MBT, it is important to consider the following factors.

- The model to be manipulated by the user should be clear and easily understandable by all stakeholders, even those not familiar with the application domain. Additionally, the model's scope, defining its capabilities, should be well-defined.
- A systematic approach is crucial for constructing the model. Starting with an abstract description from an informal requirements document, well-defined procedures should guide the construction of a suitable formal model.
- The use of algorithms and tool support is necessary to process the input model. This includes test selection and automation of tests, optimising the testing process.
- The model must include an oracle, which defines the criteria for determining whether a test has passed or failed, providing a mechanism for validating the correctness of the system's behaviour.
- The generated test cases should be sound. Ensuring the reliability of test cases is essential to avoid false negatives and false positives in the testing process.

In summary, MBT has the potential to assist in automating various testing methods across different testing stages. This study specifically concentrates on using MBT to facilitate functional system tests that rely on documents written in natural languages.

2.2 MODELLING CONCURRENT FEATURES

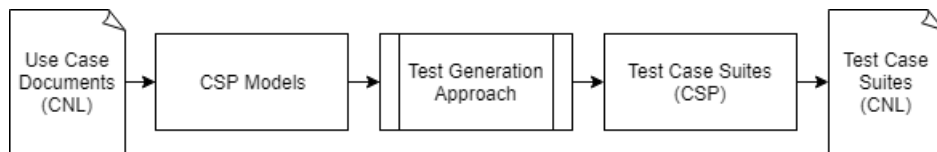
2.2.1 TaRGeT

As already mentioned, in a previous work Almeida et al. [2018] we extended the test case generation approach that is implemented by the TaRGeT tool Ferreira et al. [2010a]. In summary, the tool takes input in the form of standardised use case templates written in controlled natural language to describe the features of the system to be tested. The use cases

within the templates contain natural language descriptions of actions and the corresponding system responses, which guide the testing process.

TaRGeT's use case templates allow the inclusion of input and output values, as well as guards to control the specification flow. These enriched use cases are automatically translated into a formal specification in the notation of CSP process algebra. Figure 2 illustrates TaRGeT's generation process.

Figure 2 – TaRGeT generation process.



Source: The author (2018)

In TaRGeT, the generation of test cases is achieved through the refinement verification between CSP processes using the FDR model checker T.Gibson-Robinson et al. [2014]. Tests are obtained by counterexample traces of refinement using CSP test purposes, which represent a partial specification of the selected test scenarios derived from the input specification. The generated test cases are presented in natural language, and the traces events of the CSP specification are mapped to these elements, producing a test suite suitable for manual execution. The output of the tool is an HTML file containing all the generated test cases (see in Figure 3).

This approach aligns with the recommendations to use a clear and understandable model, to have a tool support to process the input model, and to generate sound test cases to provide reliability on testing. These features enhance the support for test generation in TaRGeT and contribute to an efficient and robust testing process.

The process of generating test scenarios for non-concurrent features is automated by TaRGeT through the execution of FDR refinements in the background until certain stop criteria are met Nogueira et al. [2016]. One common stop criterion is to generate test scenarios until a predetermined threshold of scenarios is reached. Additionally, TaRGeT has the capability to incorporate natural language test purposes, which can describe scenarios that align with specific steps and states outlined in the use case specification. Furthermore, recent advancements have been made to enhance TaRGeT's functionality by considering structural coverage Nogueira et al. [2019].

Using FDR, TaRGeT has access to the underlying LTS models (operational semantics) for the CSP specification, which allows for measuring the coverage of events or transitions for a set of test scenarios. The mapping between the underlying LTS models and the CSP specification enables TaRGeT to consider three structural criteria to control the generation of test scenarios:

- Coverage of Use Case Steps (At Least Once): Test scenarios are generated to ensure that all the steps or actions specified in the use cases are covered at least once. This criterion guarantees that the essential functionalities of the system are tested.
- Coverage of Use Case Steps and the Combinations of Input Values: In addition to covering the use case steps, TaRGeT generates test scenarios that encompass all possible combinations of input values for a given range of values. This criterion aims to explore various input combinations that the system may encounter during real-world usage.
- Coverage of Use Case Steps and the Combinations of Input Values that Match a Given Test Purpose: Test scenarios are generated to fulfill a specific test purpose or objective. This criterion allows for tailored testing to verify specific aspects or requirements of the system.

Figure 3 – TaRGeT Test Case for My_Phonebook Application.

Test Cases	
Test Case ID: 1111_MM_Func_001 Regression Level: na Execution Type: Man Description: None. Objective: None. Use Case References: 1111#UC_01 Requirements: TRS_11111_101 Setups: This is the use case main setup. Initial Conditions: 1) My Phonebook application is installed in the phone. 2) There is enough phone memory to insert a new contact.	
Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the New Contact option.	The New Contact form is displayed.
3) Type the contact name and the phone number.	The new contact form is filled.
4) Confirm the contact creation.	A new contact is created in My Phonebook application.
Final Conditions: None. Cleanup: None. Notes: Under Development	

Source: Ferreira (2010)

The next sections give a brief explanation about TaRGeT's main features.

2.2.2 Modelling Active Use Case

The template proposed by Nogueira et al. [2012, 2016] for automatic test case generation consists of features, with each feature containing multiple use cases. Each use case is comprised of one or more execution flows that can be interconnected. In the original template, a feature executes a single flow at a time, with one flow being completed before the next one begins, thus use cases are performed sequentially. We proposed an extension to the template by introducing "active" use cases, which can be executed concurrently with the flow of other use cases within a feature. Figure 4 illustrates the data associated to the email feature. It encompasses the definition of types, constants, and variables. The type `Natural` is employed to specify the range of integer values, which in this case is $\{0, 1, 2, 3, 4, 5\}$. This range is denoted as $[0, 5]$ for brevity. The limitation of the value range is necessary to prevent an excessive number of test cases during the generation of tests. The constant `MAX_EMAILS` represents the maximum number of emails that the inbox folder can accommodate. The variable `read` denotes the number of read emails and is initialised with a value of two. On the other hand, the variable `unread` indicates the number of unread emails and has an initial value of one.

Figure 4 – Data definition for Feature F1.

F1: EMAIL

Data Definition

Type	Description	Elements
Natural	Numeric range	$[0, 5]$

Constant	Description	Value
MAX_EMAILS	The maximum number of emails	4

Variable	Description	VarType	Value
read	Number of read emails	Natural	2
unread	Number of unread emails	Natural	1

Source: The author (2018)

The initial use case of Feature F1, referred to as UC01, outlines the process of checking unread emails. This use case consists of both main and alternative flows, each of which comprises a series of steps. These steps are characterised by three components: the user input action, the system state (precondition for the step), and the corresponding system response.

The first step in the main flow of UC01 (1M) involves verifying the presence of new emails (see Figure 5). In this context, an unread email is considered equivalent to a new email. Therefore, the condition for this step is the existence of at least one unread email. The use case template employs a Controlled Natural Language (CNL) notation to specify user input values (in the user action), conditional expressions (in the system state), and system outputs/updates (in the system response). These elements are enclosed between two instances of the % symbol. For example, the expression %unread > 0% in Step 1M serves as a guard that evaluates to true if the value assigned to unread is greater than zero. Subsequently, the second step (2M) involves opening unread emails. The user action expression %Input x: Natural from {1..unread}% specifies an input value x that is set by the user. This value must fall within the range of one to the current number of unread messages. In the use case template, the scope of an input value is limited to the step in which it is declared. Therefore, the scope of the value x is restricted to Step 2M. The system response entails marking messages as read and updating the number of unread and read messages accordingly. This adjustment is achieved by decrementing the number of unread messages by the value of x and incrementing the number of read messages by the same value. The output is represented as %unread := unread - x, read := read + x%. For further information on the CNL notation, please refer to Nogueira et al. [2016].

Figure 5 – Use case 1 (Email).

UC01 - Verify unread emails

Main Flow

From Step: START
To Step: END

Step ID	User Action	System State	System Response
1M	Verify the existence of newly received email.	There is unread email. %unread > 0%	The unread emails are highlighted.
2M	Open unread emails. %Input x: Natural from {1..unread}%		Unread emails are marked as read. %unread := unread - x, read := read + x%

Alternative Flow

From Step: START
To Step: END

Step ID	User Action	System State	System Response
1A	Verify the existence of newly received email.	%unread = 0%	There are no unread emails.

Source: The author (2018)

The use case flows are preceded by the fields From Step and To Step, which serve to

denote the initiation and progression of the sequence, respectively. In the context of UC01 the primary sequence originates from the step identified as START, signifying that this particular sequence can be executed from the initial state of the corresponding feature. Furthermore, the continuation of this use case is denoted by the step labelled as END, indicating that the sequence concludes upon accomplishing its final step. In instances where the origin or continuation is specified as a step identifier, the sequence starts from or continues to a particular step within another sequence.

The lower portion of Figure 5 depicts an alternative sequence for UC1 (1M), which follows a state subsequent to the initial stage of the main sequence. Step 1A includes a condition that depends on the absence of unread emails ($\%unread = 0\%$). This condition is only satisfied when the condition of Step 1M is not met. In such a scenario, the alternative sequence is executed instead of the main sequence. The primary action of Step 1A involves verifying the presence of new emails, and the system responds by indicating that no emails are found.

Figure 6 – Active use case.

UC02: Handle new Email <<active>>

Main Flow

From Step: START

To Step: END

Step ID	User Action	System State	System Response
1M	Handle x email(s). %Input x: Natural from {1..MAX_EMAILS - (read+unread)} %	$\%read + unread < MAX_EMAILS\%$	Inbox folder updated. $\%unread := unread + x\%$

Source: The author (2018)

Figure 6 illustrates an active use case, specifically Use Case UC02, which outlines the process of handling new emails concurrently with email reading. Active use cases are distinguished by the label «active» appended to the use case name. The structure of an active use case is identical to that of non-active use cases, but it possesses an independent execution flow. Consequently, an active use case can interleave with the execution of other use cases. In our example, the active use case receives a set number of messages from the network as input. This is specified in Step 1M through the input x, which must not exceed the maximum message capacity, taking into account the existing messages in the application ($\%Input\ x: \text{Natural from } \{1..MAX_EMAILS - (read+unread)\} \%$). The enabling condition for this step is that the number of existing messages must be less than the maximum allowed ($\%read + unread <$

MAX_EMAILS%). If the condition is met, the number of unread messages is updated ($\%unread := unread + x\%$). This use case emulates a component of the email application responsible for receiving new emails and operates concurrently with Use Case UC01.

Given that use cases have the ability to share variables and execute concurrently, it is plausible for race conditions to occur during the simultaneous access to these shared variables. In such scenarios, the access to critical regions can be represented by using variables and guards to regulate the concurrent access to these variables.

2.2.3 Modelling Concurrent Features

This section introduces an expanded version of the current template in order to incorporate concurrent functionalities, defined in Almeida [2019]. In order to address this, supplementary components are added to the template to effectively capture the concurrent characteristics of these features.

As an illustrative instance, we exhibit the use of an Android feature known as Android split screen, which was introduced in Android N www.android.com [a]. This feature allows the screen to be divided into two separate views, with each view independently executing a distinct application (as shown in Figure 7).

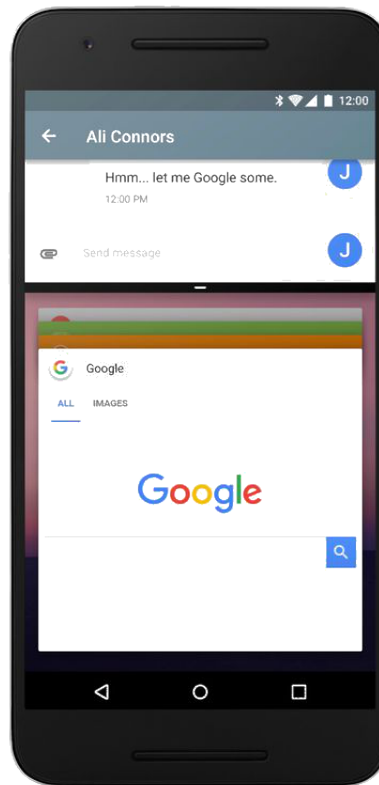
In the uppermost section of the screen, there is a primary application that remains fixed, while in the lower section of the screen, a secondary application is executed simultaneously with the primary one. In this study, we demonstrate the modelling of the Email Application, referred to as Feature F1 in the preceding section, as it operates in the uppermost view, alongside the execution of a Video Player application in the lower view.

Figure 8 illustrates Feature F2, which defines the specifications for the Video Player Application. This feature encompasses a single use case with a main flow, comprising the following steps:

- Selecting a video (Step 1M).
- Verifying if the chosen video starts playing upon clicking the play button (Step 2M).

The proposed extension aims to represent the concurrent behavior of features by specifying their composition within a document. The internal structure of the features remains unchanged. The features can be configured to function as a choice, concurrently, or a combination of both.

Figure 7 – Two applications running in parallel.



Source: Android (2018)

Figure 8 – Use case 1 (Video).

F2: VIDEO**UC01: Video Playing****Main Flow**

From Step: START

To Step: END

Step ID	User Action	System State	System Response
1M	Select a video		The video is highlighted
2M	Click on "Play" button		The selected video starts to play

Source: The author (2018)

In the use case template, the composition of the features is defined after the model of the features. In the subsequent sections, we introduce a notation for specifying this composition.

Composition := fid

| (Composition OR Composition)

| (Composition AND Composition)

As per the aforementioned notation, a composition comprises one or more feature identifiers

that can be combined using the constructors OR and AND. The OR constructor indicates that the composition's arguments are executed as a choice, allowing only one of the features to be executed at a time. In the case of iterated executions, they occur sequentially. On the other hand, the AND constructor signifies that the composed features are performed simultaneously, enabling their behaviours to be interleaved.

For example, the composition (F1 AND F2) signifies that the features F1 and F2 are executed concurrently, effectively modelling the behaviour of the Android split screen functionality, where the Email and Video Player applications run simultaneously on the same screen.

In contrast, the composition (F1 OR F2) indicates that either F1 or F2 is executed, but not both simultaneously.

2.3 TEST CASE GENERATION VIA CSP REFINEMENT

In this subsection, we present the notation and semantic model of CSP used in this thesis, following CSP syntax for the operators.

2.3.1 CSP operators

The fundamental component of the CSP notation is a process, which represents an entity capable of specifying both sequential and concurrent behaviours. A process communicates with other processes through events, and the set of events that a process can communicate forms its alphabet.

The most basic CSP process is represented by *Stop*. This process signifies a deadlock behaviour, where it does not communicate any events and remains in a state of inactivity or lack of progress.

The basic CSP operators are employed to model the sequential behaviour of use cases. The prefix operator \rightarrow is used to specify sequential events. The notation allows defining processes that execute a series of events in a particular order. For instance, if we have two events a and b , the expression $a \rightarrow P$ denotes a process that first communicates event a and then, after its completion, behaves as the process P . This operator captures the idea of sequential behaviour in CSP, allowing the representation of sequential actions in a process.

Another primitive process in CSP is represented by *Skip*, which denotes successful termination or completion. When *Skip* communicates the special event \checkmark , it means that the process

has successfully completed its execution. After communicating this \checkmark event, the *Skip* process enters a deadlock state, where it no longer communicates any events and remains inactive. This behaviour aligns with the notion that a successful process has reached its end and cannot proceed further.

In CSP, recursive processes can be constructed, enabling the specification of processes that repeat their behaviour indefinitely. For example, consider the process $P1$ that communicates the events *on* and *off* in an infinite loop. The notation for this recursive process can be expressed as follows:

$$P1 = on \rightarrow off \rightarrow P1$$

This definition indicates that the process $P1$ starts by communicating the event *on*, then proceeds to communicate the event *off*, and finally, it recursively refers back to $P1$, causing it to repeat the entire sequence of *on* and *off* events indefinitely.

The recursive nature of CSP processes allows the specification of behaviours that can be iteratively executed without ever terminating, which is a valuable feature for modelling systems with continuous or repetitive interactions.

In CSP, a channel specifies an abstraction for a set of events with a common prefix. The notation *channel* $c : T$ represents a channel c that communicates events from the set $\{c.t \mid t \in T\}$. For instance, consider the channel *channel* $c : \{0, 1, 2\}$. This channel can communicate the events $\{c.0, c.1, \text{ and } c.2\}$.

Using this channel, the process $c!0 \rightarrow STOP$ communicates the event $c.0$ and then enters a deadlock state *Stop*. It specifies a behaviour where the process communicates $c.0$ and then stops, not allowing any further communication.

Furthermore, the syntax $c?t$ signifies that the environment binds a value v from the set T to the variable t , and then communicates the event $c.v$. This notation is useful when the process expects to receive a value from the environment and subsequently communicates an event incorporating that value through the specified channel.

Sequential composition is a relevant operator in CSP for modelling sequential behaviour. The expression $P; Q$ specifies that the process P is executed first, and once it terminates successfully (behaves as *Skip*, the control is then passed to the process Q . In other words, $P; Q$ behaves like P until P completes successfully, and then Q takes over.

For example, let's consider the process $a \rightarrow Skip; b \rightarrow Stop$. The behaviour of this process can be understood as follows:

- It starts by communicating the event a .
- After successfully communicating a , it behaves like *Skip* (successful termination) and doesn't communicate any further events.
- As a result, the control is then passed to the process $b \rightarrow \text{Stop}$.
- The process $b \rightarrow \text{Stop}$ communicates b and terminates successfully as *Stop*.

Hence, the overall behaviour is equivalent to $a \rightarrow b \rightarrow \text{Stop}$. The sequential composition operator effectively allows us to simplify the process representation by removing the intermediate *Skip* behaviour.

The parallel composition operator $P \parallel [X] Q$ specifies the concurrent behaviour between processes P and Q . In this composition, the events in the set X are synchronised, meaning they must occur simultaneously on both sides of the parallel composition. All other events communicate independently.

The expression $P \parallel\parallel Q$ denotes the interleave of the processes P and Q , which is a special case of parallel composition where there is no synchronisation (i.e., an empty synchronisation set X).

For example, consider the process $a \rightarrow b \rightarrow \text{Stop} \parallel [a] a \rightarrow c \rightarrow \text{Stop}$. This behaviour is equivalent to the process $a \rightarrow (b \rightarrow \text{Stop} \parallel\parallel c \rightarrow \text{Stop})$, which first communicates the event a and then behaves as the interleave of $b \rightarrow \text{Stop}$ and $c \rightarrow \text{Stop}$. The interleave $b \rightarrow \text{Stop} \parallel\parallel c \rightarrow \text{Stop}$ allows the events b and c to be communicated independently, and their order is non-deterministic.

A crucial aspect of parallel composition is that it only terminates when both processes in the composition have terminated (distributed termination). This property ensures that both processes are synchronised and that the parallel composition is finished only when all the parallel processes have completed their execution.

In CSP notation, indexed operators provide a concise and powerful way to express compositions of processes. Consider the set *Cardinal_direction* that contains the CSP events $\{\text{north}, \text{south}, \text{east}, \text{west}\}$. The process $\square x : \text{Cardinal_direction} @ x \rightarrow \text{Stop}$ uses the indexed external choice operator to express that it behaves as the choice of processes with the form $x \rightarrow \text{Stop}$, where x is replaced by each element in the set *Cardinal_direction*.

So, the process $\square x : \text{Cardinal_direction} @ x \rightarrow \text{Stop}$ can be understood as for each element x in the set *Cardinal_direction*, the process communicates the event x and then

terminates (*Stop*). As a result, the process behaves as the explicit choice between the individual processes $north \rightarrow Stop$, $south \rightarrow Stop$, $east \rightarrow Stop$, and $west \rightarrow Stop$.

Indexed operators can also be applied to other CSP operators, such as interleaving. They offer a concise and flexible way to express behaviours that depend on a set of elements or events, reducing the need for repetitive specifications and improving the clarity of process definitions.

The CSP operator \backslash is used for hiding process communications. When we have a process $P \backslash X$, it means that P communicates all events except the events in the set X . In other words, the process $P \backslash X$ hides the events in X , preventing them from being communicated, while allowing all other events of P to proceed as usual.

Additionally, the notation $P \triangle Q$, where \triangle stands for the interruption operator, indicates that the process Q can interrupt the behaviour of P if an event of Q is communicated. This composition behaves as P until an event in Q is communicated.

These operators provide a way to control the flow of events in the composition of processes, enabling the specification of more complex and flexible system behaviours. They are particularly useful when modelling systems with interrupt-driven or event-driven interactions.

The traces model is one of the simplest semantic models in CSP. In this model, a process is represented by the set of all the traces it can perform. A trace is a sequence of visible actions performed by a process. Visible actions are events that are observable or externally visible from the environment. In the given example, the traces model of the process $P = x \rightarrow y \rightarrow Stop$ is the set $\{\langle \rangle, \langle x \rangle, \langle x, y \rangle\}$. The empty trace $\langle \rangle$ represents the initial state where no action is performed, indicating the process is at the start before any events are communicated. Likewise, the traces semantics of the process *Skip* is the set $\{\langle \rangle, \langle \checkmark \rangle\}$. Let the interleaving for the traces s and t be defined by the rules

$$\begin{aligned}
 \langle \rangle \parallel s &= \{s\} \\
 s \parallel \langle \rangle &= \{s\} \\
 \langle a \rangle \frown s \parallel \langle b \rangle \frown t &= \{\langle a \rangle \frown u \mid u \in s \parallel \langle b \rangle \frown t\} \\
 &\quad \cup \{\langle b \rangle \frown u \mid u \in \langle a \rangle \frown s \parallel t\}
 \end{aligned}$$

Each trace in the set captures a possible sequence of events that the process P can perform from its initial state to its termination. Traces provide a concise and formal way to analyse the

behaviour of processes and understand their possible execution paths. Our approach performs traces refinement verifications using the FDR tool T.Gibson-Robinson et al. [2014] to generate test scenarios.

In CSP, a process Q refines a process P , denoted as $P \sqsubseteq_t Q$, if the set of traces of Q is a subset of the set of traces of P . For instance, the process $P1$ is defined as $a \rightarrow Skip; accept \rightarrow Stop$, and the traces of $P1$ is $\{\langle \rangle, \langle a \rangle, \langle a, accept \rangle\}$. The process $Q1$ is defined as $a \rightarrow Skip$, and the traces of $Q1$ is $\{\langle \rangle, \langle a \rangle, \langle a, \checkmark \rangle\}$.

To check if $P1$ refines $Q1$, we need to verify whether the traces of $Q1$ are a subset of the traces of $P1$. However, $\langle \rangle, \checkmark, accept \rangle$ is a trace that belongs to $P1$ but does not belong to $Q1$. Therefore, $Q1 \sqsubseteq_t P1$ does not hold, and $\langle a, accept \rangle$ serves as a counter-example.

In general, when checking for refinement, if a counter-example exists, it shows that the process Q does not refine process P as there is at least one trace that is allowed in Q but not in P . For more in-depth information about the traces and other semantic models of CSP refer to Roscoe [1998].

2.3.2 CSP Semantics for Concurrent Features

This section outlines the CSP semantics for the extended use case template discussed earlier. The content is organised according to the elements presented in the use case template. The full details can be found in Almeida [2019]. Types and constants from the use case are directly translated into types and constants within the CSP model. For specific data elements related to Feature F1 (illustrated in Figure 4), the CSP representation is given in Table 2.

In the CSP model, the type *Natural* is represented as a *nametype* (name type) on line 01. Additionally, the constant *MAX_EMAILS* is defined as a *constant* in the CSP model, and it keeps the same name as in the use case specification (line 03).

Variables are considered elements of the set *Vars*, which is explicitly defined as a *datatype* on line 05. This definition allows the model to incorporate and manipulate these variables.

Table 1 – CSP model for data elements of Feature F1.

01	<i>nametype</i> $F1_Natural = \{0..5\}$
02	
03	$MAX_EMAILS = 4$
04	
05	<i>datatype</i> $Vars = F1_read \mid F1_unread$
06	
07	<i>channel</i> $get, set : Vars.F1_Natural$
08	
09	<i>channel</i> $startStep, endStep : IDS_F.IDS_UC.IDS_S$

Table 2 – CSP model for data elements of Feature F1.

01	<i>nametype</i> $F1_Natural = \{0..5\}$
02	
03	$MAX_EMAILS = 4$
04	
05	<i>datatype</i> $Vars = F1_read \mid F1_unread$
06	
07	<i>channel</i> $get, set : Vars.F1_Natural$
08	
09	<i>channel</i> $startStep, endStep : IDS_F.IDS_UC.IDS_S$

In CSP, processes are stateless, meaning they do not have variables. Instead, state is typically modelled using a separate memory process that stores the necessary data.

In Table 2 (line 07), the channels *get* and *set* are declared. These channels serve a specific purpose in the CSP model of a use case. The *get* channel is used to read the value of a variable from the memory process, while the *set* channel is used to update the value of a variable in the memory process.

Later, the memory process is combined in parallel with the processes that represent the features. This parallel composition allows the use cases to communicate with the memory process using the *get* and *set* channels. By employing these channels, the use cases can retrieve information from the memory process or modify its content as needed, enabling effective communication and data exchange within the CSP model.

The usage of channels *get* and *set* in the semantics of use case UC01, as shown in Table 3, is exemplified below:

- In line 08 of Table 3, the communication *get!F1_unread?unread* is used to read the value of the variable *F1_unread* from the memory process. The obtained value is then bound to the local variable *unread* for use within the Use Case UC01.
- As another example, in line 16 of the same table, the event *set!F1_read!(read + x)* is a message to the memory process. This message requests the memory process to update the variable *F1_read* to the value of the expression $(read + x)$. This way, the memory process will store the new value, and it will be accessible for future interactions with other parts of the CSP model.

Table 3 – CSP model for Use Case UC01 of Feature F1.

01	<i>channel in_1_UC01_2M_x : F1_Natural</i>
02	
03	<i>F1_UC01 = F1_UC01_START</i>
04	
05	<i>F1_UC01_START = (F1_UC01_1M \square F1_UC01_1A)</i>
06	
07	<i>F1_UC01_1M = startStep.1.1.1 \rightarrow</i>
08	<i>get!F1_unread?unread \rightarrow unread > 0</i>
09	<i>verify_newly_received \rightarrow unread_highlithed \rightarrow</i>
10	<i>endStep.1.1.1 \rightarrow Skip; (F1_UC01_2M)</i>
11	
12	<i>F1_UC01_2M = startStep.1.1.2 \rightarrow</i>
13	<i>get!F1_unread?unread \rightarrow get!F1_read?read \rightarrow</i>
14	<i>in_1_UC01_2M_x?x : {1..unread} \rightarrow</i>
15	<i>open_unread \rightarrow unread_marked_read \rightarrow</i>
16	<i>set!F1_unread!(unread - x) \rightarrow set!F1_read!(read + x) \rightarrow</i>
17	<i>endStep.1.1.2 \rightarrow Skip</i>
18	
19	<i>F1_UC01_1A = startStep.1.1.3 \rightarrow</i>
20	<i>get!F1_unread?unread \rightarrow unread == 0</i>
21	<i>verify_newly_receive_A \rightarrow no_unread_A \rightarrow</i>
22	<i>endStep.1.1.3 \rightarrow Skip</i>

Source: The author (2023)

The complete CSP model for this use case is provided in the following section. For more comprehensive details regarding the memory process refer to Nogueira et al. [2010].

The CSP model for Use Case *UC01* of Feature *F1*, as presented in Table 3, defines the behaviour of the process *F1_UC01*. This process represents the initial state of *UC01* and is denoted as *F1_UC01_START*, which denotes the beginning state of the use case. Consequently, *START* is represented as a process in the CSP model.

As both the main flow and alternative flows of *UC01* start from *F1_UC01_START*, the behaviour of *F1_UC01_START* is modelled as a choice between two processes: *F1_UC01_1M*, which represents the first step of the main flow, and *F1_UC01_1A*, which represents the first step of the alternative flow. By using the choice construct, the CSP model is able to select between the main flow and the alternative flow based on specific conditions specified in the model.

The process *F1_UC01_1M* represents the step *1M* of Use Case *UC01*. Initially, the process *F1_UC01_1M* communicates the control event *startStep.1.1.1*. This event may trigger other processes or actions related to the beginning of step *1M*. The current value of the variable *unread* is read by the process *F1_UC01_1M*, communicating the event *get!F1_unread?unread* (line 08). This event sends a request to the memory process to retrieve the value of the variable *F1_unread*, and the obtained value is assigned to the local variable *unread*. The behaviour *g & P*, where *g* is a guard and *P* is a process, is equivalent to *if g then P else Stop*. In the context of the model, this construct is used to check a condition represented by the guard *unread > 0* (line 08). If the guard holds true (i.e., *unread > 0*), the process flow continues with process *P*. Otherwise, if the guard evaluates to false (i.e., *unread > 0*), the process execution will deadlock, meaning it will come to a halt. Overall, this behaviour ensures that the process *F1_UC01_1M* checks the value of *unread* and proceeds with the flow only if the condition *unread > 0* holds, effectively handling the main flow of the Use Case *UC01*. If the condition is not satisfied, the process will not progress further, mimicking the behaviour of a deadlock.

If the flow continues, the event *verify_newly_received* that represents the step action is performed, followed by (*unread_highlighted*) that represents the system response, the control event *endStep.1.1.1* and successful termination happens. After terminating, the control is taken by the process that represents Step *2M* (*F1_UC01_2M*).

If the condition *unread > 0* holds, the process flow continues with the main flow of Use Case *UC01*. The event *verify_newly_received* representing the step action is performed. This event may trigger further processes or actions related to the verification of newly received data or information. After performing the step action, the event *unread_highlighted* represents the

system response. This indicates that the system responds to the performed action with the event *unread_highlighted*. The control event *endStep.1.1.1* is communicated, marking the successful termination of Step 1*M*. After the successful termination of Step 1*M*, the control is taken by the process that represents Step 2*M* (*F1_UC01_2M*). This means that the process *F1_UC01_2M* will start executing, continuing the flow of the Use Case *UC01* with Step 2*M*. This sequence of events represents the progression of the main flow of Use Case *UC01*, demonstrating how the CSP model handles the steps and system responses during the execution of the use case.

In the testing theory developed in Nogueira et al. [2012] and applied here, it is essential to maintain the separation of events belonging to different steps in the CSP model's traces. Specifically, each user action (modeled as an input event) must be followed by the respective system response (an output event) within the same step. However, concurrent execution of steps from different use cases does not violate this aspect of the theory. To ensure the atomicity of a step, a standard mechanism of critical regions is employed. The control events *startStep.1.1.1* and *endStep.1.1.1* are used to define the start and end of a critical region, representing the beginning and end of a step, respectively.

Table 2 declares the channels *startStep* and *endStep* at line 09. The sets *IDS_F*, *IDS_UC*, and *IDS_S* are indices for the features, use cases, and steps, respectively. In this specific example, these sets are represented as {1..2}, {1..2}, and {1..3}, respectively. These sets help in organising and referencing the features, use cases, and steps in the CSP model, allowing for clear definition and control over the critical regions to maintain the atomicity of steps.

The process *F1_UC01_2M* (line 12) represents the CSP model for Step 2*M* of Use Case *UC01*. The process *F1_UC01_2M* initially reads the variable values from the memory using the *get* operation (not explicitly shown in the provided context). This allows the process to retrieve the current values of relevant variables for further use. The process inputs a value *x* (line 14), which represents the number of emails to be opened. This input value *x* is communicated through the channel *in₁_UC01_2M_x* (line 01). The channel serves as the means of communication between different processes in the CSP model. Subsequently, the process communicates the events *open_unread* and *unread_marked_read*, representing the action of opening emails and the system response, respectively. These events specify the actions and reactions that occur in response to the user input in Step 2*M* of Use Case *UC01*. The events *set!F1_unread!(unread - x)* and *set!F1_read!(read + x)* represent the updates specified in Step 2*M*. These events indicate that the process is updating the values of variables *F1_unread*

and $F1_read$ in the memory process based on the provided conditions. Finally, the process $F1_UC01_2M$ terminates successfully, concluding Step $2M$ of Use Case $UC01$.

The process $F1_UC01_1A$ (line 19) is the CSP model for Step $1A$. It initially retrieves the number of unread messages from the memory variable $unread$ using a *get* event (not explicitly shown in the provided context). This allows the process to access the current value of the variable for further evaluation. The process then verifies the system condition $unread == 0$. If this condition fails (i.e., $unread$ is not equal to zero), the process will deadlock. This implies that Step $1A$ cannot proceed if there are any unread messages, as the condition requires the number of unread messages to be zero. If the system condition holds true (i.e., $unread == 0$), the process continues its execution. In this case, the process communicates the necessary events to verify the existence of newly received email and the corresponding system response, which indicates that there are no unread emails. After completing the actions and system responses, the process $F1_UC01_1A$ terminates successfully, concluding Step $1A$ of Use Case $UC01$.

The processes $F1_UC01_1M$ and $F1_UC01_1A$ are in choice within the process in line 5 ($F1_UC01_START$), and their guards are disjoint. This means that only one of the two processes can progress at a time based on the value of the variable $unread$, which represents the number of unread messages. If the guard of $F1_UC01_1M$ ($unread > 0$) evaluates to true, then the process $F1_UC01_1M$ will be chosen, allowing the main flow to proceed. This means that if there are any unread messages, the main flow will be taken. On the other hand, if the guard of $F1_UC01_1A$ ($unread == 0$) is true, then the process $F1_UC01_1A$ will be chosen, allowing the alternative flow to proceed. This means that if there are no unread messages, the alternative flow will be taken.

The active use case does not have any specific structure. However, it differs from non-active use cases in that it is intertwined with the feature's non-active use cases, as stated in the CSP model for the feature). The process $F1_UC02$ initially reads the values of the variables using *get* events (Table 5, line 08). This allows the process to retrieve the current values of relevant variables for further evaluation. The process evaluates the system condition (line 09). Depending on whether the condition holds or not, the behaviour proceeds differently. If the system condition holds true, the process inputs a value x defined by the environment (line 10). This input value x is communicated through the channel appropriate for the use case. Subsequently, the process communicates the necessary events that represent the action to handle new emails and the corresponding system response (line 11). These events signify the actions and reactions that occur as part of Use Case $UC02$. The process uses a *set* event

to update the value for the *unread* variable (line 12). This event indicates that the process is updating the value of the *unread* variable in the memory process based on the actions taken in Use Case *UC02*. The process communicates the control event *endStep.1.2.1* (line 12) to mark the successful termination of Use Case *UC02*. Finally, the process *F1_UC02* terminates, concluding Use Case *UC02*.

Table 4 – Formalisation of Feature Composition

01	$F_UCs = CNA_UCs \parallel CA_UCs$
02	
03	$CNA_UCs = \text{if } \#NA_UCs = 0 \text{ then } Skip$
04	$\text{else } \square UC : NA_UCs @ UC$
05	
06	$CA_UCs = \text{if } \#A_UCs = 0 \text{ then } Skip$
07	$\text{else } \parallel UC : A_UCs @ (UC \square Skip)$

Source: The author (2023)

Table 5 – CSP model for Use Case UC02 from Feature F1

01	$channelin_1_UC02_1M_x : Naturals$
02	
03	$F1_UC02 = F1_UC02_START$
04	
05	$F1_UC02_START = F1_UC02_1M$
06	
07	$F1_UC02_1M = startStep.1.2.1 \rightarrow$
08	$get!F1_unread?unread \rightarrow get!F1_read?read \rightarrow$
09	$read + unread < MAX_EMAILS$
10	$in_1_UC02_1M_x?x : 1..(MAX_EMAILS - (read + unread)) \rightarrow$
11	$handle_emails \rightarrow inbox_updated \rightarrow$
12	$set!F1_unread!(unread + x) \rightarrow$
13	$endStep.1.2.1 \rightarrow Skip$

Source: The author (2023)

The CSP model for a feature called *F*, which does not have variables, is denoted as *F_UCs*. The process *F_UCs* is specified in Table 4 and represents the interleaving of two compositions: *CNA_UCs*, which consists of non-active use cases, and *CA_UCs*, which consists of active use cases. *NA_UCs* represents the set of use cases not tagged as active, while *A_UCs* represents the set of use cases tagged as active.

If NA_UCs is not empty, CNA_UCs behaves as an indexed external choice of the use cases in NA_UCs . Otherwise, it behaves as *Skip*. Similarly, if A_UCs is not empty, CA_UCs behaves as an indexed interleaving of $(UC \sqcap SKIP)$, where UC belongs to A_UCs . Otherwise, it also behaves as *Skip*.

The choice $(UC \sqcap SKIP)$ implies that CA_UCs considers execution flows that interleave with the active use case (UC) and flows that do not (when it behaves as *Skip*).

The CSP model for a feature F that includes variables is represented as the composition $(F_UCs \parallel [a_MEM] F_MEMORY \triangle Skip)$, where F_MEMORY is the memory process, and a_MEM is the memory alphabet. In this model, the F_UCs process and the F_MEMORY process run in parallel, sharing the memory alphabet a_MEM . The parallel composition allows them to interact and exchange information through the shared variables in a_MEM . However, to ensure the successful termination of the feature process, the F_MEMORY process is interrupted by *Skip* on the right-hand side of the parallel composition. This setup ensures that the feature process F_UCs can proceed to completion without waiting for the F_MEMORY process, allowing for a more efficient and flexible system behaviour.

The process $F1$, as described in Table 6, represents the CSP model for Feature $F1$. The process $F1$ includes two combinations of use cases: $F1_UC01$, which represents the combination of all use cases, and $(F1_UC02 \sqcap Skip)$, which represents the combination of active use cases. The first combination, $F1_UC01$, likely includes the interactions and behaviour of all use cases within Feature $F1$, regardless of their activation status. The second combination, $(F1_UC02 \sqcap Skip)$, represents the interactions and behaviour of the active use cases within Feature $F1$. The use cases in $F1_UC02$ are the ones that are currently active, and the *Skip* process allows the successful termination of the feature when there are no active use cases.

Table 6 – CSP model for Feature $F1$.

01	$F1 = (F1_UC01 \parallel (F1_UC02 \sqcap Skip))$
02	$\parallel [aF1_MEM]$
03	$(F1_MEMORY \triangle Skip)$

Source: The author (2023)

To be concise, we exclude the CSP model for Feature $F2$.

The system testing model combines all the features and uses the process *stepCR* in its definition (Table 7). By composing *stepCR* in parallel with the features, the model ensures

that the use case steps are executed atomically. Table 7 presents the specification for the process $stepCR$. The process $stepCR$ is designed as a recursive process that waits for the environment to signal two events from the channels $startStep$ and $endStep$. The $startStep$ event indicates the beginning of a step, while the $endStep$ event indicates the completion of that step. Both events must be communicated for the process $stepCR$ to proceed further. Let $CONTROL$ be the set that includes all the events from the channels $startStep$ and $endStep$. When the $stepCR$ process is composed in parallel with the features using synchronisation set $CONTROL$, it ensures that only one step is performed at a time. This synchronisation mechanism guarantees that the system executes each use case step in an isolated and atomic manner, preventing any concurrent interference and ensuring a reliable testing environment.

Table 7 – CSP process that ensures critical region.

01	$stepCR = startStep?f?uc?s \rightarrow endStep!f!uc!s \rightarrow stepCR$
----	---

Source: The author (2023)

The system to be tested, referred to as SYS , is represented by the following CSP process $(Fs \parallel [CONTROL]) (stepCR \triangle Skip) \setminus CONTROL$. Here, Fs represents the combination of features. In this model, SYS combines the features (Fs) in parallel with the recursive process $stepCR$, synchronised using the $CONTROL$ set, and also in parallel with the $CONTROL$ process.

The features are represented by Fs and are composed together using the $\parallel [CONTROL]$ operator, which synchronises their behaviour with the events in the $CONTROL$ set. This ensures that the execution of features happens in coordination with the $startStep$ and $endStep$ events. The process $stepCR$ is recursively defined to wait for the environment to communicate events from the $startStep$ and $endStep$ channels. This recursive process enables the system to handle the steps in an atomic and controlled manner. Moreover, the parallel composition of SYS with the $CONTROL$ process allows for the proper synchronisation of the overall system behaviour with the $CONTROL$ events. Consider the example:

$$SYS0 = ((F1 \sqcap F2) \parallel [CONTROL]) (stepCR \triangle Skip) \setminus CONTROL$$

$$SYS1 = ((F1 \parallel F2) \parallel [CONTROL]) (stepCR \triangle Skip) \setminus CONTROL$$

The *stepCR* process is interrupted by *SKIP* in the *SYS* composition to ensure proper termination of the parallel composition. In the CSP model, features can be combined using the *OR* and *AND* constructors. To obtain the CSP model for a composition of features F_s , the feature IDs are replaced with their respective feature processes. The *OR* constructor is represented by the CSP external choice operator, while the *AND* constructor is represented by the CSP interleaving operator. For instance, let's consider the system models for the compositions $F1 \text{ OR } F2$ and $F1 \text{ AND } F2$, which are denoted as *SYS0* and *SYS1* respectively.

2.3.3 Test Case Generation Strategy

This section demonstrates how FDR is used to automatically create test cases for concurrent features, based on the method introduced in Nogueira et al. [2016]. We use a special event (*accept*) to mark the scenarios we want to produce from the specification process, say S . Let S' be the specification process modified by including an *accept* event at the end of the desired scenarios. Since the mark event is not in the alphabet of S , the refinement $S \sqsubseteq_t S'$ does not hold, and counter-examples yielded by FDR for this refinement verification are test scenarios.

We exemplify test generation using the system specification model *SYS0* introduced in the previous section. As illustration, we consider the scenarios that lead to successful termination. The modified process for *SYS0* that includes the event *accept* after the traces that lead to successful termination is the process $SYS0; \text{accept} \rightarrow \text{Stop}$. The *assert* command of FDR runs a refinement verification and yields the shortest counterexample trace if the refinement does not hold. Thus, the FDR assertion

$\text{assert } SYS0 \sqsubseteq_t SYS0; \text{accept} \rightarrow \text{Stop}$

does not hold and yields the following counterexample trace.

$t1 = \langle \text{get}.F1_unread.1, \text{verify_newly_received}, \text{unread_highlithed}, \\ \text{get}.F1_unread.1, \text{get}.F1_read.2, \text{in_1_UC01_2M_x.1}, \text{open_unread}, \\ \text{unread_marked_read}, \text{set}.F1_unread.0, \text{set}.F1_read.3, \text{accept} \rangle$

Excluding *get* and *accept* events, the trace above represents the behaviour of the alterna-

tive flow of the Use Case $UC01$ of Feature $F1$. To generate other test scenarios from $SYS0$ we use the CSP function $Proc$ Nogueira et al. [2014], formally defined as

$$Proc(\langle \rangle) = Stop$$

$$Proc(s) = head(s) \rightarrow Proc(tail(s))$$

where $head(s)$ and $tail(s)$ yield the head and the tail of a non-empty sequence s . Such a function receives as input a sequence of events and generates a process whose maximum trace corresponds to the input sequence. For instance, to generate a second test scenario we use the process $SYS0 \sqcap Proc(t1)$ as the specification process. Such a process contains the trace $t1$, hence $t1$ is not a counterexample for the refinement assertion below,

$$assert\ SYS0 \sqcap Proc(t1) \sqsubseteq_t SYS0; accept \rightarrow Stop$$

which yields the following test scenario as a counterexample for the above refinement.

$$\begin{aligned} t2 = & \langle get.F1_unread.1, get.F1_read.2, in_1_UC_02_1M_x.1, \\ & handle_emails, inbox_updated, set.F1_unread.2, get.F1_unread.2, \\ & verify_newly_received, unread_highlithed, get.F1_unread.2, \\ & get.F1_read.2, in_1_UC01_2M_x.1, open_unread, \\ & unread_marked_read, set.F1_unread.1, set.F1_read.3, accept \rangle \end{aligned}$$

In general, for obtaining the $(n+1)$ th test scenario (counterexample) from a specification, we need to augment the left-hand side of the refinement expression with the test scenarios already generated, and verify the expression using FDR. Formally, the refinement expression for obtaining the $(n+1)$ th test scenario is

$$S \sqcap Proc(ts_1) \sqcap \dots \sqcap Proc(ts_n) \sqsubseteq_t S'$$

There is a total of six test scenarios that can be obtained from the process $SYS0$ using the expression above. The iterations can be repeated until the desired set of test scenarios is achieved, such as by generating a fixed number of tests.

The trace $t2$ represents a test scenario for intra-feature concurrency in Feature $F1$. In this scenario, the main flow of the Use Case $UC01$ is preceded by the active use case (Use Case $UC02$), which is performed concurrently. In another example, Figure 9 depicts a test case that can be obtained from the sixth iteration (test scenario), representing the interleaving of steps.

We also present a test case that can be obtained from the first iteration (test scenario) of a inter-feature behaviour, resulting from the interaction of the features $F1$ and $F2$. The corresponding trace (sequence of events) yielded by FDR is presented as follows.

$t1_F1_F2 = \langle select_video, video_highlighted, get.F1_unread.1,$
 $verify_newly_received, unread_highlighted, get.F1_unread.1,$
 $get.F1_read.2, in_1_UC01_2M_x.1, open_unread,$
 $unread_marked_read, set.F1_unread.0, set.F1_read.3, mem_update,$
 $click_play_video, video_playing, accept \rangle$

Figure 9 – Test case from the sixth iteration of TC generation for Feature F1

Test Case ID: 006

Initial Conditions: read = 2, unread = 1

Id	Steps	Expected Results
1	Verify the existence of newly received email	The unread emails are highlighted
2	Handle 2 email(s)	Inbox folder updated
3	Open unread emails	Unread emails are marked as read

Final Conditions: read = 4, unread = 0

Source: The author (2019)

Figure 10 – Test case from $t1_F1_F2$

Test Case ID: 001

Initial Conditions: read = 2, unread = 1

Id	Steps	Expected Results
1	Select a video	The video is highlighted
2	Verify the existence of newly received email	The unread emails are highlighted
1	Open unread emails	Unread emails are marked as read
2	Click on "Play" button	The selected video starts to play

Final Conditions: read = 3, unread = 0

Source: The author (2019)

The format of the generated test cases is suitable for manual execution (see Figures 9 and 10).

3 A SOUND STRATEGY TO GENERATE TEST CASES FOR CONCURRENT FEATURES

In the industrial context of this research, we conceived a strategy to generate use cases that combine the behaviour of existing test cases by reverse engineering existing test cases (more details in Almeida [2019]), and then generate additional test cases from the obtained use cases. A preliminary and informal version of the strategy was presented in Section 2.2, with the following limitations: (i) the consistency of test step sequences was informally and manually addressed; (ii) the quiescent behaviour was not considered; and (iii) generating a use case model from test cases and new test cases from the use case model was performed via refinement verifications using FDR, which proved to be costly.

The sound test case generation approach we propose here addresses these three issues, see Figure 11. The flow at the top part of the figure proposes solutions to issues (i) and (ii). This is formalised in the remainder of this section. The flow at the bottom part of the Figure is an optimisation that additionally addresses issue (iii), which is the subject of the next section. The input to both flows is a set of existing test cases for individual mobile features, and the output is a set of (automated) test cases for concurrent feature execution.

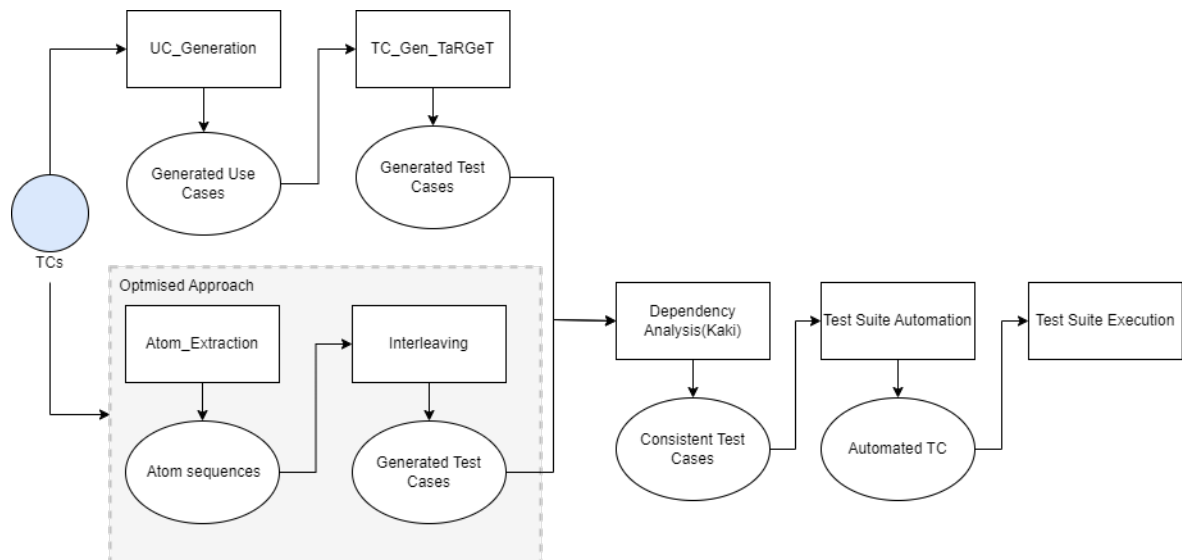


Figure 11 – Generation process with dependency analysis and optimised approach.

Formalising the flow at the top of the Figure 11 requires a use case model that is extracted from the input test cases using a reverse engineering process. Once the use cases are extracted, the generation of test cases for concurrent features is performed by the TaRGeT tool Ferreira et al. [2010b], which first converts the use cases into CSP models. These models are then

used to generate tests by running FDR refinements. The refinement verifications produce test scenarios until a stop criterion is met. Finally, the CSP events in the test scenarios are translated into natural language to create a test suite. This process is integrated with a dependency analysis tool called Kaki de Arruda [2022], a solution to issue (i), as further discussed in Chapter 5. Soundness is addressed via the formal definition of a conformance relation denoted cspio_q (Section 3.2). The construction of sound test cases is addressed in Section 3.2.2 and the effective generation of a test suite is presented in Section 3.2.3.

3.1 CONFORMANCE RELATIONS AND CONCURRENCY

The formal notion of conformance is required to reason about the properties of the generated test cases. From the semantics of formal specification languages, it is possible to derive test cases from systems specifications in a rigorous and automatic manner Bernot et al. [1991]. Moreover, testing theories are able to state the soundness of the test cases based on the formal definition of conformance Gaudel [2005]. Existing theories Tretmans [1999], Nogueira [2012], Carvalho et al. [2013c] rely on a well-defined mathematical relation between the system specification and the implementations under testing (IUTs). Such a relation assumes a set of hypotheses over the IUTs, such as the possibility to build a model for the implementation using a formal notion Tretmans [1999].

A seminal theory is introduced by Tretmans in Tretmans [1999]. This work proposes a conformance testing approach, where the models for the specification and the implementation are expressed as Labeled Transition Systems (LTS) and **io** (input/output conformance) is the conformance relation. The relation splits the observable events in inputs (I) and outputs (U), where an input is provided by the environment and the output is produced by the system. Inputs are always accepted by the system and system output can never be blocked.

Quiescence / Suspension Traces Testing in the context of **io** is based on the observation of visible behaviours and it is characterised by the comparison of the observed suspension traces of the implementation, say $S\text{Traces}(IUT)$, with the suspension traces of the specification, say $S\text{Traces}(S)$. Therefore, information about both traces of a system and quiescence must be recorded.

A quiescent state is characterised by the absence of an output. In the suspension traces, the observation of quiescence is expressed by a special event (δ). The work in Jard and Jéron [2005] highlights the kinds of quiescence behaviour:

- A **deadlock** state is a state where the system is prevented from continuing its execution, where it cannot evolve anymore. Since we assume input enabledness, the system accepts inputs and doesn't produce outputs.
- An **output quiescent** state is a state where the system is waiting only for an input from the environment.

What distinguishes a normal trace from a suspension trace is the fact that in the suspension trace it is possible to observe quiescence. For instance, the suspension traces $S\text{Traces}(S)$ of a specification S corresponds to all the observable behaviours including sequences of inputs, outputs and quiescence.

In the **ioco** formal definition, the set of states reachable from q after a trace σ is yielded by the function $q \text{ after } \sigma = \{q' \mid q \xrightarrow{\sigma} q'\}$. Additionally, $\text{initials}(q) = \{a \in L \mid q \xrightarrow{a}\}$ provides the collection of observable events that may be triggered from q and, considering L_O as output actions, $\text{out}(q) = \text{initials}(q) \cap (L_O \cup \{\delta\})$ yields the set of outputs that may be performed in q (including quiescence).

In suspended/quiescent systems, where no outputs are enabled, it is necessary to wait for the environment to provide an output. The suspension traces of S , $\Delta(S)$, are denoted by $S\text{Traces}(S)$ and represent all behaviours of S which can be observable by the environment (inputs, outputs and quiescence).

Definition 1. (Input-output conformance). Let s be an input-output labeled transition system (IOLTS) and i an input-enabled IOLTS, both with the same alphabets. Then,

$$i \text{ ioco } s \equiv \forall \sigma \in S\text{Traces}(s) \bullet \text{out}(\Delta(i) \text{ after } \sigma) \subseteq \text{out}(\Delta(s) \text{ after } \sigma)$$

Informally, an implementation i is considered to be correct if, and only if, it never produces an output that would not be produced by s in the same conditions. The theory supported by **ioco** is largely used to testing conformance but it cannot be directly used to reason about CSP models, once it is based on LTS models.

The conformance relation that is the basis for this work is named **cspio** (CSP Input-Output Conformance) Nogueira [2012]. For **cspio**, as test hypothesis, it is assumed that implementations to be tested can be formalised as CSP processes. Moreover, like in **ioco**, its alphabet can be split into input and output events and both implementation and specification are livelock free.

3.1.1 The **cspio** Relation

The conformance relation **cspio** Nogueira [2012], inspired by **ioco**, is formalised in terms of the CSP process algebra where specifications and implementations are expressed via CSP processes. Such a relation is supported by the test hypothesis that there is an *I/O* process that specifies an implementation under test (*IUT*). In such a relation, the alphabet (assumed to be known) is split into input and output disjoint sets. Complementary, both *IUT* and specification alphabets are also assumed to be compatible. Both specifications and implementations are represented as triples of the form (P, A_I, A_O) where P is a CSP process, A_I the input alphabet and A_O the output alphabet.

Definition 2. (Compatible alphabets). Let $S = (P_S, A_{I_S}, A_{O_S})$ be the specification and $IUT = (P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$ the implementation models. The alphabets of S and IUT are compatible iff $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$.

Moreover, the implementation is able to accept any input from the alphabet (input-enabled) and always produce some output (output-enabled). The formal definition of an I/O input-enabled process is presented below. Consider that the function $initials(P) = \{a \mid \langle a \rangle \in \mathcal{T}(P)\}$ yields the set of events offered by the process P . An I/O process is input enabled when the inputs offered after each of its traces is the same as its input alphabet.

Definition 3. (Input enabled I/O process). Let $M = (P_M, A_I, A_O)$ be an I/O process. Then, M is input enabled iff $\forall t : \mathcal{T}(P_M) \bullet A_I \subseteq initials(P_M/t)$

Complementary, the formal definition below comprises an output-enabled implementation, where an output event can always be observed.

Definition 4. (Output enabled I/O process). Let $M = (P_M, A_I, A_O)$ be an I/O process. It is output enabled iff $\forall t : \mathcal{T}(P_M); \exists i : A_I, o : A_O \bullet o \in initials(P_M/t \frown \langle i \rangle)$

A way to automatically check whether an I/O process obeys the two last definitions is available in Nogueira [2012].

In what follows, **cspio** is formalised. Consider $out(M, s)$ as an auxiliary function that provides the set of output events of the process component of the I/O process M , P_M , after the trace s . Formally, $out(M, s) = if\ s \in \mathcal{T}(P_M)\ then\ initials(P_M/s) \cap A_{O_M}\ else\ \emptyset$. For an implementation model to conform with its specification, **cspio** establishes that any output

event observed in an implementation model is also observed in the specification S , after any trace of S . Hence, $IUT \text{ cspio } S$.

Definition 5. (CSP input–output conformance). Consider $IUT = (P_{IUT}, A_{IUT}, A_{O_{IUT}})$ an implementation model and $S = (P_S, A_{I_S}, A_{O_S})$ a specification, such that $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$ (compatible alphabets). Then

$$IUT \text{ cspio } S \equiv \forall s : \mathcal{T}(P_S) \bullet \text{out}(IUT, s) \subseteq \text{out}(S, s)$$

3.2 THE cspio_q : EXTENDING cspio TO DEAL WITH QUIESCENCE

Despite the fact that there is a close connection between **cspio** and **ioco** conformance verification, the first relation is based on traces while the second is based on suspension traces.

Another important characteristic of the **cspio** theory is that it considers quiescence-free implementations. This limitation has practical consequences in general and, particularly, in the context of testing concurrent systems. Additionally, to ensure that the generated tests are consistent (sound) according to **cspio**, the step of a use case should be atomic, that is, during the execution of the test it must be guaranteed that an output is found after an input. This restricts the generation of tests for concurrent scenarios where there is an interleaving between inputs and outputs, for instance, and there might be a sequence of outputs for a single input or scenarios where an input is followed by other inputs before an output event occurs. Moreover, due to the nature of the concurrent behaviour and the consequent interaction of different execution flows between applications, a test case can possibly engage in multiple scenarios, such as:

- multiple inputs: several inputs in sequence without the occurrence of an output between them ($\langle \dots i, i, i, \dots \rangle$)
- multiple outputs: several outputs in sequence, without the occurrence of an input between them ($\langle \dots o, o, o, \dots \rangle$).

For some scenarios one may not observe an output between I/O actions, different from the cases (normally sequential systems) where a response corresponding to a given stimulus is

expected ($\langle i, o, i, o \dots \rangle$). It is important to point out that quiescence is not necessarily linked to a concurrent behaviour, it can be observed in any type of system. In the context of sequential systems, there may be a specific interest in monitoring a limited set of outputs, even when a clear response is observable through the interface. Thus, quiescence is a common behaviour while testing sequential and concurrent systems. Our approach addresses quiescence in the context of **cspio** to allow testing of systems with the observation of quiescence.

In order to generate consistent and sound tests for a range of concurrent applications and overcome the aforementioned limitations, our approach extends the concepts of **cspio** and incorporates quiescence as described in the **cspioco** relation Cavalcanti et al. [2016], which provides a semantic treatment of **ioco** within the context of CSP, using suspension traces. For our purposes of reusing and extending the testing theory introduced in Nogueira [2012], however, it is convenient to incorporate quiescence behaviour in the context of **cspio**, rather than using **cspioco**. The principal factor is the intention to mechanise the generation process. This mechanization is already offered by **cspio**, in terms of the CSP process algebra. Thus, it becomes more feasible to reuse what has already been done.

3.2.1 The cspio_q relation

The cspio_q relation is based on **cspio** Nogueira et al. [2014] but relaxes a significant restriction imposed by **cspio** that requires that the interaction with the IUT strictly alternates inputs and outputs, a property that we have previously defined as output-enabledness. This restricts the generation of tests for concurrent scenarios where the interaction of different execution flows between applications can possibly guide a test case to engage in multiple sequence of inputs without an output between them. The observation of quiescent behaviour allows this desired flexibility. In what follows, we formalise the relevant concepts to define the cspio_q relation.

The quiescent behaviour of a process can be inferred and annotated using the *prioritise* feature in the FDR refinement checking tool. The notation $\text{prioritise}(P, \langle X_1, \dots, X_n \rangle)$ represents the process that defines priority between events. This process has a similar behaviour to P , but it prevents any event in X_i (where $i > 1$) from happening when there is a possibility of an internal event (τ), (\checkmark) termination, or an event in X_j (where $j < i$). We then define a process P_{qui} that captures the quiescence states of a process P as follows.

Definition 6. (*Quiescence Annotation*) Consider P is a CSP process whose output alphabet is A_O , and a special event qui that represents a quiescent behaviour. The process P_{qui} denotes the process P annotated with a qui-loop in the states where no event in A_O is offered. Formally,

$$P_{qui} \hat{=} \text{prioritise}(P \parallel \parallel \text{RUN}(\{qui\}), \langle A_O, \{qui\} \rangle)$$

where $\text{RUN}(s) = \square ev : s \bullet ev \rightarrow \text{RUN}(s)$ is a process that continuously offers the events from the set s . The basic intuition is that quiescence happens only when there is no output event. If there is an output event, it takes priority over quiescence, and therefore quiescence is prevented from happening.

The relation cspio_q establishes a conformance notion in the context of quiescent behaviour. Consider A_I and A_O the input and output alphabets of an I/O process, and $A_{O_{qui}} = A_O \cup \{qui\}$ with $qui \notin (A_O \cup A_I)$.

Definition 7. (cspio_q : *CSP input-output conformance with quiescence*) Consider $IUT_{qui} = (P_{IUT_{qui}}, A_{IUT}, A_{O_{IUT_{qui}}})$ an implementation model and $S_{qui} = (P_{S_{qui}}, A_{IS}, A_{O_{S_{qui}}})$ a specification, such that $A_{IS} \subseteq A_{IUT}$ and $A_{O_{S_{qui}}} \subseteq A_{O_{IUT_{qui}}}$ (compatible alphabets). Then

$$IUT \text{ cspio}_q S \hat{=} \forall s : \mathcal{T} (P_{S_{qui}}) \bullet \text{out}(IUT_{qui}, s) \subseteq \text{out}(S_{qui}, s)$$

In this relation, after every trace of the specification model, the output events observed in an implementation model (including quiescence) are a subset of the output events allowed by the specification.

3.2.2 Constructing Sound Test Cases

We now present the steps to generate sound test cases for cspio_q . Let TC , S and IUT be I/O processes that are models for the test case, the specification and the implementation, respectively. Additionally, the alphabet of the IUT is assumed to be compatible with the alphabet of the specification. A test case $TC = (P_{TC}, A_{ITC}, A_{OTC} \cup VER)$ generated from S to test IUT is an I/O process whose input events are in the set $A_{ITC} \subseteq A_{O_{IUT_{qui}}}$ and whose output events are in $A_{OTC} \subseteq A_{IS} \cup VER$, with $VER = \{pass, fail, inc\}$, such that $VER \cap (A_{IUT} \cup A_{O_{IUT}}) = \emptyset$.

The parallel composition $EXEC_{qui} = P_{IUT_{qui}} \parallel [A_{IUT} \cup A_{O_{IUT_{qui}}}] \parallel P_{TC}$ captures the execution of a test against an implementation with the observation of quiescence. Such a composition can result in the communication of a verdict event that defines the execution result: when a system behaves as expected, it behaves as $PASS = pass \rightarrow Stop$, meaning the test passes in the execution. Otherwise, if the system behaves as $FAIL = fail \rightarrow Stop$, the execution fails. Finally, if the system behaves as $INC = inc \rightarrow Stop$, the execution has an inconclusive verdict.

The refinement below verifies the presence of a verdict event $v \in VER$ in the traces of the CSP model for a test execution $EXEC_{qui}$. On the right-hand side of the refinement, input and output events are hidden from $EXEC_{qui}$, so only verdict events are communicated. If the refinement holds, the trace that represents the verdict (v) is present in the traces of the execution; otherwise, if the refinement does not hold, the verdict event will not be communicated and thus is not part of the traces of the test execution.

$$EXEC_{qui} \setminus (A_{IUT} \cup A_{O_{IUT_{qui}}}) \sqsubseteq_t v \rightarrow Stop$$

A generated test is said to be sound if, and only if, whenever the test fails in its execution, it is guaranteed that the *IUT* does not conform to the specification. In other words, the generated tests do not reject correct implementations. Definition 8 formalises soundness according to the \mathbf{cspio}_q theory.

Definition 8. (Sound test case). Let *IUT* be an implementation I/O process, *S* the specification, *TC* a test case I/O process and $EXEC_{qui}$ the execution of *TC* against *IUT* with the observation of quiescence. Then *TC* is a sound test case if the following holds.

$$\langle fail \rangle \in \mathcal{T}(EXEC_{qui} \setminus (A_{IUT} \cup A_{O_{IUT_{qui}}})) \Rightarrow \neg (IUT \mathbf{cspio}_q S)$$

The steps to build a process component of a test case $TC (P_{TC})$ in order to test the *IUT* are as follows. First, the output events offered by the specification after each event of the test case must be known. These outputs are recorded in an annotated trace (*atrace*) that is obtained by recording the outputs expected at the point each event of the test scenario (*ts*) is offered. Moreover, events and outputs are associated as $\langle (ev_1, outs_1), \dots, (ev_{\#ts}, outs_{\#ts}) \rangle$, with ev_i an event that belongs to *ts* and $outs_i$ being the outputs after the trace $\langle ev_1, \dots, ev_{i-1} \rangle$.

If $ev_i \in A_O$, then $outs_i = out(S, \langle ev_1, \dots, ev_{i-1} \rangle) - \{ev_i\}$, else $outs_i = \emptyset$, for $1 \leq i \leq \#ts$ and $\langle ev_1, \dots, ev_{i-1} \rangle$ is a prefix of ts .

Next, the function $TC_BUILDER$ uses the annotated trace ($atrace$) as a parameter, which recursively behaves like the process $SUBTC$ for each pair $(ev, outs)$. If the last element of a trace is reached, $TC_BUILDER$ yields the process $PASS$. The process $SUBTC$ is responsible for creating the body of a test and initially offers the event ev to the implementation and then behaves like $Skip$ to mark the successful termination of the process. The primitive $ANY(evset : \mathbb{P}_{\alpha_s}, next) = \square ev : evset \bullet ev \rightarrow next$, selects events from the specification's offered set, $evset$. If any of these chosen events is communicated, it proceeds according to the next action. If not, it results in deadlock.

$$TC_BUILDER(\langle \rangle) = PASS$$

$$TC_BUILDER(\langle (ev, outs) \rangle \frown tail) = SUBTC((ev, outs)); TC_BUILDER(tail)$$

where

$$SUBTC((ev, outs)) = ev \rightarrow Skip$$

$$\square (ev \in A_{O_S} \ \& \ ANY(outs, INC))$$

$$\square ANY(A_{O_{IUT_{qui}}} - outs, FAIL)$$

When ev is a test output (implementation input), it is communicated to the implementation, and the test fragment terminates successfully. Due to the input-enabled behaviour, the implementation is always ready to accept inputs. On the other hand, if ev is a test input (implementation output) and because the test cannot block implementation outputs, the process must be ready to synchronise with any output response of P_{IUT} . The test reaches an inconclusive verdict for the cases where an output event is communicated by the implementation ($P_{IUT_{qui}}$) that is not expected by the test scenario (ev) but is an output of the specification. Otherwise, the test reaches the verdict *fail* when P_{IUT} communicates an output event not in the specification or presents a quiescent behaviour.

A test cannot choose between input and output to avoid controllability conflicts Jard and Jéron [2005]. The process $TC_BUILDER$ does not allow such kind of choice thus, it is free of controllability conflicts. A test case yielded by $TC_BUILDER$ is sound according to

Theorem 1.

Theorem 1. (*TC_BUILDER is sound*). Let $S = (P_{S_{qui}}, A_{I_S}, A_{O_{S_{qui}}})$ be a specification, ts a test scenario from S and $IUT = (P_{IUT_{qui}}, A_{I_{IUT}}, A_{O_{IUT_{qui}}})$ an implementation model, such that $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. If $atrace$ is an annotated trace obtained from ts , then $TC = (TC_BUILDER(atrace), A_{I_{TC}}, A_{O_{TC}})$, with $A_{I_{TC}} = A_{O_{IUT_{qui}}}$ and $A_{O_{TC}} = A_{I_S}$, is a sound test case.

Proof sketch. Consider $\langle fail \rangle \in \mathcal{T}(EXEC_{qui} \setminus (A_{I_{IUT}} \cup A_{O_{IUT_{qui}}}))$, thus, by the definition of $EXEC_{qui}$, there is a trace $t \frown \langle fail \rangle$ that belongs to the traces of the following specification $P_{IUT_{qui}} \parallel [A_{I_{IUT}} \cup A_{O_{IUT_{qui}}}] TC_BUILDER(atrace, A_{I_{TC}}, A_{O_{TC}})$.

Moreover, from the definition of the parallel composition operator and the definition of the process $TC_BUILDER$, we have that t equals $s \frown \langle o \rangle$, such that s belongs to the traces of the processes $P_{IUT_{qui}}$ and to the traces of $P_{S_{qui}}$. Furthermore, the output o belongs to $A_{O_{IUT_{qui}}} - outs(S_{qui}, s)$. It implies the implementation process $P_{IUT_{qui}}$ produces a trace $s \frown \langle o \rangle$ where s belongs to the traces of $P_{S_{qui}}$, $o \in A_{O_{IUT_{qui}}}$, and $o \notin outs(S_{qui}, s)$. Consequently, o represents an output (or a quiescence) that belongs to $out(IUT_{qui}, s)$ and does not belong to $out(S_{qui}, s)$, which falsifies $IUT \text{ cspio}_q S$. \square

3.2.3 Test Suite Generation

In what follows, we present an example of the generation process in a real context. The Feature *Networking* is responsible for testing different mobile network settings, including carriers (Tim, Claro, Vivo, AT&T, Verizon, TMobile) and SIM configurations (SIM1/SIM2). Consider the scenario where the interaction of mobile components under $3G - SIM1$ network setting is being assessed. The possible interactions between those components are related to receiving/sending mobile voice calls and receiving/sending text messages (MMS/SMS).

Figure 12 shows the use case specification for $3G - SIM1$ network setting. This example models the concurrent scenario where multiple inputs (in sequence) are possible. The document follows a well defined template composed essentially by data elements (Types, Constants and State Variables) and the use case descriptions (Step ID, User Action, System State, System Response). The type OnOff determines whether a component is active (On) or not active (Off). The variable onM0VoiceCall stands for the status of a mobile originator (MO)

voice call, while `onMTVoiceCall` refers to the status of a mobile terminator (MT) voice call. Their initial value is off. The use cases depicted in Figure 12 exclusively present main flows. Each flow consists of a series of defined steps, comprising three elements: the user's input action, a system condition (the prerequisite for executing the step), and the corresponding system response. For instance, in the first step (1M) of UC01 a MO voice call is initiated. The use case template introduces a Controlled Natural Language (CNL) notation to delineate user input values (user action), conditional expressions (system state), and system outputs/updates (system response). These components are encapsulated between the % symbol. For instance, the representation of a successfully connected MO voice call is given in the System Response by the expression `%onMOVoiceCall:=On%`.

Data Definition

NewType	Description	Value
OnOff	Determines whether the component is active or not active	On, Off

Variable	Description	VarType	Value
onMOVoiceCall	MO Voice Call status	OnOff	Off
onMTVoiceCall	MT Voice Call status	OnOff	Off

UC01 - Verify 3G-SIM-1 behavior [MO Voice Call]

Step ID	User Action	System State	System Response
1M	Make MO voice call from SIM1 and wait until it gets connected.	<code>%onMTVoiceCall == Off%</code>	<code>%onMOVoiceCall := On%</code>
2M	End the Voice Call.		<code>%onMOVoiceCall := Off%</code>

UC02 - Verify 3G-SIM-1 behavior [MT voice call]

Step ID	User Action	System State	System Response
1M	Make MT voice call from SIM1 and wait until it gets connected.	<code>%onMOVoiceCall == Off%</code>	<code>%onMTVoiceCall := On%</code>
2M	End the voice call.		<code>%onMTVoiceCall := Off%</code>

UC03 - Verify 3G-SIM-1 behavior [MT SMS]

Step ID	User Action	System State	System Response
1M	While call is active send MT SMS on SIM1.	<code>%onMOVoiceCall == On%</code>	MT SMS is successful while the call is active.

UC04 - Verify 3G-SIM-1 behavior [MO SMS]

Step ID	User Action	System State	System Response
1M	While call is active send MO SMS on SIM1.	<code>%onMTVoiceCall == On%</code>	MO SMS is successful while the Voice Call is active.

Figure 12 – 3G-SIM1 Use Case

In order to build a sound test case ($TC_BUILDER(atriace)$) we adapted the algorithm in Nogueira et al. [2014] to consider quiescence. We first extract a test scenario performing refinement verification using the CSP process that is the formal specification for the use case model, which is the input to construct an annotated trace (*atriace*). For this example, consider the CSP process $F1_UC1$ that represents the formal semantics of the use case model.

We demonstrate the test generation strategy using the process $F1_UC1$ following the approach presented in Subsection 2.3.3. The modified version of $F1_UC1$, which incorporates the *accept* event following the traces culminating in successful termination, is $F1_UC1; accept \rightarrow STOP$. FDR runs a refinement verification and provides the shortest counter-example trace if the refinement is not satisfied. Hence, the FDR assertion

$$F1_UC1 \sqsubseteq_t F1_UC1; \text{ accept} \rightarrow STOP$$

does not hold and yields the following counterexample trace:

$$ce_1 = \langle \text{receiveMTVoiceCall}, \text{sendMOSMS}, \text{moSMSSuc}, \text{endMTVoiceCall}, \\ \text{initiateMOVoiceCall}, \text{sendMTSMS}, \text{mtSMSSuc}, \text{endMOVoiceCall}, \text{accept} \rangle$$

We illustrate the strategy using the shortest counterexample extracted from the process $F1_UC1$. Following the test generation process of TaRGeT, the test scenario $F1_UC1_{ts_1}$ is obtained from ce_1 by removing the control event (accept). Such an event is used only for test generation and is not in the alphabet of the use case model ($F1_UC1$).

$$F1_UC1_{ts_1} = \langle \text{receiveMTVoiceCall}, \text{sendMOSMS}, \text{moSMSSuc}, \text{endMTVoiceCall}, \\ \text{initiateMOVoiceCall}, \text{sendMTSMS}, \text{mtSMSSuc}, \text{endMOVoiceCall} \rangle$$

When the specification requires an output and none is produced, the output is represented as an empty set (\emptyset) to indicate quiescence. Otherwise, $outs$ is populated with the expected outputs. The process to obtain an annotated trace is detailed in Nogueira et al. [2014]. The following annotated trace is obtained from $F1_UC1_{ts_1}$

$$atrace_{ts1} = \langle (\text{receiveMTVoiceCall}, \emptyset), (\text{sendMOSMS}, \emptyset), \\ (\text{moSMSSuc}, \{\text{moSMSSuc}\}), (\text{endMTVoiceCall}, \emptyset), (\text{initiateMOVoiceCall}, \emptyset), \\ (\text{sendMTSMS}, \emptyset), (\text{mtSMSSuc}, \{\text{mtSMSSuc}\}), (\text{endMOVoiceCall}, \emptyset) \rangle$$

Next, once an annotated trace is obtained from ts , one can build a sound test case $TC1 = TC_BUILDER(atrace_{ts1}, A_{ITC}, A_{OTC})$.

In the context of this work, a quiescence occurs due to the absence of outputs. It means that, whenever the specification is expecting an output and a quiescence occurs, this leads to a fail verdict.

Quiescence never leads to pass or inconclusive verdicts. In the $TC_BUILDER$ process, the set $outs$ may include qui , and the event ev can also behave as a quiescence.

We present an example of a test case ($TC1$) designed to verify the interaction between two components: *Calling* and *Message*. We illustrate this interaction in the form of a se-

quence diagram. As illustrated in Figure 13, the interaction sequence begins with the input action *makeCallTo()* directed to the Calling component. The absence of an output event in response to this input is properly captured by a quiescent state. Next, an additional input action, *sendSMSTo()*, is directed at the Message component, while the call is still active. Only then the first output, *success*, is observed, indicating a successful interaction between the components and confirming that a message was received while a call was active. Finally, the input action *endCallTo()* is triggered, followed by an output event, *success*, meaning that the call successfully terminated. The subsequent actions consist of a symmetric variation of the previous scenario where the *Calling* component initiates a call with another device, rather than receiving one, and the *Message* component sends a message to this same device.

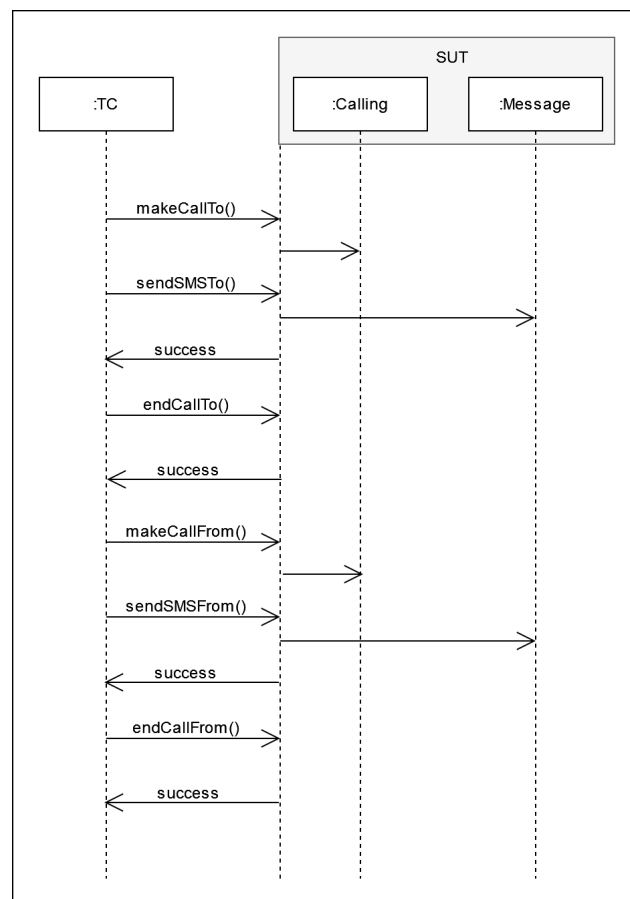


Figure 13 – Example interaction of TC1 with SUT.

As already mentioned, the original theory in Nogueira et al. [2014] is not able to handle more than one input or more than one output in sequence; it assumes an alternation of inputs and outputs that is too restrictive in some contexts, including that of testing mobile features.

4 OPTIMISED TEST GENERATION STRATEGY

This chapter presents an optimisation of the proposed approach, taking into account the practical context in which this research is conducted.

In accordance with the Reverse Engineering approach outlined in Chapter 3, use cases were derived from pre-existing test cases. These use cases become subsequently input for generating test cases for concurrent features. Reverse engineering was applied due to the unavailability of use cases. Although this procedure has proved effective, the reverse engineering process is complex due to the need to generate an input at a higher level of abstraction (use cases) from more concrete artifacts (test cases). Also, the correspondence is not one-to-one: a use case is normally generated from the combination of several test cases.

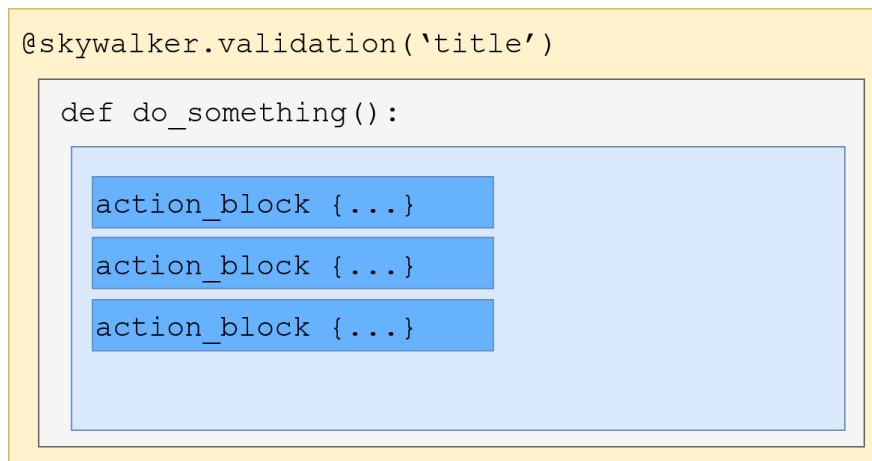
Aiming at a more efficient alternative that is closer to our industrial partner's real automation and execution environment, a direct extraction approach is presented in this chapter. This involves the generation of test cases for concurrent features by combining existing test cases for individual features.

The upcoming sections showcase work that took place in the actual Motorola development environment. This setting allowed for the proposed approach's scope to be optimised, encompassing the various code patterns found in the test case scripts and addressing different development requirements. For instance, the original code was written using various structures, prompting us to classify them into different levels of granularity that may vary according to the development team and/or degree of automation of the scripts. We categorised three different levels that range from a coarse granularity level to a fine granularity. The correlation between these levels is highlighted in Figure 14 using three different colours. The coarse level is denoted validation (yellow colour), the intermediate level comprises functions inside the validation (grey colour), and the fine level contains blocks of actions (blue colour).

In the coarse level of granularity, the script is designed such that the validation is defined in terms of a single function. A validation is defined as a thorough Python script document that encapsulates a specific action, such as enabling Wi-Fi. In this way, the validation itself is used to perform interleavings, without considering their possible internal compositions.

In some scenarios, permuting a validation as a whole is enough to generate paths that lead to new bugs. In other cases, exploring interleaves of finer grain units is necessary to make the interleaving process more flexible and generate new possibilities for interactions. In that

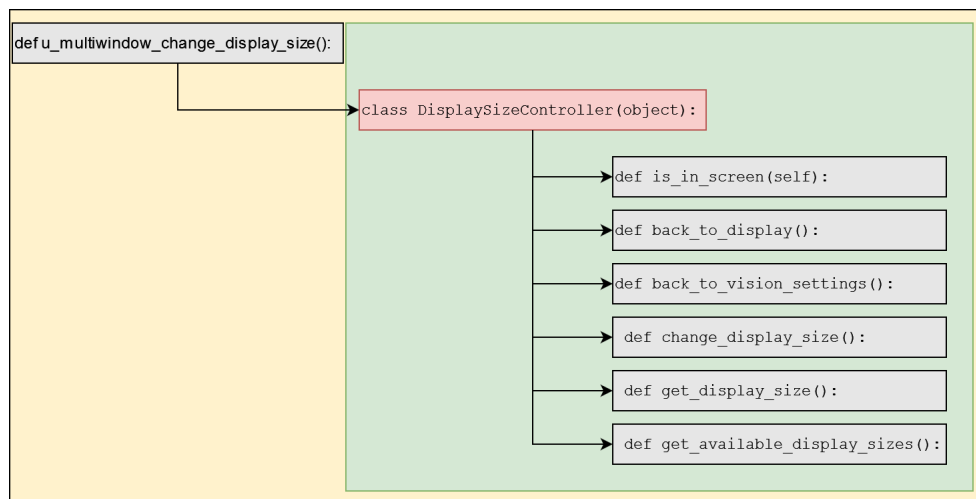
Figure 14 – Overview of Atom Granularity Levels



Source: The author (2024)

sense, we allow the possibility of permuting functions within a validation. Highlighted in grey in Figure 15, the functions within the validation `u_multiwindow_change_display_size` may be permuted with functions from other validations. In what follows, this behaviour is portrayed, where the fictitious test cases `mca_01` and `mca_02` (Figure 16), derived a combination of the functions presented in each test case (Figure 17).

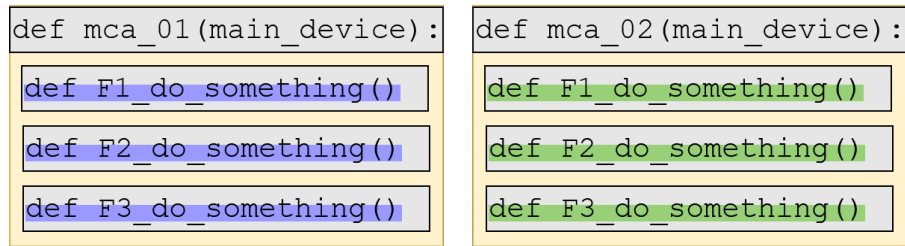
Figure 15 – Atom Granularity (Function Level)



Source: The author (2024)

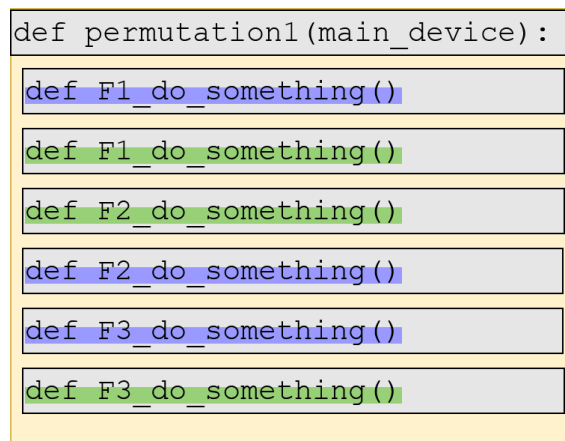
The finer granularity occurs at the action block level. An action block is represented by a piece of code which contains a valid behaviour to be interpreted as an atom. Figure 18 correspond to the test script `mca_371293`, which is composed of small snippets of code. In this example, a valid action block (atom) is composed by a pre-setup, which is not mandatory, followed by:

Figure 16 – Original test cases sample, mca_01 and mca_02.



Source: The author (2024)

Figure 17 – Interleaving of mca_01 and mca_02.

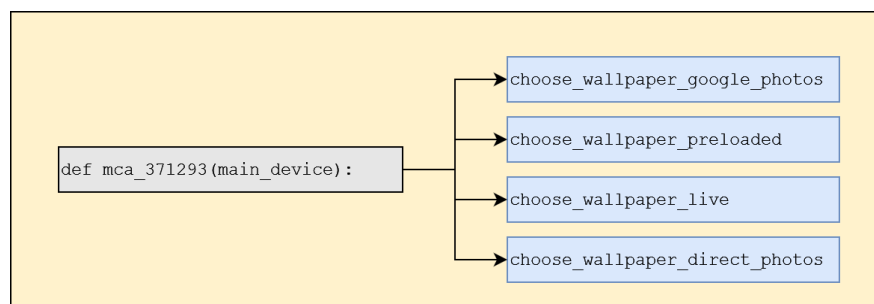


Source: The author (2024)

- A log describing the action,
- The actual executable code, and,
- An assertion to verify the correctness of the execution according to the expected result.

Due to privacy concerns related to Motorola's proprietary code, the code snippet corresponding to those action blocks has been omitted from this representation.

Figure 18 – Atom Granularity, highlighted in blue (Action Block Level)



Source: The author (2024)

Each composition represents an action block (atom) and therefore those blocks are the testable units used to perform the interleavings. There is no programmatic/syntactic built-in delimiter that encapsulates an action block, so this level of interleaving becomes more challenging to apply automatically.

4.1 TEST GENERATION VIA INTERLEAVING OF ATOMS

We illustrate how atoms are extracted and interleaved to generate tests.

The atom extraction process is given by the syntactical identification of one (or more) actions within a step of a test case. An action, commonly expressed using a verb, denotes an act or occurrence of a fact. Later in the generation process (Figure 11), the extracted atoms are interleaved to generate tests with multiple execution possibilities. In what follows, we highlight an example that demonstrates the atom extraction process.

Consider the test cases Managing contact (TC01) and Performing calls (TC02) in Tables 8 and 9. Each step in both tables gives rise to a single atom. Nevertheless, some steps may be formed of more than one imperative sentence connected by conjunction (and), giving rise to more than one atom.

Table 8 – TC01.

Test Step ID	Test Step Description (input)	Expected Results (output)
1M	Add a contact.	The contact is successfully added.
2M	Edit a contact.	The contact is successfully edited.
3M	Remove a contact.	The contact is successfully removed.

Source: The author (2023)

Table 9 – TC02.

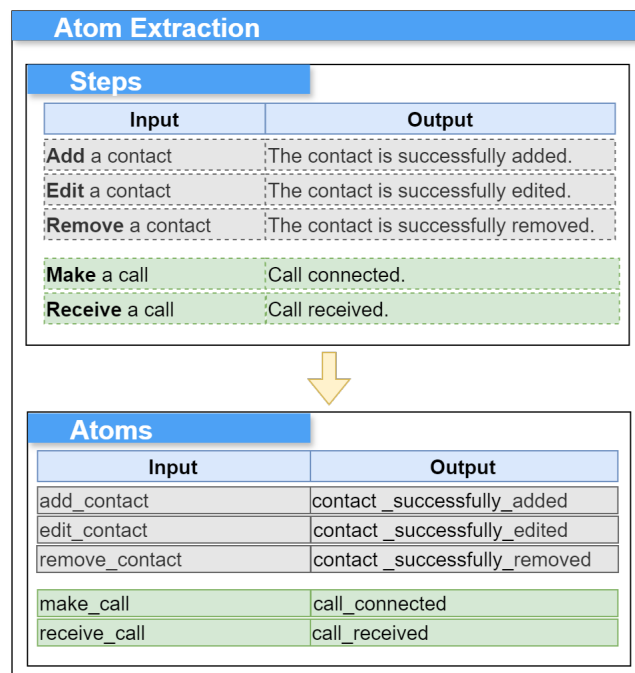
Test Step ID	Test Step Description (input)	Expected Results (output)
1M	Make a call.	Call connected.
2M	Receive a call.	Call received.

Source: The author (2023)

From Tables 8 and 9, we obtain five atoms (as shown in Figure 19). Each atom is represented by an identifier that abbreviates the corresponding text. For instance, add_contact is

the identifier for *Add a contact*.

Figure 19 – Atom Extraction



Source: The author (2024)

Considering an automated procedure, the identification of the step actions is performed using syntactic analysis as part of a natural language processing technique where each word is classified according to its grammatical function. The words that denote a behaviour (verbs) guide the construction of atoms.

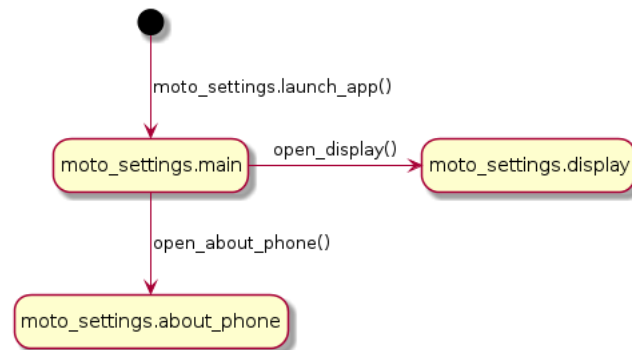
After identifying the atoms, Motorola's script database is checked to verify the presence of automation scripts that match the highlighted behaviours for each atom. In cases where the atom and the test case script do not align, the developer is alerted, indicating the lack of codependency. To address this, it is necessary to provide a corresponding script using a tool called The Force, a Python framework designed by Motorola to provide an abstraction over Android devices, applications, and screens, aiming to support the development of test automation through the Page Object design pattern.

The Force consists of controllers, which are the entities representing the "objects" in the Page Object pattern. Typically, there are two categories of controllers: Screen-bound controllers and Non-screen-bound controllers. Screen-bound controllers possess a window attribute and represent tangible screens within an Android application, such as the Settings Main screen or the Settings Display screen. On the other hand, Non-screen-bound controllers lack a window

attribute and symbolise conceptual elements within a device, like the entire Settings Application.

One of the improvements offered by The Force is automated navigation, which involves the establishment of pathways between controllers and the creation of an internal navigation graph (refer to Figure 20).

Figure 20 – Force Controller for Moto Settings Feature



Source: The Force (2018)

Using the control behaviour defined as in this graph, when a test invokes any actions within the `moto_settings.display` controller, one of the following scenarios occurs:

- If the device is already on the Display screen, the action is executed promptly.
- If the device is on a different screen, the Force initiates automated navigation to the Display screen and subsequently executes the intended action.

Tables 10 and 11 present the traceability between the test, its automation script (The Force code), and the respective atoms. For the Managing contact test case the following automated actions can be observed: `contacts.add_contact`, `contacts.edit_contact`, `contacts.remove_contact`.

Table 10 – Managing contact [Automation Traceability]

Test Step ID	Validation [Automation Script]	Atoms	Atom ID
TC01_1M	<code>contacts.add_contact</code>	<code>add_contact</code>	TC01_1M_01
TC01_2M	<code>contacts.edit_contact</code>	<code>edit_contact</code>	TC01_2M_02
TC01_3M	<code>contacts.remove_contact</code>	<code>remove_contact</code>	TC01_3M_03

Source: The author (2024)

Table 11 – Performing Calls [Automation Traceability]

Test Step ID	Validation [Automation Script]	Atoms	Atom ID
TC02_1M	calls.make_call	make_calls	TC02_1M_01
TC02_2M	calls.receive_call	receive_calls	TC02_2M_02

Source: The author (2024)

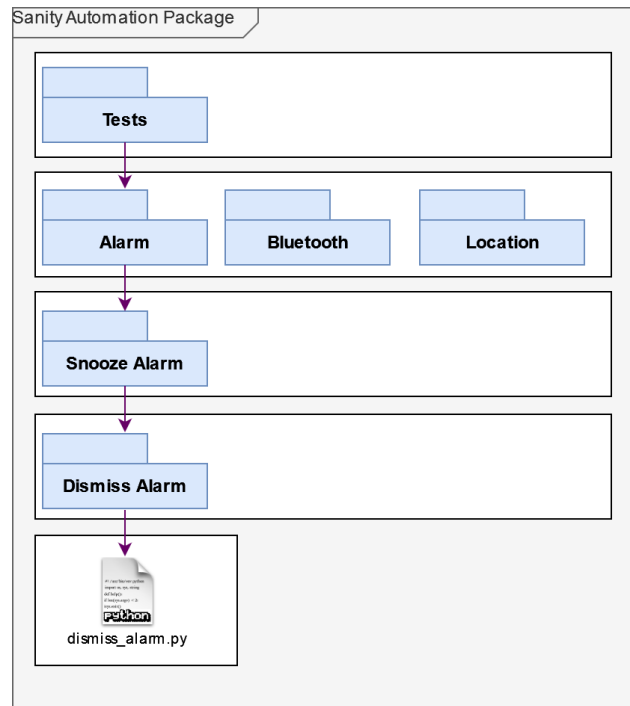
Moreover, the traceability tables introduce the concept of “validation” for each atom. This concept is related to the development environment of Motorola’s automation scripts. Since the approach aims at the total (or partial automation) of the test case generation and execution, the integration with Motorola’s environment and existing tools is crucial.

Motorola’s automation scripts are developed using Python and are supported by a variety of proprietary solutions/tools for test automation/execution. The development process is guided by a standard model and a pattern. One of the standards designed by Motorola follows a tree structure (Uneti) in which each action of a test case corresponds to a validation. The validations are extracted from the respective TCs and amalgamated into a single validation hierarchy, arranged in an optimal sequence. In order to reach every validation, one must conduct a depth-first search in the test case until it reaches the smallest units, as shown in Figure 21. For instance, to access the validation related to dismiss the alarm function in the Sanity’s Suite Automation Package, a depth traversing is applied to the packages alarm, snooze alarm and dismiss alarm.

This standard model was originally used to facilitate the reuse of context along the various packages and circumvent the need for any preliminary setup procedures in the future. As a result, the validation of the `home.go_to_home` function, for instance, can be applied to multiple tests that are related to this particular action.

Concerning the pattern previously mentioned, another development pattern defines that for each automated test, there is a respective Python file containing all the code (or the main code) necessary to run the test case. Although this structure does not offer the benefit of eliminating the redundancy of identical code in multiple tests that share the same functions, it does provide the advantage of flexibility in making any necessary modifications to a specific test, as it consolidates all the test code into a single file. This alternative code structure enables more direct manipulation of the code, facilitating the extraction of atoms in a more accurate and efficient manner. It takes into account the fact that the document itself can already serve as a potential validation. Additionally, it allows for the exploration of other levels within the

Figure 21 – Sanity Automation Package (Uneti Structure)



Source: The author (2024)

document, as mentioned earlier in this chapter.

Once the atoms have been extracted and the existence of compatible automatic scripts have been carried out, the dependency check between the atoms takes place.

The interleaving works by shuffling the atoms, creating all possible execution flows, as long as they make sense according to the dependency restrictions established in the Dependency Analysis step. Figure 22 illustrates some possible interleavings that can be generated from the test cases Managing Contacts (gray) and Performing Calls (green). Each rectangle represents an atom and for each new generated interleaving the order of the atoms is different, symbolising various possible paths. For instance, iTC_001, involves an interaction between atoms from both test cases, where their paths intersect.

Figure 22 – Interleaving of atoms

Interleaving	
ITC_001	
Input	Output
add_contact	contact_successfully_added
make_call	call_connected
edit_contact	contact_successfully_edited
remove_contact	contact_successfully_removed
receive_call	call_received
ITC_002	
Input	Output
make_call	call_connected
add_contact	contact_successfully_added
edit_contact	contact_successfully_edited
receive_call	call_received
remove_contact	contact_successfully_removed
ITC_003	
Input	Output
add_contact	contact_successfully_added
make_call	call_connected
edit_contact	contact_successfully_edited
receive_call	call_received
remove_contact	contact_successfully_removed
...	

Source: The author (2023)

4.2 SOUNDNESS OF THE OPTIMISED APPROACH

In Chapter 3 we have addressed the soundness of the proposed approach to test case generation. Recall that soundness establishes that if the execution of a generated test case results in a fail verdict, then one can assure that the implementation under test does not conform to the specification. In other words, the uncovered failure is an implementation problem, not a problem with the generated test itself.

The purpose of this section is to show that test cases generated by the optimised approach presented in this chapter preserve soundness. Let us start by recalling the test generation flow presented in Chapter 3 and repeated here (for convenience) as the top level flow in Figure 11 (see Chapter 3). The other flow captured by this figure involves two steps (at the bottom of the figure, highlighted in grey) and represents the optimised approach. Note that it differs from the original approach just by the steps Atom_Extraction and Interleaving replacing the steps UC_generation and TC_Gen_TaRGeT.

In what follows, we formalise the atom extraction process and the interleaving of atoms. Let $TCs = \{ts_1, ts_2, \dots, ts_M\}$ be a set of test scenarios ts_x for $1 \leq x \leq M$. Considering the use case format presented in Chapter 2, M is an even number because a test interchanges input and output events, and each ts_x has the form $\langle i_1, o_1, \dots, i_K, o_K \rangle$. Each subsequence $\langle i_x, o_x \rangle$ represents a test step. In addition, let *extract* be a function that takes as input a test scenario ts and yields a sequence of atoms extracted from the test. Each sequence of atoms has the form $\langle i_1, o_1, \dots, i_N, o_N \rangle$, where N is the number of atoms. If there is exactly one atom for each test step, $N = K$ and the sequence of atoms coincides with the test scenario; otherwise, N is greater than K . In addition, consider the function *extraction* that takes as input a set of test scenarios and yields a set of sequences of atoms; formally:

$$extraction(TCs) = \bigcup \{extract(ts) \mid ts \in TCs\}$$

Using the proposed semantics, the test scenarios illustrated in Tables 8 and 9 can be represented by the set $TCs_1 = \{ts_1, ts_2\}$, such that

$$\begin{aligned} ts_1 &= \langle add_contact_i, add_contact_o, edit_contact_i, edit_contact_o, \\ &\quad remove_contact_i, remove_contact_o \rangle \\ ts_2 &= \langle make_call_i, make_call_o, receive_call_i, receive_call_o \rangle \end{aligned}$$

Moreover, the expression $extraction(TCs_1)$ yields a set of sequence of atoms.

We now present the algorithm for the optimised strategy, as depicted in Algorithm 1. Both the algorithm's input and output are a sequence of sequences of atoms. Each sequence in the input represents a sequence of atoms, and each sequence in the output results from the interleaving of the input sequences. The output is initially the empty sequence (Line 1). Then, the procedure *INTER* (Line 4) is called with two arguments: the algorithm input and an empty sequence. The algorithm returns the value for result produced by *INTER*. The argument prefix (initially empty) is a sequence of atoms used to construct the interleaved sequences recursively. The base case happens when all the sequences that belong to sequences are empty (Line 5). If they are, the interleaving stored in prefix contains a complete interleaving with atoms for all the sequences. The procedure appends the current prefix to the overall result list (Line 6). Otherwise, if there is some element to include in the interleaving, it loops through each sequence (Line 8), and if the sequence has at least two elements (Line 9), it stores the input and output of the atom in the head of the sequence in *in* and *out* (Lines 10 and 11), removes such elements from the sequence, and performs a recursive call with the updated

sequences and prefix (Lines 12 and 13). This recursive approach ensures all interleavings are generated and stored in the result.

Algorithm 1 Interleaving atoms

Input: sequences: $seq(seq(A_{IUT} \cup A_{O_{IUT}}))$
Output: result: $seq(seq(A_{IUT} \cup A_{O_{IUT}}))$

```

1: result  $\leftarrow \langle \rangle$ 
2: INTER(sequences,  $\langle \rangle$ )
3: return result
4: procedure INTER(seqs:  $seq(seq(A_{IUT} \cup A_{O_{IUT}}))$ , prefix:  $seq(A_{IUT} \cup A_{O_{IUT}})$ )
5:   if all_empty(seqs) then
6:     result  $\leftarrow$  append(result, prefix)
7:   else
8:     for i  $\leftarrow$  0 to length(seqs) -1 do
9:       if length(seqs[i]) > 1 then
10:        in  $\leftarrow$  seqs[i].[0]
11:        out  $\leftarrow$  seqs[i].[1]
12:        seqs[i] = sublist(seqs[i], 2)
13:        INTER(seqs, prefix  $\frown \langle in, out \rangle$ )
14:      end if
15:    end for
16:  end if
17: end procedure

```

The procedure in Algorithm 1 contrasts with the test scenario generation using the original (non-optimised) strategy. The latter is based on refinement verifications, each one performed to yield an interleaving of the test scenarios. As reported in Nogueira et al. [2019], the refinement expression used for such a scenario generation is PTIME-hard on the sum of the number of states of the operational model (LTS) for the specification and the implementation. Using the optimised algorithm avoids constructing the complete state space of the use case model and the use of refinement checking using FDR.

The soundness of the optimised strategy is characterised using the function *intl* that takes as input and outputs a set of sequences of atoms. The definition of this function uses some additional notation. We use $seqs(A)$ to represent the set of sequences formed of elements from the set A , such that each sequence does not repeat elements. The notation $set(s)$ represents the set with the elements of the sequence s . Let $Satoms$ be a set of sequences of atoms. Then, $intl(Satoms)$ is defined as $set(interleave(seq_atoms))$, where $seq_atoms \in seqs(Satoms)$ and *interleave* is the function that computes Algorithm 1.

The call to $intl(TCs_1)$ yields a set with all the interleavings of sequences of atoms illustrated in Figure 19. Such a set has ten sequences of interleavings of the input atoms, including

the three sequences depicted in Figure 22. We partially present the content of this set.

$$\begin{aligned} & \{ \langle \text{add_contact}_i, \text{add_contact}_o, \text{make_call}_i, \text{make_call}_o, \\ & \quad \text{edit_contact}_i, \text{edit_contact}_o, \text{remove_contact}_i, \text{remove_contact}_o, \\ & \quad \text{receive_call}_i, \text{receive_call}_o \rangle \\ & \langle \text{make_call}_i, \text{make_call}_o, \text{add_contact}_i, \text{add_contact}_o, \\ & \quad \text{edit_contact}_i, \text{edit_contact}_o, \text{receive_call}_i, \text{receive_call}_o, \\ & \quad \text{remove_contact}_i, \text{remove_contact}_o \rangle \\ & \langle \text{add_contact}_i, \text{add_contact}_o, \text{make_call}_i, \text{make_call}_o, \\ & \quad \text{edit_contact}_i, \text{edit_contact}_o, \text{receive_call}_i, \text{receive_call}_o, \\ & \quad \text{remove_contact}_i, \text{remove_contact}_o \rangle \\ & \langle \text{add_contact}_i, \text{add_contact}_o, \text{edit_contact}_i, \text{edit_contact}_o, \\ & \quad \text{remove_contact}_i, \text{remove_contact}_o, \text{make_call}_i, \text{make_call}_o, \\ & \quad \text{receive_call}_i, \text{receive_call}_o \rangle \\ & \langle \text{add_contact}_i, \text{add_contact}_o, \text{edit_contact}_i, \text{edit_contact}_o, \\ & \quad \text{make_call}_i, \text{make_call}_o, \text{remove_contact}_i, \text{remove_contact}_o, \\ & \quad \text{receive_call}_i, \text{receive_call}_o \rangle, \dots \} \end{aligned}$$

Theorem 2 establishes that the tests produced by the optimised approach are equivalent to those produced by the original approach that creates an explicit use case model. A corollary of this theorem is that the optimised approach yields sound test cases.

Consider the function *uc_gen* that takes as input a set of test scenarios and outputs a CSP process that combines the behaviour of the input test scenarios. Such a process captures the semantics of concurrent use cases in the format of the TaRGeT tool. The function *tc_gen* inputs the use case model specified in CSP and outputs the set of test scenarios that combine the input test cases using the TaRGeT test generation approach.

Theorem 2. (*The optimised approach is sound*)

Let *TCs* be a set of test scenarios whose steps contain a unique atom each, then $\forall \text{ } TCs \bullet$
 $tc_gen(uc_gen(TCs)) = intl(extraction(TCs)).$

The justification is as follows. The use cases obtained by *uc_gen* combine the behaviour of the original test steps. These are input to the TaRGeT tool (function *tc_gen*) that extracts test scenarios to generate new test scenarios. The sequences generated by TaRGeT are inter-

leavings of the sequences of atoms extracted from these test scenarios using the optimised approach.

Proof sketch. The proof shows that the proposition's left-hand side (LHS) equals the right-hand side (RHS).

We start with the LHS expression $tc_gen(uc_gen(TCs))$.

As presented in Section 2.3, the semantics of $uc_gen(TCs)$ is equivalent to the CSP process

$$\begin{aligned}
 & ((s \rightarrow i_{1_1} \rightarrow o_{1_1} \rightarrow e \rightarrow \dots \rightarrow s \rightarrow i_{(\#(ts_1)/2)_1} \rightarrow o_{(\#(ts_1)/2)_1} \rightarrow e \rightarrow Skip \\
 & ||| s \rightarrow i_{1_2} \rightarrow o_{2_2} \rightarrow e \rightarrow \dots \rightarrow s \rightarrow i_{(\#(ts_2)/2)_2} \rightarrow o_{(\#(ts_2)/2)_2} \rightarrow e \rightarrow Skip \\
 & \dots \\
 & ||| s \rightarrow i_{1_M} \rightarrow o_{2_M} \rightarrow e \rightarrow \dots \rightarrow s \rightarrow i_{(\#(ts_M)/2)_M} \rightarrow o_{(\#(ts_M)/2)_M} \rightarrow e \rightarrow Skip) \\
 & |[\{s, e\}] | ((\lambda S. s \rightarrow e \rightarrow S) \triangle Skip) \setminus \{s, e\}
 \end{aligned}$$

Such a process formalises the combination of concurrent use cases in the format of TaRGeT tool. For the sake of simplicity, we abstract the existence of the memory process introduced in Section 2.3, without loosing precision. Each process in the interleaving captures the semantics of a use case whose flow originates from a test scenario in TCs , following a reverse engineering process. The special events e and s control the atomicity of a pair of inputs and respective outputs. Such events are an abbreviation to the events `startStep` and `endStep`, introduced in Section 2.3. The recursive process S (an abbreviation to the process `stepCR` presented in Section 2.3) is composed in parallel with the interleaved processes synchronising in the control events to ensure each new step only initiates after a previous step has concluded. The interruption ensures the parallel composition terminates with success. Finally, the control events are hidden and not visible on the traces. Due to such semantics, the properties of distributed termination and the interleaving semantics of the CSP parallel composition operators (see Chapter 2), $uc_gen(TCs)$ behaves as *Skip* only after all the use case flows have terminated; this guarantees when termination is observed, the steps of the scenarios have been interleaved. Moreover, every trace in $\mathcal{T}(uc_gen(TCs))$ that ends with \checkmark is a maximum trace.

The function tc_gen abstracts the TaRGeT test scenario generation presented in Section 3.2.3. Considering the behaviour of $uc_gen(TCs)$ and the refinement checking expression verified using FDR, the TaRGeT test generation strategy yields all traces t such that $(t \frown \checkmark) \in \mathcal{T}(uc_gen(TCs))$. Consequently, the test generation combines the input test sce-

narios using interleaving semantics to yield traces that combine the input test scenarios, keeping the order of the input and the output of each step.

The RHS has the expression $intl(extraction(TCs))$. Because there is a unique atom for each step of the test scenarios in TCs , the *extraction* function behaves as an identity function and the expression $extraction(TCs)$ yields TCs . Thus, RHS is equivalent to $intl(TCs)$. As explained, the function *intl* interleaves the sequences of atoms extracted from the test scenarios. Moreover, the order of the input and output of the atoms does not change during the combination. In this way, the sequences of atoms combined by the Algorithm 1 equals the traces yield by the FDR tool for the same input set of test scenarios.

Consequently, LHS equals RHS. □

With this result, we emphasise that it is possible to build sound test cases directly from test scenarios. Since the optimised strategy yields the same test scenarios as the original approach, it yields sound test cases.

4.3 OPTIMISED TEST GENERATION COST ANALYSIS

In this section, we provide a brief discussion on the time cost (PTIME) of automatically generating sound test cases. We offer a preliminary comparison of the computational resources required to generate a test suite using both the original and optimized approaches. A more in-depth mathematical analysis will be conducted in future work.

The original test case generation approach, based on Nogueira et al. [2014], uses the FDR tool to verify trace refinement by comparing the traces. If the implementation enables an event not allowed by the specification, the refinement fails, producing a counter-example trace. The approach's complexity arises from the normalisation of transition systems, which ensures every trace reaches a final state and minimises false counter-examples. However, both normalisation and refinement checking are computationally expensive—PSPACE-hard for trace refinement and EXPTIME-hard for normalisation—making test case generation resource-intensive, with costs depending on the size of the specification and test scenarios. The total effort grows with the number of iterations and the average size of each test case.

The overall complexity of the original approach is determined by the size of the specification and implementation state spaces, the average length of generated test cases, and the number of iterations. The cost of a single iteration is approximately:

$$norm(System) + |System|^2 + (\#ts \times (norm(System) + (|System| \times \#ts)))$$

where $\#ts$ is the size of the test scenario.

The optimised approach instead uses an interleaving-based algorithm to generate test cases, where the input is a sequence of atomic actions rather than states and the output is a sequence of interleavings, which represents different orderings of the atomic actions. The Algorithm 1 is relatively efficient as it recursively generates interleavings by selecting actions from the input sequences and appending them to a prefix. The complexity depends mainly on the length of sequences and the branching factor of possible interleavings at each step (i.e., how many different orderings can be formed). In this case, the time complexity is exponential, reflecting the exponential growth in the number of valid input-output combinations as the sequence length (T) increases.

While both the original and optimised approaches share an exponential complexity, the original approach suffers significantly more in terms of computational cost due to several additional factors. Specifically, the original approach not only computes interleavings once but does so repeatedly, processing each sequence length (N) individually. Additionally, it incurs further computational cost due to the normalization process applied to each interleaving computation, amplifying its overall complexity.

To assess the time complexity (EXPTIME) for generating test cases under the original and optimised approaches, we'll analyse the cost functions and principal computational factors for each bellow.

Original Approach: EXPTIME Complexity with Normalization

In the original approach, calculating the traces refinement entails several costly operations:

- Normalization ($norm(System)$): This step, essential for avoiding false refinement counterexamples, involves an EXPTIME-hard process. It requires behavioral equivalence checks (bi-simulation) to ensure that the normalised LTS mirrors the behavior of the original system.
- Refinement Verification ($comp(N(System), Q)$): The traces refinement is verified by comparing states between the implementation and specification using a breadth-first

search across the cartesian product of their LTSs, leading to an approximate complexity of $|P| \times |Q|$, where $|P|$ and $|Q|$ denote the number of states in each LTS.

Since this verification must be repeated N times (once for each test case in the suite), the overall complexity is approximately:

$$N \times (norm(System) + |System|^2 + mean(T) \times (norm(System) + |System| \times mean(T)))$$

The $|System|^2$ term reflects the complexity of exploring the cartesian product during refinement verification. Repeating this for every N requires system normalisation and refinement recalculation, escalating the complexity.

Optimized Approach: Interleaving Algorithm Complexity

As discussed in Clarke et al. [1999], verifying interleavings can lead to exponential complexity due to the large number of state combinations, especially in concurrent and parallel processes. Additionally, according to Cormen et al. [2009], many combinatorial problems, such as those solved by backtracking and dynamic programming, require examining all subsets or possibilities, resulting in exponential time complexity. These algorithms often involve exploring all possible interleavings of tasks, leading to a time complexity of $O(2^n)$, where n is the number of tasks to be interleaved.

The optimised approach reduces test generation complexity by using an interleaving algorithm that works with sequences rather than states. The algorithm interleaves input sequences, with complexity primarily based on the sequence count and their average length T . Thus, it has a complexity of approximately $O(2^T)$, where T represents the average sequence length. Although still exponential, it avoids system normalisation and Cartesian product computation, significantly reducing computational cost compared to the original approach.

We can relate the state count in the original approach to the sequence length in the optimised approach. The number of states $|System|$ approximately equates to T^N , indicating that the original approach's state-based method implicitly deals with an exponential function of the sequence length. Therefore, the original approach holds a global exponential complexity of approximately $O(T^2N)$ accounting for normalization and refinement verification for each N . On the other hand, the optimised approach holds a complexity of $O(2^T)$, which is more efficient

as it avoids normalization and works directly with sequences. In that sense, although both approaches are exponential in nature, the optimized approach is substantially more efficient for high values of N and T as it avoids the accumulated cost of normalization and the state Cartesian product calculation.

5 TOOL SUPPORT

This Chapter addresses tool support, discussing the tool architecture (Section 5.2), the tool workflow (Section 5.3), and the integration process with a dependency analysis tool, Kaki (Section 5.4).

5.1 TOOL'S OVERVIEW AND DEVELOPMENT STACK

This proprietary tool is designed to support Motorola's testing engineers by providing an integrated solution for creating test cases tailored to Motorola's development environment. Aimed at enhancing testing efficiency and accuracy, the tool is optimised for professionals managing complex test scenarios that involve multiple feature interactions.

The tool employs contemporary web and software development frameworks and languages, selected strategically to ensure scalability, reliability, and optimal performance. The front end is developed using Vue.js <https://vuejs.org/>, a progressive JavaScript framework that excels in creating complex, dynamic web applications. Vue's reactive data binding and component-based architecture facilitate efficient handling of real-time updates and interactive elements, which are critical for maintaining user engagement in a test management tool.

To enhance the user interface, Vuetify <https://vuetifyjs.com/> is integrated as a UI component library, providing Material Design-inspired <https://m3.material.io/>, ready-to-use components. This library streamlines the design process with its pre-styled elements, ensuring that the interface remains visually cohesive and responsive across various devices.

The back-end (API Layer) is designed as a REST API (more details in the following section), adhering to RESTful principles to ensure flexibility, scalability, and ease of integration.

The API bridges the front end and business layer, facilitating data flow and user requests through various endpoints. Key endpoints include:

1. Upload Test Cases: Allows users to upload test case scripts.
2. Parameter Configuration: Enables the setting and management of parameters for test execution.
3. Result Retrieval: Facilitates access to results generated by test executions.

As for the manipulated files, uploaded files are processed through the API, parsed, and then managed by the business layer components for dependency analysis, execution, and result display. Our data consists in

1. Python Test Scripts: The system processes test cases written in Python, enabling seamless integration and execution within the Atom Engine.
2. Domain Model Files (.dot): A .dot file (Graphviz) defines the domain model, which is essential for structuring dependencies and visualising test case relationships within the business logic.

More details about the tool's architecture is presented as follows.

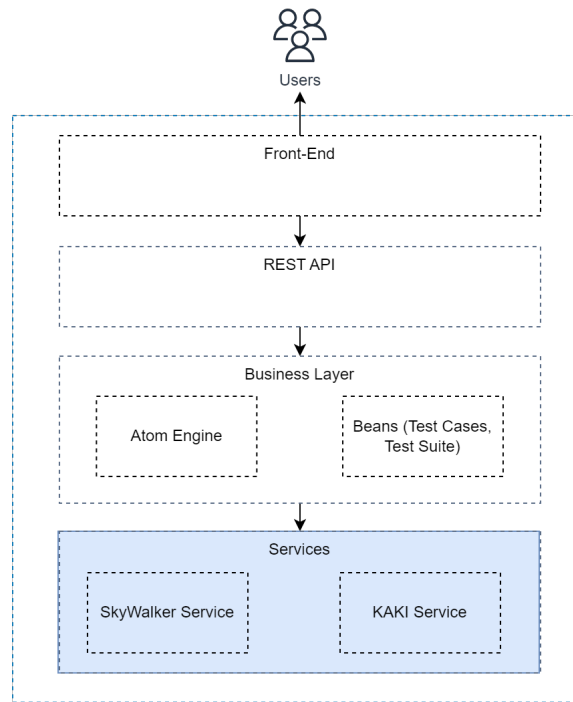
5.2 TOOL'S ARCHITECTURE

The tool architecture (Figure 23) follows a design pattern structure that divides the system into distinct layers, each responsible for a specific set of functionalities. These layers are structured hierarchically, where upper layers rely on services and data of lower layers. This design encourages modularity, separation of responsibilities, and scalability, making it a key option for developing durable and easily manageable software systems. The front-end layer connects with the REST API layer, which then communicates with the business layer. The business layer consists of different services that handle specific tasks in the process. Next, we outline the roles of each layer in our architectural design.

The front-end layer acts as the system's interface, tasked with creating a user-friendly platform for engaging with users. It incorporates visual elements and interactive functions to improve user experience. The REST API facilitates communication between the front-end and the business layer by offering endpoints for handling user requests and executing operations such as uploading test cases, setting parameters, and fetching results. It follows RESTful design principles to ensure scalability, flexibility, and interoperability.

The core layer of the system, housing the business rules, algorithms, and processing logic, is the Business Layer. This layer is responsible for generating the interleaving of test cases, analysing dependencies, and executing the generated test cases (Atom Engine). It comprises modular components, each handling specific tasks within the overall process.

Figure 23 – Tool's Architecture



Source: The author (2024)

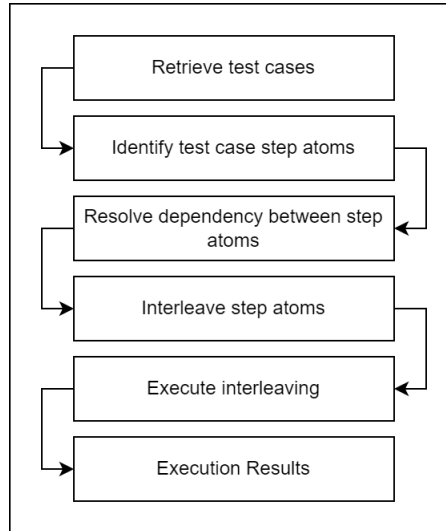
Within the Services, there are modular components responsible for specific functionalities, such as the dependency analysis module (KAKI Service) and the Skywalker service, where all the outcomes of the executions are displayed. These components consolidate related functionalities to enhance code reusability, maintainability, and scalability.

5.3 TOOLS USAGE WORKFLOW

Figure 24 summarises the workflow of our test case generation tool. It starts with the tool retrieving test cases and identifying the atoms related to them (*Input Test Case* tab on Figure 25). Next, the dependency between step atoms is resolved (*Dependency Analysis Checker* tab). Then, the tool proceeds to generate the interleaving of test cases based on specified criteria. The user is prompted with the generated test cases (*Generated Test Cases* tab). Finally, users execute the test cases and view the results through the *Execution Results* tab, gaining insights into the outcomes for test case analysis and validation. In each interface, the user can preserve their progress by selecting the *Save* button or discard it by choosing *Delete*.

The *Cancel* button redirects the user to the previous screen. A more in-depth explanation and corresponding screens for each component are provided in what follows.

Figure 24 – Tool's usage workflow



Source: The author (2024)

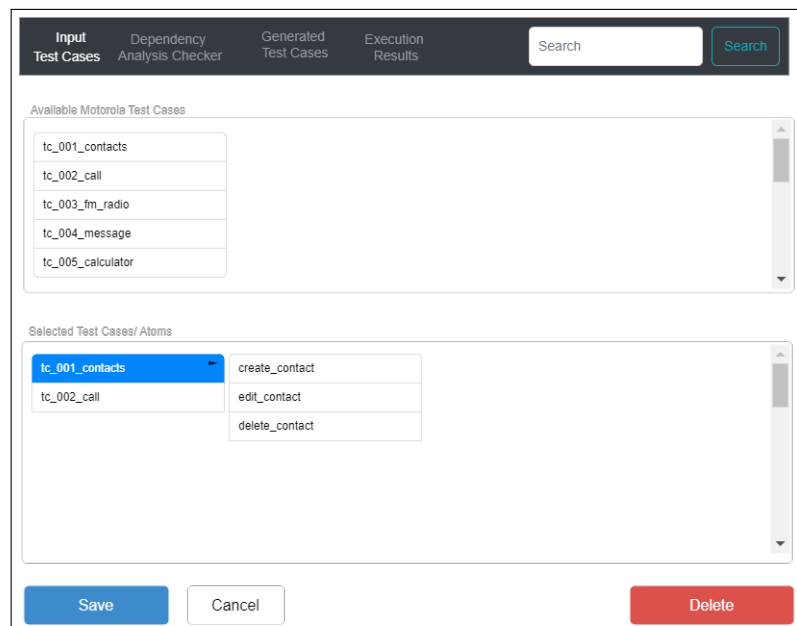
The workflow begins with the *Input Test Case* tab (Figure 25), where users interact with the graphical interface to visualise available test cases in the first frame (*Available Motorola Test Cases*) and then select the ones they wish to generate new paths in the *Select Test Cases/Atoms* frame. Those test cases are retrieved from Motorola's automation scripts databases, and for each one of them, the user can visualise all the corresponding atoms.

Next, in the *Dependency Analysis Checker* tab (Figure 26), the users must upload domain models containing predefined dependencies between test case steps (*Upload Domain Model* frame). In this step, the dependencies are analysed in real-time to avoid conflicts or inconsistencies that may impact the test case generation. This enhances the accuracy and reliability of the test cases. Moreover, in the second frame (*Generate Test Cases*), users can specify the desired number of tests within a specified range. With original test cases inputted and dependencies analysed, the tool is ready to generate new test cases from the interleaving of atoms (Generate Test Cases button).

The user can visualise the generated test cases in the following screen (Figure 27). They can then choose which test case to run by selecting them and clicking on the Execute button.

Finally, in the *Execution Results* tab (Figure 28), users can visualise the outcomes of the executed interleavings, including pass, fail, or inconclusive verdicts. This screen provides detailed insights into the execution status of each interleaved test case, aiding in test case analysis and

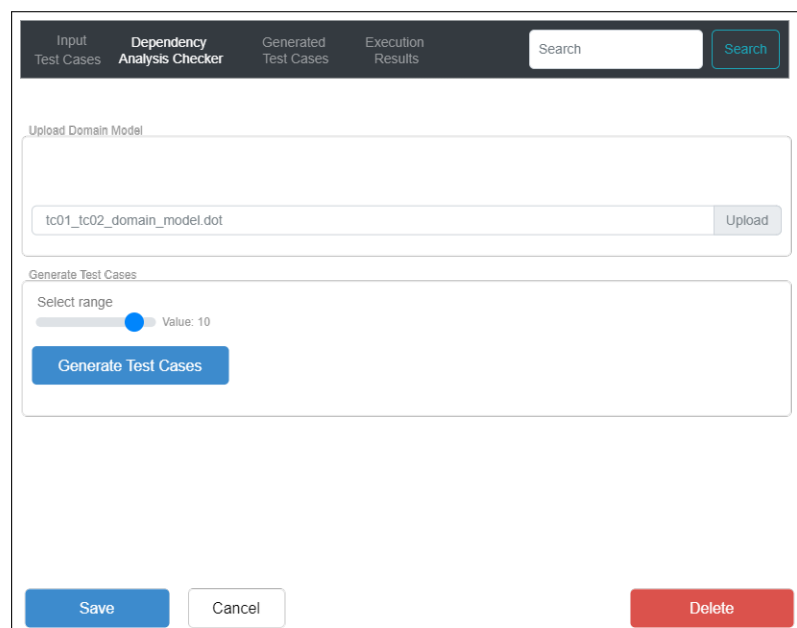
Figure 25 – Screen 1 - Test Cases



The screenshot shows the 'Input Test Cases' tab of a software interface. At the top, there is a navigation bar with tabs: 'Input Test Cases', 'Dependency Analysis Checker', 'Generated Test Cases', and 'Execution Results'. A search bar with a 'Search' button is located on the right. Below the navigation bar, the 'Available Motorola Test Cases' section contains a list of test cases: 'tc_001_contacts', 'tc_002_call', 'tc_003_fm_radio', 'tc_004_message', and 'tc_005_calculator'. The 'Selected Test Cases/Atoms' section shows a list of selected test cases: 'tc_001_contacts', 'tc_002_call', and 'tc_003_fm_radio'. To the right of this list, there is a table with three rows: 'create_contact', 'edit_contact', and 'delete_contact'. At the bottom, there are three buttons: 'Save', 'Cancel', and 'Delete'.

Source: The author (2024)

Figure 26 – Screen 2 - Dependency Analysis

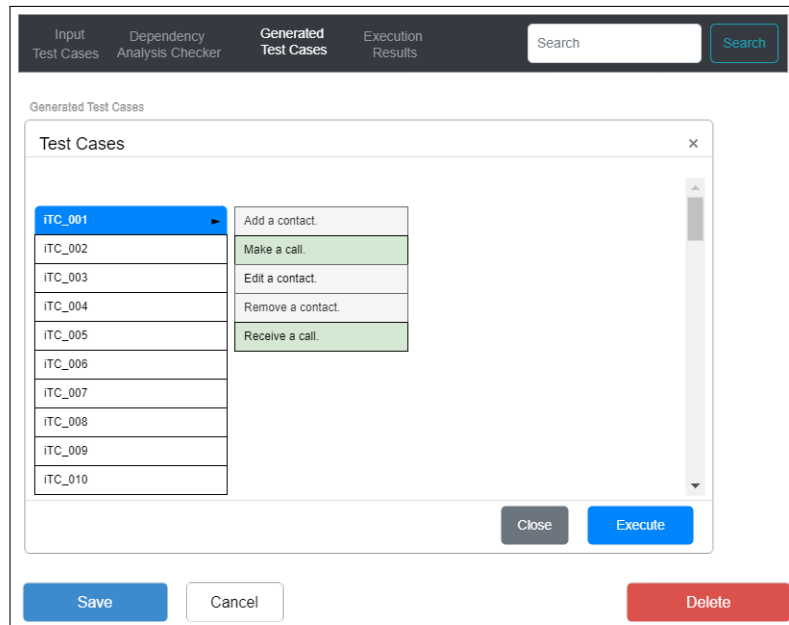


The screenshot shows the 'Dependency Analysis Checker' tab of the same software interface. The navigation bar at the top is the same. Below it, the 'Upload Domain Model' section has a text input field containing 'tc01_tc02_domain_model.dot' and an 'Upload' button. The 'Generate Test Cases' section features a 'Select range' slider with a blue dot and the text 'Value: 10'. Below the slider is a 'Generate Test Cases' button. At the bottom, there are three buttons: 'Save', 'Cancel', and 'Delete'.

Source: The author (2024)

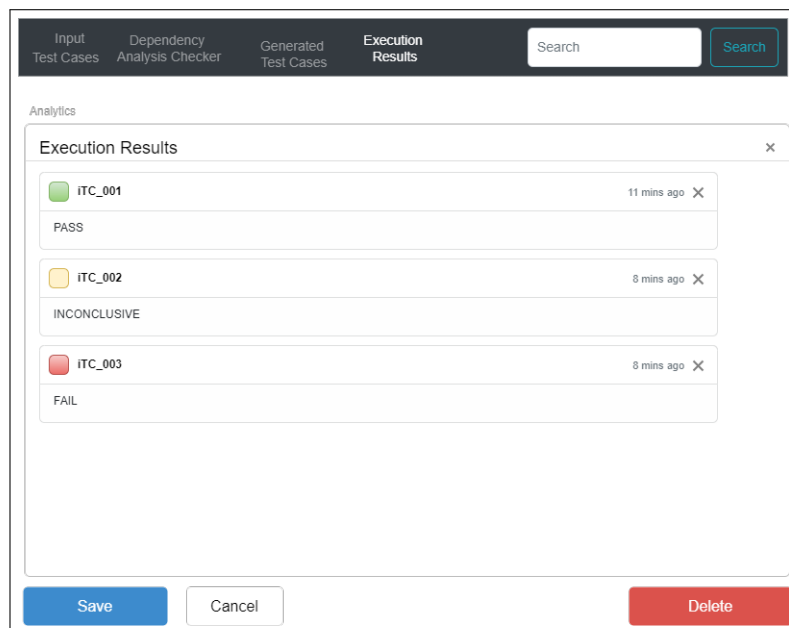
validation. Through interactive visualisations and intuitive interfaces, users can gain valuable insights into the effectiveness of their testing efforts and identify potential improvements.

Figure 27 – Screen 3 - Generated Test Cases



Source: The author (2024)

Figure 28 – Screen 4 - Execution Results



Source: The author (2024)

5.4 DEPENDENCY ANALYSIS: KAKI

Kaki de Arruda [2022] is a tool that verifies the consistency of a sequence of test steps by identifying the dependencies between them to generate valid execution sequences. Such a tool runs in background as part of the dependency analysis step of our tool workflow (see Figure

24). We explain how Kaki is used to support our tool.

In general terms, in Kaki, one must build a *Domain Model*, which characterises the application domain of a given test case by defining *Frames* and its associations. A valid and consistent *Frame* must contain an operation, patient, and extra information (slots), along with possible associations, which may be dependencies (for an action to perform correctly, some previous ones must occur) or cancellations and matching rules. A relevant facet of the tool comprises the consistency analysis of the test cases. The tool provides a GUI and an API for integration with other tools. For more details, refer to de Arruda [2022].

The integration process of the Kaki tool with the proposed approach happens via the Kaki API. We illustrate with a simple example how the dependency analysis is able to verify (and possibly update) a sequence of steps to ensure the dependency of every step is resolved.

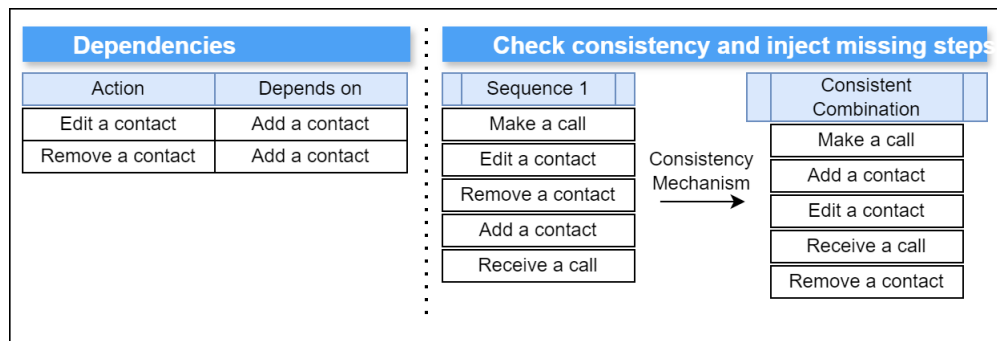
Consider the interaction between the *Managing contact* and *Performing calls* Test Cases (described in Tables 8 and 9, Chapter 4). We first provide the information needed to define a *Domain Model*. We focus on the associations between actions. Such associations may be expressed by dependencies or other relations we do not explore in this work; we focus on dependency analysis. In the illustrated example (Figure 29), we defined some dependencies according to the domain observation. For instance, to execute the action ‘Edit a contact’, the contact must be added first (‘Add a contact’). In what follows, we present the *Domain Model* relevant to the example provided, formatted as it would appear in the tool, as a dot file.

```
digraph DomainModel {

    edit_contact -> add_contact [class="depends"];
    remove_contact -> remove_contact [class="depends"];

}
```

Next, in the consistency analysis step, those actions are checked individually for possible direct dependencies or dependencies involving transitivity. Finally, the consistency mechanism suggests a valid execution sequence. Those final combinations are then used to build consistent test suites along with their automation and automatic execution.

Figure 29 – Dependency Analysis for *Performing calls* and *Managing contact* Test Cases.

Source: The author (2024)

5.5 LIMITATIONS

Potential limitations that could be relevant for the tool:

1. **Scalability Challenges with Large Test Suites:** As the complexity and volume of test cases grow, the tool might encounter performance bottlenecks, especially during dependency analysis and execution sequencing. Handling extensive interdependencies across thousands of test cases can require significant computational resources, which may slow down processing times or lead to delays in result generation.
2. **Accurate Dependency Mapping:** The effectiveness of the tool's dependency analysis relies on accurate and up-to-date dependency mappings within the domain model. If dependencies are incorrectly defined or omitted, the tool may fail to generate the correct sequence of test cases, potentially leading to inaccurate results or missed test coverage.
3. **Dependency on Python Test Scripts:** Since the tool processes test cases written in Python, it may not support other programming languages or frameworks directly. This limitation could restrict its applicability to projects or teams that rely on alternative languages for test scripting, potentially limiting its versatility in a multi-language environment.

6 EVALUATION

In this chapter, we discuss the outcomes of a hands-on case study that was carried out to assess the efficiency of the suggested approach in comparison to the conventional test case generation methods used by Motorola's Mobility Test Team. We focus on two assessment criteria, namely bug detection and coverage, to substantiate our conclusions. The subsequent sections elaborate on the planning, execution, and results of this evaluation.

6.1 CONTEXT AND MOTIVATION

The purpose of this research is to propose an approach for automatically generating sound tests using natural language models that can assess concurrent applications. One of the specific goals is to validate the proposed approach. To achieve this, a practical assessment is conducted to determine the effectiveness of the tests generated by the proposed approach in the context of mobile applications with concurrent behaviour. The effectiveness of the approach is evaluated by comparing the set of tests generated using the approach with the tests created by the Motorola's testing team. Both sets of tests were executed on the same software version of a mobile phone. To measure the effectiveness we collect the number of detected bugs and coverage of the application for each approach used in the evaluation.

6.2 PLANNING

6.2.1 Context Selection

This study is conducted within Motorola's development environment, specifically targeting the testing of mobile phone software that contains concurrent behaviour.

6.2.2 Participants

The participants involved in this case study are:

1. The test cases generated by the proposed approach.

2. The test cases generated by Motorola's engineers.

6.2.3 Variables

1. **Independent Variable:** The method used to generate the test cases (proposed approach versus generated by Motorola engineers).
2. **Dependent Variables:**
 - a) **Number of uncovered bugs:** The number of bugs identified by each approach.
 - b) **Test coverage:** The percentage of software components covered by each approach.

6.2.4 Research Questions

This research aims at answering the questions presented in Table 12.

Table 12 – Research Questions

Metric	Technique/Tool	ID	Description
#Uncovered Bugs	Automation Scripts	RQ1	Tests generated by the proposed approach effectively identify bugs that Motorola's original tests miss considering Motorola's proprietary automation scripts?
Coverage	Keyword	RQ2	Tests generated by the proposed approach cover different components compared to the original Motorola tests considering a key-word filtering technique?

6.2.5 Metrics

1. Number of uncovered bugs: Evaluates the effectiveness of the generated test suites using real-life scenarios within Motorola's development environment.
2. Coverage: The coverage is assessed using a filter mechanism based on keywords, enabling the retention of a more precise log data for coverage evaluation.

6.2.6 Design

The study follows a comparative case study design, where both testing approaches are executed under the same conditions. The test suites are applied to the same version of the mobile software (build) and hardware (model), and data regarding bugs and coverage will be collected.

1. Instrumentation

- a) Test Automation Tools: Motorola's proprietary automation tools were used to run both test suites and capture data on bug detection and coverage.
- b) Keyword Filtering Mechanism: A mechanism were used to filter logs based on keywords to ensure precise data collection related to test coverage.

1. Operation

a) Preparation

- i. Selection of Software (build): The mobile phone software used for testing are the same for both test suites, ensuring a consistent baseline.
- ii. Setup of Test Suites: Both test suites (the one generated by the proposed approach and the manually created) were prepared and executed under controlled conditions.

b) Execution

- i. The test cases generated by both approaches will be executed on the same version of the mobile software.
- ii. Data on uncovered bugs and coverage will be collected and logged for analysis.

c) Data Collection Procedure

- i. Bugs Identified: The number of bugs discovered by each test suite were recorded.
- ii. Test Coverage: The extent to which each test suite covers different software components were logged using LogCat tool, with data filtered using a keyword-based technique to ensure precision.

The subsequent sections offer a comprehensive examination of one approach of each category, highlighting the particular situations in which they were employed, how they were carried out, and the results achieved.

6.3 RESULTS AND DISCUSSIONS

6.3.1 Number of Uncovered bugs

For the bugs evaluation we take into account Motorola's actual development process and applying the optimised approach (Chapter 4) to several features. As a result of the pairwise combination of these features, more than 18 bugs were detected. The interaction between the following components led to the occurrence of the identified bugs: Camera, Home Screen, Themes, Contacts, Wallpaper, Messages, and Google Duo. The unexpected behaviors varied from changes in visual components, such as buttons and truncated text, to more critical errors, such as Application Not Responding. We present in Table 13 a compilation of all bugs along with their corresponding descriptions.

For this experiment, the Sanity and CoreRegression suites were taken into consideration. The subset of test cases considered for the evaluation underwent a thorough analysis to determine their suitability for execution. The criteria included identifying automatic test cases, those with concurrency characteristics, and those compatible with the hardware and software versions of the devices used in the experiments.

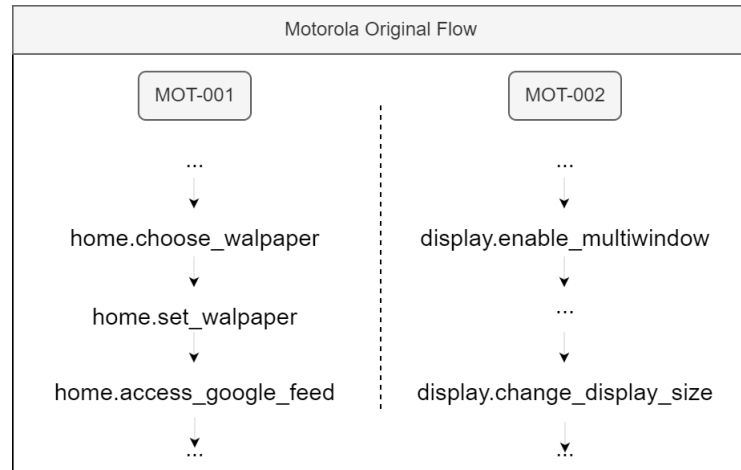
This investigation yielded 6 distinct inputs: 3 scripted test cases (enabling straightforward execution) and 3 exploratory test charters. The latter involves more complex investigations and relies on testers' expertise. The selection of charters aimed to infuse greater dynamism into the experiments, facilitating exploration across diverse scenarios and functionalities. Unlike scripted tests, exploratory testing allows testers to uncover defects and unforeseen behaviours through real-time interaction with the system.

For illustrative purposes, we provide an example of such an experiment by systematically delineating the steps that resulted in the issue documented in BUG-018 (refer to Table 13). However, it is worth noting that due to confidentiality agreements with our industrial partner, we cannot provide specific details about the hardware/software of the devices used in the experiment, neither the test case identifiers. Consequently, certain images may be blurry, and we have labelled the original Motorola test cases as MOT_00X and the generated interleaving as iTC_00X.

Figure 31 illustrates one of the interleaving (iTC_001) generated from the original test cases MOT_001 and MOT_002 (Figure 30), following the workflow described in Chapter 4. To keep it concise, some atoms derived from the test cases have been excluded, focusing solely

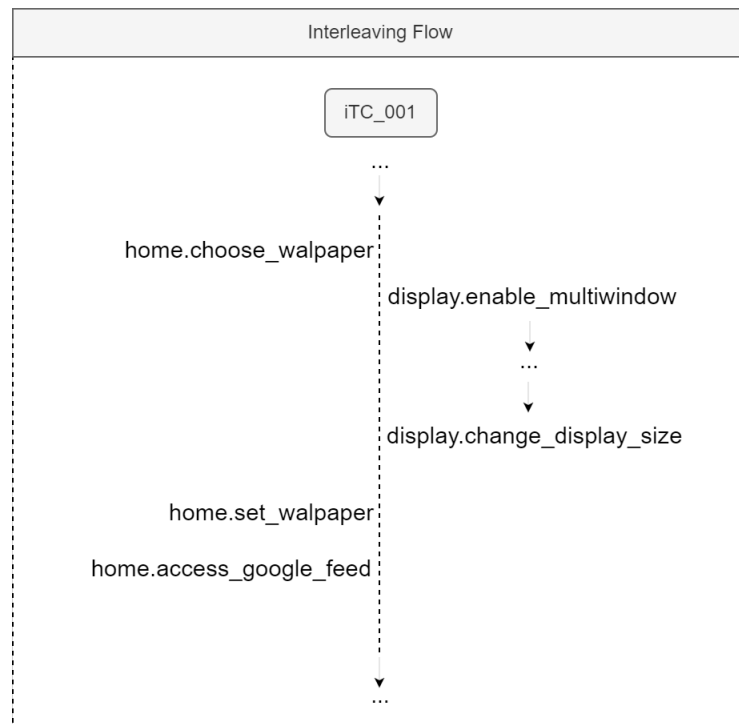
on the interactions that resulted in the BUG-018. We would like to emphasize that only this particular bug was validated by the industrial partner, while the others were classified as a true bug by the author.

Figure 30 – Fragments of the original Motorola test cases, MOT_001 and MOT_002.



Source: The author (2024)

Figure 31 – iTC_001 generated from MOT_001 and MOT_002.



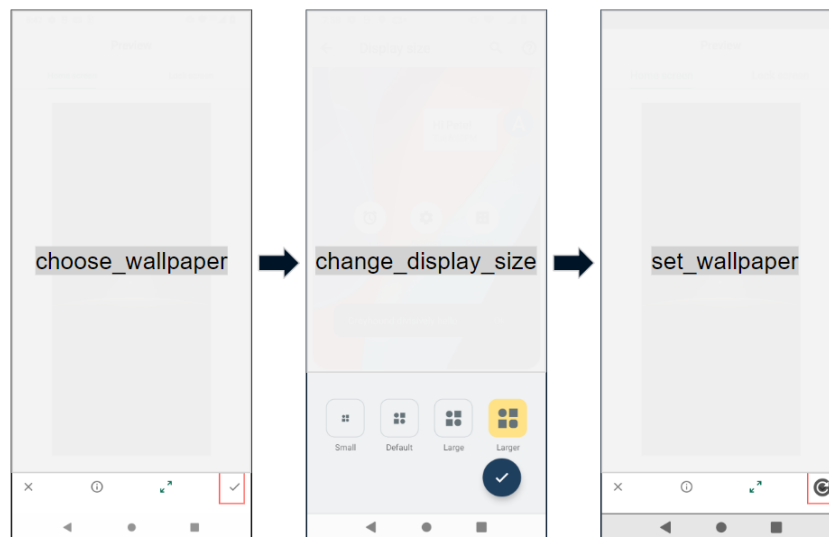
Source: The author (2024)

Originally, the test case MOT_001 tests interactions on the home screen, including tasks like adding, deleting, rearranging apps and widgets, and changing the wallpaper. On the other

hand, the test case MOT_002 focuses on testing interactions in multi-window mode, such as altering screen orientation and modifying display size.

When the test cases were run, in isolation, following their original step sequences, no issues were found. However, the generated interleaving, displayed in Figure 31, shows a sequence where the actions `choose_wallpaper` (MOT_001) and `change_display_size` (MOT_002) were interleaved. This change caused an issue on the wallpaper screen (highlighted in red in Figure 32). To change the wallpaper, the user must select and confirm his choice by pressing the tick button. However, increasing the font size caused the tick button to be covered by a reload button, making it impossible for the user to click on the button. This problem was also tested on a Google Pixel (pure Android-based) phone to confirm its accuracy.

Figure 32 – BUG-018.



Source: The author (2024)

6.3.2 Coverage Rate

Another factor to consider when evaluating the effectiveness of the test suites is the percentage of internal components covered or executed during the testing process.

Coverage is a widely used technique in software testing that checks how much of an application has been tested during the execution of a test case. A coverage criterion is a set of rules that measures the percentage of the internal structure of the implementation that has been covered by the test cases. In code coverage, common criteria include function coverage,

statement coverage, and branch coverage. The more coverage achieved, the more source code is executed.

We have initiated a study that focuses on selecting keywords for filtering the log. The use of keywords helps to refine the results of the analysis and provides an opportunity to investigate various scenarios. This is made possible by the significant reduction in the number of lines of code, and the fact that most of the resulting lines are directly linked to the steps executed in each test case.

Through this evaluation, we are also aiming to study some phenomena observed during the evaluation, such as subtle variations in the final results of the analysis depending on which settings the test case was executed. For instance, when performing a complete permutation with the steps in a given sequence, the coverage percentage of the same steps being performed individually may differ. Another factor that can cause a difference in the results is whether or not the phone is reset between executions. When the phone is not reset before starting an execution, it loads the log of previous executions, which can contaminate the execution log with irrelevant information.

6.3.2.1 Keyword Coverage Scope

To ensure an appropriate selection of test cases, we conducted an assessment of Motorola's testing areas (refer to Table 14 for a small sample). Motorola operates in several teams and testing areas aimed at different stages of the development process.

Table 14 – Sample of Motorola's Test Teams and areas

PLATFORM		MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
Automation	Auto Compliance	Modem Smoke (Manual and Automated)	Automation	Automation: Frevo	Automation
	Auto CoreApps/FWK				
	Auto CTS Verifier				
	Auto KPI				
	Auto Regression [GMS Regression]				
	Auto Regression [LATAM Regression]				
	Auto Sanity				
	Sanity Board				
	Smoke Board Tool				
	Smoke Tests				
	Stability				
Sanity		Modem Sanity	Google Acceptance	Localization	Sanity
Regression		Modem Regression	MODs/Alexa	Internationalization	
Compliance		Modem Hot Swap	Sanity		
3rd Party Apps		Modem Exploratory			
KPI					
WiSL					
SWAT					

Source: The author (2023)

In collaboration with the testing teams, we conducted an investigation to select the most appropriate test suites for implementing the proposed approach. Our focus was on selecting test suites that exhibited some degree of concurrency and were either already automated or had the potential to be automated. Nevertheless, it should be noted that several tests still require human intervention for their execution.

Table 15 displays a summary of the assessment, encompassing details regarding the number of tests in each designated test suite (of a particular testing campaign), the number of tests that present a potential concurrent behaviour, the number of chosen evaluation candidates and the number of test cases generated in accordance with TaRGeT's approach described in Chapter 2. The numbers in the #TCs column represent only a small fraction of the tests in Motorola's portfolio. It is important to mention that the TaRGeT tool generated 707 test cases by rearranging the steps of the 13 original Motorola test cases.

The automation status, as well as the execution and coverage status are more detailed in Tables 16, 17 and 18. In the respective tables, the colour green is used to represent automated test cases, a pass verdict, and the capture of logs. On the other hand, the colour orange indicates that the test case is not automated, resulting in a fail verdict, and no log capture. The cells with white background in the tables indicate that the test cases were not executed.

Table 16 – Selected test cases automation status

MOTOROLA				
PLATFORM	MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
MCA-1052	MTG-40199	MCA-542	I18N-210	PACE-737486
MCA-135	MTG-40198	MCA-703	I18N-223	PACE-271289
	MTG-40201			PACE-737490
TaRGeT				
PLATFORM	MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
TAR-PLA-001	TAR-MOD-001	TAR-PROD-001	TAR-GLOB-001	TAR-EXP-001
TAR-PLA-002	TAR-MOD-002	TAR-PROD-002	TAR-GLOB-002	TAR-EXP-002
TAR-PLA-003	TAR-MOD-003	TAR-PROD-003	TAR-GLOB-003	TAR-EXP-003
TAR-PLA-004	TAR-MOD-004	TAR-PROD-004	TAR-GLOB-004	TAR-EXP-004
TAR-PLA-005	TAR-MOD-005	TAR-PROD-005	TAR-GLOB-005	TAR-EXP-005
TAR-PLA-006	TAR-MOD-006		TAR-GLOB-006	TAR-EXP-006
TAR-PLA-007			TAR-GLOB-007	TAR-EXP-007
TAR-PLA-008			TAR-GLOB-008	TAR-EXP-008
TAR-PLA-009			TAR-GLOB-009	TAR-EXP-009
TAR-PLA-010			TAR-GLOB-010	TAR-EXP-010

Source: The author (2023)

Table 17 – Selected test cases execution status

MOTOROLA				
PLATFORM	MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
MCA-1052	MTG-40199	MCA-542	I18N-210	PACE-737486
MCA-135	MTG-40198	MCA-703	I18N-223	PACE-271289
	MTG-40201			PACE-737490
TaRGeT				
PLATFORM	MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
TAR-PLA-001	TAR-MOD-001	TAR-PROD-001	TAR-GLOB-001	TAR-EXP-001
TAR-PLA-002	TAR-MOD-002	TAR-PROD-002	TAR-GLOB-002	TAR-EXP-002
TAR-PLA-003	TAR-MOD-003	TAR-PROD-003	TAR-GLOB-003	TAR-EXP-003
TAR-PLA-004	TAR-MOD-004	TAR-PROD-004	TAR-GLOB-004	TAR-EXP-004
TAR-PLA-005	TAR-MOD-005	TAR-PROD-005	TAR-GLOB-005	TAR-EXP-005
TAR-PLA-006	TAR-MOD-006		TAR-GLOB-006	TAR-EXP-006
TAR-PLA-007			TAR-GLOB-007	TAR-EXP-007
TAR-PLA-008			TAR-GLOB-008	TAR-EXP-008
TAR-PLA-009			TAR-GLOB-009	TAR-EXP-009
TAR-PLA-010			TAR-GLOB-010	TAR-EXP-010

Source: The author (2023)

Table 18 – Selected test cases execution and log capture Status

MOTOROLA				
PLATFORM	MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
MCA-1052	MTG-40199	MCA-542	I18N-210	PACE-737486
MCA-135	MTG-40198	MCA-703	I18N-223	PACE-271289
	MTG-40201			PACE-737490
TaRGeT				
PLATFORM	MODEM	PRODUCT TEST	GLOBALIZATION	EXPERIENCES
TAR-PLA-001	TAR-MOD-001	TAR-PROD-001	TAR-GLOB-001	TAR-EXP-001
TAR-PLA-002	TAR-MOD-002	TAR-PROD-002	TAR-GLOB-002	TAR-EXP-002
TAR-PLA-003	TAR-MOD-003	TAR-PROD-003	TAR-GLOB-003	TAR-EXP-003
TAR-PLA-004	TAR-MOD-004	TAR-PROD-004	TAR-GLOB-004	TAR-EXP-004
TAR-PLA-005	TAR-MOD-005	TAR-PROD-005	TAR-GLOB-005	TAR-EXP-005
TAR-PLA-006	TAR-MOD-006		TAR-GLOB-006	TAR-EXP-006
TAR-PLA-007			TAR-GLOB-007	TAR-EXP-007
TAR-PLA-008			TAR-GLOB-008	TAR-EXP-008
TAR-PLA-009			TAR-GLOB-009	TAR-EXP-009
TAR-PLA-010			TAR-GLOB-010	TAR-EXP-010

Source: The author (2023)

In the following discussion, we outline the execution process of the selected test cases, describe the type of analysis performed, and present the results obtained from these tests. This provides a detailed account of the test case execution, the methodologies used for analysis, and the outcomes observed during the evaluation.

6.3.2.2 *Keyword Coverage Execution and Results*

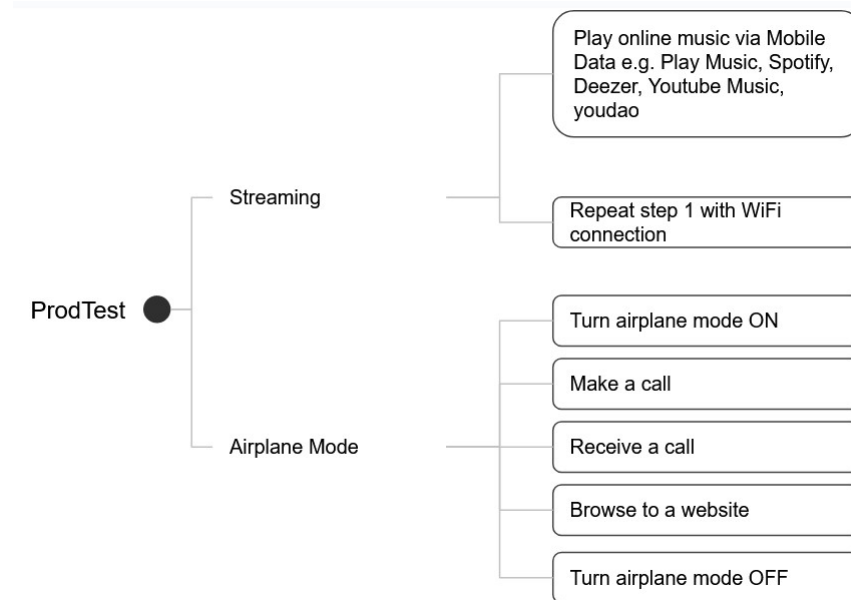
Once selected, the test cases were executed, taking into consideration various scenarios. Although some of the suites demonstrated better applicability for the execution of the approach in terms of exercising concurrent features and presenting the possibility of automation (when not already automated), all suites underwent extensive analysis. Among the analyses conducted, the following stand out:

- **Analysis 1:** Application of Filters. It verifies whether the proposed approach provides better coverage than the original Motorola test cases after applying a filter mechanism, which provides a more meaningful log, eliminating unnecessary information.
- **Analysis 2:** Tests the effects of applying a filter mechanism plus resetting the smartphone before each run. The study investigates the impact of clearing the log before execution, and analyses how it influences the outcomes.
- **Analysis 3:** Filtering and resetting the smartphone before the initial execution only. The subsequent executions are carried out without resetting the smartphone. The objective is to determine if there is any effect on the results when the smartphone is reset only for the first run.
- **Analysis 4:** Using a filter mechanism and multiple variations of execution order (performing the same permutation in different sequences). The impact of interchanging the order of steps on the execution results is analysed.

To keep it concise, we provide only the details of the planning, execution, and outcomes of some of the Product Test suite analyses. Two test cases from the Product Test suite were chosen as the basis for this evaluation. TC-MCA-542 (Table 19) focuses on verifying if the user is able to stream online music, while TC-MCA-703 (Table 20) tests the Airplane Mode feature in different settings.

The next step was to construct a hierarchical model (Figure 33) that allows the identification of potential interactions between the various steps.

Figure 33 – Selected Test Cases Hierarchical Model



Source: The author (2023)

Following the reverse engineering approach presented in Chapter 2, we proceed to derive use cases from the initial test cases. This involves analysing the test cases and extracting relevant information to create use cases that reflect the system's behaviour and functionality. Table 21 displays the use case UC_02 that was generated from the TC-MCA-703 test.

Figure 34 – Generated Permutation from the interleaving of Product Test use cases (UC_01: Streaming ||| UC_02: Airplane Mode)

Steps	Expected Results
1) Play online music via Mobile Data e.g. Play Music, Spotify, Deezer, Youtube Music, youdao.	Verify user is able to stream online music.
2) Turn airplane mode ON.	Airplane mode is turned ON - Data service icons are NOT present in the status bar.
3) Make a call.	The device can't establish a call or browse on the web.
4) Receive a call.	The device can't establish a call or browse on the web.
5) Browse to a website.	The device can't establish a call or browse on the web.
6) Turn airplane mode OFF.	Airplane mode is disabled and the device is registered again to the network - Data service icons are present.
7) Repeat step 1 with WiFi connection.	Verify user is able to stream online music.

Source: The author (2023)

After interleaving the test cases, they were automatically run using Motorola's automatic execution tool, and the execution log was recorded through LogCatwww.android.com [b]. To assess coverage, we employed a relative coverage metric. This metric is evaluated considering two sets, OTS_e and MTS_e , which stand for the lines covered by the optimised approach and the manual execution carried out by a Motorola team, respectively. Hence, the coverage of

OTS_e is calculated by the formula

$$\frac{\#OTS_e}{\#(OTS_e \cup MTS_e)} \times 100.$$

The coverage of MTS_e is calculated similarly. This calculation represents the relative coverage of a set of elements to the union of all elements observed. We adopted this coverage metric since we could not access the source code. The execution outcomes for all suites are presented later in this Chapter.

Key-word filtering to improve coverage quality

Filters based on keywords were employed to achieve a more meaningful. The systematisation of keywords was based on the study of the test cases, where the choice of keywords is guided by the significance of each particular word within the test context. The following example demonstrates how to identify keywords that can be used to filter logs for a test case. Given the steps in Table 20, choosing words that are relevant to the main goal of the test is crucial. On that test case, the objective is to assess whether the phone can establish a connection when the airplane mode is turned on or off. Thus, individual log lines containing the words call, dialer, ringing, incoming call, airplane mode, wifi and website are related to the scope and objective of this test and, therefore, good candidates to be used in the filtering process.

In what follows, the findings for analyses 3 and 4 applied to the Product Test Suite are presented. We demonstrate the analyses that include resetting the smartphone at the start of the initial execution in order to show the impact of this on the outcomes.

[RQ2] Tests generated by the proposed approach cover different components compared to the original Motorola tests considering a key-word filtering technique?

Considering the applied analyses we compared the execution results of two different variations of the same generated permutation, TAR-PROD-001 and TAR-PROD-002, with the execution results of the Motorola original test cases from which the permutations were generated, MCA-542 and MCA-703, resetting the smartphone only in the first run.

Filter: "call", "dialer", "ringing", "incoming call", "airplanemode", "wifi", "website"

Table 22 indicates there is a slight difference in the filtered values for each permutation variation. This occurs because it is not possible to control what is triggered during execution, even if the same set of keywords is used for both permutations. This slight variation exists in all runs since the smartphone never remains in exactly the same condition at all times. In addition, in this scenario, both permutations covered more lines than Motorola's original tests.

Based on this, we can infer that the use of filters enhanced the identification of the components covered during the execution. Moreover, variables like resetting the smartphone prior to execution or using different execution paths combined (more than one permutation) may cause disparities in the coverage outcomes.

Overall, considering all the scenarios in which this study was conducted, the tests generated using the proposed approach present a better coverage than the ones created by Motorola. In contrast, however, the suggested approach does not consistently offer superior coverage compared to Motorola in all analyses or test suites. In specific situations, there may be a marginal reduction in coverage when employing the proposed approach, especially when using only one permutation. This reinforces the idea that the greater the number of paths to explore (permutations), even if they originate from the same set of tests, the higher the potential of coverage to be achieved by the application. Despite the findings, a comprehensive examination of the log is necessary to fully grasp the observed phenomena.

Finally, the implementation of a filter mechanism has enabled us to navigate into a more sophisticated approach that precisely establishes a correlation between the executed lines of code and the actual testing steps. Moreover, it is possible to use filters combined with other techniques such as code instrumentation. Code instrumentation entails the insertion of a developer-designated tag into the automated testing code. When executed, this method allows for the filtering of log lines containing the inserted tag, resulting in a more precise and refined coverage. The scope of this project includes the execution of a case study involving code instrumentation. This is still under exploration.

6.3.3 Threats to Validity

We address the potential threats to the validity of our evaluation in what follows.

1. **Internal Validity:** The possibility of bias due to the test environment or tools. Consistent use of automation tools across both approaches helps mitigate this. However, a notable threat arises from the need to select only a subset of test cases for execution. The proposed approach generates a large volume of test cases, making it impractical to execute all. To address this, only the first 10 test cases generated were selected for evaluation. This selection criterion does not guarantee that the chosen subset is optimal, nor can it ensure that the subset is free from selection bias, potentially affecting the validity of results.
2. **External Validity:** The study is specific to Motorola's mobile phone software, so generalising the results to other types of software may be limited.
3. **Construct Validity:** The study depends on accurate measurement of bugs and coverage, with the risk that some bugs may go unnoticed by either approach.
4. **Conclusion Validity:** A statistical power to detect differences between the two approaches was not performed.

Table 13 – Compilation of uncovered bugs.

ID	Description
BUG-001	[Messages] The orientation of the image appears inverted.
BUG-002	[Home] The Home screen is displayed blank.
BUG-003	[Messages] Camera shut button cut.
BUG-004	[Contacts] When altering the theme settings twice (activating and deactivating) on the editing page, the contact name fails to display.
BUG-005	[Messages] App Crash.
BUG-006	[Split-screen] The phone's components will be displayed as truncated when the screen is partially opened in split-screen mode.
BUG-007	[Split-screen] When a call is in a split-screen view, rotating the phone to landscape orientation (180°) will result in the call buttons being partially obscured.
BUG-008	[Theme] During a split-screen view mode call, an error is displayed on the change theme screen when receiving a call.
BUG-009	[Theme] When altering the theme settings twice, toggling between enabling and disabling, on the editing interface, the navigation reverts back to the original screen and any modifications made to the photo are not saved.
BUG-010	[Message] When activating the Dark Theme feature following the capture of a photo within the Message App, the image fails to be included or attached.
BUG-011	[Message] Message Edit View goes back to previous page when changing the theme.
BUG-012	[Message] The video cannot be included in the Google message as a result of a screen malfunction that occurred following a theme change.
BUG-013	[Google-duo] In a multi-window setting, the captured image appears to be horizontally flipped when a photograph is taken.
BUG-014	[Google-duo] When transitioning between applications, the image displayed in Google Duo may appear reversed.
BUG-015	[Contact] When attempting to choose specific contacts for SIM 2, the "Manually assign contacts" interface experiences a crash following a modification in the phone's theme.
BUG-016	[Camera] ANR (application not responding) occurs following the adjustment of font size while configuring the rear camera photo resolution.
BUG-017	[Wallpaper] ANR occurrence following two instances of screen manipulation, involving alterations in font size and theme.
BUG-018	[Wallpaper] It is not feasible to verify the alteration of the wallpaper following a modification in the display dimensions.

Source: The author (2024)

Table 15 – Overview of the assessment applied to Motorola's Test Suites

TEST SUITE	#TCs	#Concurrent TCs	#Selected TCs	#Generated TCs
Platform	216	52	3	126
Modem	64	64	3	6
Product Test	53	5	2	5
Globalization [Internationalization]	160	2	2	70
Experiences	58	9	3	500
Total	551	132	13	707

Source: The author (2023)

Table 19 – TC-MCA-542: Streaming

ID	Test Step Description	Expected Results
MCA-542	Play online music via Mobile Data e.g. Play Music, Spotify, Deezer, Youtube Music, youdao	Verify user is able to stream online music
	Repeat step 1 with WiFi connection	

Source: The author (2023)

Table 20 – TC-MCA-703: Airplane Mode

ID	Test Step Description	Expected Results
MCA-703	Turn airplane mode ON	- Airplane mode is turned ON - Data service icons are NOT present in the status bar
	- Make a call - Receive a call - Browse to a website	The device can't establish a call or browse on the web
	Turn airplane mode OFF	- Airplane mode is disabled and the device is registered again to the network - Data service icons are present

Source: The author (2023)

Step ID	User Action	System State	System Response
1M	Turn airplane mode ON		- Airplane mode is turned ON - Data service icons are NOT present in the status bar %APM:= On%
2M	Make a call	%APM == On%	The device can't establish a call or browse on the web
3M	Receive a call	%APM == On%	The device can't establish a call or browse on the web
4M	Browse to a website	%APM == On%	The device can't establish a call or browse on the web
5M	Turn airplane mode OFF	%APM == On%	- Airplane mode is disabled and the device is registered again to the network - Data service icons are present

Table 21 – Use case UC_02:Airplane mode derived from TC-MCA-703.

Table 22 – Scenario 1: Comparison results of two different variations resetting the smartphone only in the first run

Parameters	TAR-PROD-001 (#)	TAR-PROD-002 (#)
Total universe lines	745	794
Motorola Total Execution lines	421, P: 56.51%	421, P: 53.02%
Proposed Approach Total Execution lines	580, P: 77.85%	608, P: 76.57%
Intersection Motorola—Proposed Approach	256, P: 34.36%	235, P: 29.60%
Difference Motorola—Proposed Approach	165, P: 22.15%	186, P: 23.43%
Difference Proposed Approach—Motorola	324, P: 43.49%	373, P: 46.98%

Source: The author (2023)

7 RELATED WORK

This chapter highlights studies that use models as input for test generation. It is divided into three sections, each focusing on a specific type of model.

The first section examines works that concentrate on generating test cases for concurrent systems. Such studies explore methods to effectively generate tests that account for the complexities and interactions of concurrent systems.

The second section showcases works that use models authored in natural language as input. These studies investigate techniques for automatically generating test cases based on requirements or specifications written in natural language.

The third section discusses works that employ diagrammatic and formal models as input for test generation. Such studies explore approaches to automatically generate test cases using models represented in diagrams or formal languages, which provide a more precise and structured representation of the system under test.

The final section explores the application of generative AI and large language models (LLMs) in the generation of test cases.

By categorising the works based on the type of input model, this chapter provides a comprehensive overview of research efforts in test generation from various modelling perspectives.

7.1 TEST OF CONCURRENT SYSTEMS

The research in Peres [2009] presents a black-box testing methodology aimed at identifying crashes in mobile applications through the execution of automated user scenarios. The proposed technique emulates authentic user interactions—such as tapping, swiping, and data entry—without necessitating an understanding of the application’s underlying code. By automating these interactions, the method effectively detects conditions under which the application may fail or crash, particularly in response to unexpected inputs or edge cases. This approach is specifically tailored for testing graphical user interface (GUI) components, where unpredictable user behaviors or environmental factors (such as low battery levels or poor connectivity) may induce instability. This work was inspired by the concept of atoms as testable units. In this regard, both studies effectively tailor test cases to detect specific types of software bugs: ours, targeting bugs arising from concurrent interactions, and the other, focusing

on crashes due to user interactions.

The test generation strategy presented in Andrade and Machado [2012] shares some similarities with our approach while also highlighting a few key differences. This strategy takes as inputs models expressed in natural language, specifically use case templates, which are designed to capture interruptions in mobile device applications. Interruptions refer to urgent events, such as an incoming call, that temporarily pause the execution of the current state of the application until the interruption event is resolved. Although interruptions can be seen as a specific type of concurrent behaviour, they differ from the general interleaving of flows in that the original execution flow is halted, the interruption is handled until completion, and then the original flow resumes. In contrast, our proposed approach does not explicitly model interruptions since modern mobile device applications are not typically interrupted by external events like calls. Furthermore, our work extends beyond interruption modelling. We consider two levels of concurrency: intra-feature concurrency, which focuses on concurrent behaviour within a single feature, and inter-feature concurrency, which addresses interactions and dependencies between multiple features.

Various methods have been employed to evaluate concurrent applications. The approach proposed in this thesis builds upon our previous work Almeida et al. [2018] for the test case generation of concurrent systems, where we presented initial ideas for extending the natural language model introduced in Nogueira et al. [2014] to incorporate the modelling of inter-feature and intrafeature concurrency. The extension proposed in Almeida et al. [2018] follows a conservative approach, ensuring that the original template elements, such as use case data, inclusion relations, and extension relations, are preserved. An improvement was later provided in Almeida [2019], which main contributions can be summarised as: CSP semantics to capture intra-feature concurrency; tool support has been developed to input use cases with concurrent features and to translate use cases into a CSP model with concurrency; a systematic process is proposed to reverse engineering existing test cases into use cases for further test case generation. This process is used in the empirical evaluation to obtain the use case models input by the test generation tool. More details of the cited contribution are presented in the Chapter 2 of this thesis.

The paper in Yu et al. [2020] introduces ConTesa, which is a novel tool for augmenting test suites in the context of concurrent software. Its primary objective is to generate new test cases that explore both code modifications and the resulting thread interleavings affected by those changes. ConTesa employs a dual-pronged approach that reuses existing test inputs while

expanding its coverage of possible interleavings through the use of random thread schedules. Additionally, it leverages an incremental symbolic execution technique to generate additional test inputs and interleavings, specifically targeting new concurrency-related behaviours in the program. This paper introduces a distinction in the interpretation of concurrency compared to our own approach. While we primarily concentrate on concurrency at the screen or GUI level, the paper in Yu et al. [2020] operates at a lower level, specifically focusing on concurrency. These divergent perspectives highlight the differing scopes and levels of concurrency that are addressed.

The paper in Murthy and Ulrich [2017] presents the design of a distributed test system tailored for testing distributed GUI applications. It introduces additional services such as coordination and synchronisation among the deployed GUI test frameworks, test verdict arbitration, and debugging support. Furthermore, the paper proposes a test specification method for test cases, which is based on an event flow graph model. This method enables the automatic execution of test cases using the suggested distributed test system. This approach allows the expansion of testing from single GUI application systems to distributed GUI application systems. It is important to note that the paper does not provide specific implementation results for the proposed system. Different from our approach, in that paper, graphical elements and test events (stimulus, response, stimulus/response, test system) need to be implemented as test scripts for execution on a test execution framework. A mapping of the local events contained in concurrent events from the test case specification to concrete test actions at the GUI of a SUT component or actions of the test system must be defined. In contrast to the approach described in Murthy and Ulrich [2017], ours focuses on handling concurrency by identifying actions within the textual scope of test cases that suggest concurrent behaviour and interaction between components.

In Offutt and Thummala [2019] it is introduced a novel approach based on Petri nets models that captures the behaviour of web applications. This model serves as the foundation for a technique proposed in the paper to design tests that specifically target concurrency in web applications. Additionally, the paper introduces new coverage criteria that are defined within the context of Petri nets. The approach takes into account the concurrency that arises from web-specific features such as HTTP sessions, behaviour of a browser, multiple instances of a browser, or shared data between web application requests. In contrast to the paper's focus on concurrency in web applications, our focus is on concurrency that arises from the parallel execution of features within the context of smartphones. While our work primarily operates

at the text level, focusing on natural language, the cited paper takes a different approach. Instead, such a paper uses a tool to extract the navigational structure of web applications.

In Sun [2008], test scenarios are generated based on a UML activity diagram and specified concurrency coverage criteria (weak, moderate, or strong concurrency coverage). This process produces a collection of test cases by transforming the activity diagram specifications into an intermediate representation using a set of predefined transformation rules. Subsequently, algorithms are employed to derive a set of test scenarios, which serve as the basis for generating test cases. Each test case represents a combination of choices with corresponding values that can lead to the execution of a specific test scenario. The extent of testing for concurrent elements is set by the chosen coverage criteria. To meet the criteria, tests are generated dynamically, allowing for control over the number of generated test cases. The primary focus of this approach is on generating functional test scenarios.

A software testing method is proposed in Cao and Wang [2018]. This method uses a Communicating Sequential Processes (CSP) model of the Implementation Under Test (IUT) to capture the concurrent behaviour of the system, as well as the expected properties of the System Under Test (SUT). The method begins by validating the model using model-checking techniques, ensuring its correctness and reliability. Subsequently, contracts are integrated into the IUT, and during the execution of the IUT, these contracts are continuously evaluated to verify if any violations occur. The presence of contract violations indicates that the IUT fails to adhere to the specified requirements. The primary objective of this method is to identify specific event sequences that represent incorrect behaviour within a real-world multi-thread testing environment. It is important to note that this approach differs from ours, as our focus lies in testing the graphical user interface (GUI) of mobile applications. In contrast, the work described in Cao and Wang [2018] involves code instrumentation and is primarily applied in the domain of safety-critical systems.

The work in Thummala and Offutt [2016] propose an alternative strategy for testing concurrent systems. This approach involves using a Petri net-based model specifically designed for web applications. The test generation process is guided by various coverage criteria, including Structural and Behavioural Analysis Coverage, Concurrent Behaviour Coverage Criteria, and RACC. These criteria leverage the structural and behavioural properties of the Petri net model to define the tests. They consider aspects such as model transitions, input data, HTTP sessions, and Petri net guard conditions to generate effective tests. It is worth noting that this approach differs from our own, as our focus is on the domain of mobile applications and the

testing of GUI components. In contrast, the work described in Thummala and Offutt [2016] is centered around web applications and employs a Petri net-based model. However, similar to our approach, that work also emphasises testing the Presentation Layer components, specifically targeting the Interaction Under Test (IUT) through the GUI. In terms of concurrency, their approach deals with the synchronous request-response cycle triggered by user interactions.

The approaches presented in these works primarily focus on specific types of concurrency (e.g., interruptions, thread-level interleavings, or Petri net models) or are applied to different domains (e.g., web applications or safety-critical systems). In contrast, our proposed approach provides a more comprehensive solution for testing concurrency in mobile applications, particularly those involving complex user interactions and feature dependencies. By using natural language descriptions to generate test cases, our approach simplifies the testing process while ensuring more comprehensive coverage of concurrent behaviours within the context of mobile GUIs.

7.2 TEST GENERATION FROM NATURAL LANGUAGE MODELS

Several approaches to generate test cases from natural language specifications have been proposed.

The authors in Carvalho et al. [2015], proposed a strategy to automatically generate test cases from Natural Language requirements (NAT2TEST - NATural language requirements to TEST cases), applied to data-flow reactive systems. Similar to our work, they use CNL (SysReqCNL) for authoring unambiguous requirements. Their generation approach comprises at least three fixed phases: (1) syntactic analysis, (2) semantic analysis, and (3) DFRS generation. DFRS is an internal formalism that can be translated into other formalisms that are the input for automatic test generation: Software Cost Reduction (SCR) Carvalho et al. [2013b], Internal Model Representation (IMR) Carvalho et al. [2013a] or CSP. Due to the focus on embedded systems, SysReqCNL is not suitable for specifying the flow of interactions of a user with the application GUI. On the other hand, the CNLs in our work focus on mobile applications described as interactions via a user interface. Similarly to our approach, this work represents data information as inputs/output values and can model interleaving between different system flows. Nevertheless, in their work concurrent behaviour arises only when the precondition guards of different system functionalities become enabled at the same time. Hence, concurrency is not expressed explicitly, as it is in our approach. They consider discrete

or continuous temporal properties, not yet explored in our work. Concerning the conformance notion adopted, they proposed a time-based input-output relationship (CSPTIO) that adapts the CSPIO conformance relation concept to consider time requirements to test reactive data flow systems.

A novel approach for automated detection and extraction of semi-structured requirements from requirement documents is introduced in Fischbach et al. [2019]. The input for the approach is requirements documents that contain semi-structured descriptions of business rules. Such descriptions are identified using machine learning algorithms that detect pseudo-code-like constructs within the natural language text. Once identified, the semi-structured descriptions are translated into Cause-Effect-Graphs (CEGs) using a rule-based approach. The CEGs are then used to generate test cases. The authors discovered that approximately 14% of the lines in requirement documents consist of "pseudo-code"-like descriptions of business rules, which offer a promising starting point for automating the creation of test models due to their structured nature. By employing their proposed solution, the authors achieved an 86% reduction in time required for test model creation, without compromising the quality of the models. Furthermore, the paper explores the challenges associated with manual creation and maintenance of test models, as well as the limitations of natural language when it comes to specifying requirements. Similar to our work, the authors of the paper input text requirements. The authors recognise the need to structure and organise the content of the document to make it amenable to automation. The limitations of the cited paper suggest it may not be suitable for all types of requirements documents and may require some manual intervention to achieve accurate results.

7.3 TEST GENERATION FROM DIAGRAMMATIC AND FORMAL NOTATIONS

The formal notion of conformance allows testers and developers to reason about the correctness of the generated test cases and the behaviours of the SUT. Existing theories in the field Tretmans [1999], Carvalho et al. [2013c], Nogueira et al. [2014], Cavalcanti et al. [2016] rely on a well-defined mathematical relation between the system specification and the *IUT*.

The paper in Tretmans [1999] proposes a formal approach for testing concurrent systems using labelled transition systems and **io**co (input/output conformance) as the conformance relation. The research presented in Nogueira et al. [2014] draws inspiration from the concept of **io**co and is the foundation for the formal definition of our proposed approach. It introduces

a testing theory (**cspio**) based on the CSP process algebra, which distinguishes input and output events. The authors also establish the equivalence between **cspio** and Tretmans' **ioco** relation. While the theory of **cspio** addresses systems that alternate inputs and outputs, it does not deal with systems that are able to input a sequence of input or that produce multiple outputs: these are possibilities in concurrent systems. Such a limitation is one of the main issues addressed in this thesis.

The article in Cavalcanti et al. [2016] presents a denotational semantics for CSP using suspension traces. This semantics specifically addresses the distinctions between inputs and outputs. The paper establishes healthy conditions for the suspension-traces model and proposes a characterisation of the conformance relation **ioco**. Additionally, the author in Cavalcanti et al. [2016] propose a strategy for automating the verification of conformance based on **ioco** and suspension-trace refinement using CSP tools. Furthermore, it opens up avenues for exploring algebraic laws and compositional reasoning techniques based on **ioco**. Although we share a common foundation in **ioco**'s conformance relation, there are distinctions in the specific semantics we adopt. While the paper relies on suspension traces, our approach adopts traces model annotated with special events to represent quiescence.

The paper in Carvalho et al. [2013c] proposes a timed input-output conformance relation (**CSPTIO**) that is formalised in CSP. This relation is designed to verify data-flow reactive systems. The authors provide a proof of the soundness of the proposed relation and validate its effectiveness by testing it on critical systems in the aeronautics and automotive domains. Both Carvalho et al. [2013c] and the approach proposed in the current thesis share a common adoption of the CSP formalism. The primary distinction lies in the domain of application. While Carvalho et al. [2013c] concentrates on critical automotive systems, our study is applied to mobile applications. Additionally, Carvalho et al. [2013c] introduces a timed input-output conformance relation (**CSPTIO**) that considers the temporal aspect of the systems under investigation. In contrast, our approach does not model any time-related constraints.

The paper Yimman et al. [2017] introduces a technique based on dynamic programming to produce concurrent test cases extracted from UML activity diagrams. Initially, the approach constructs a concurrent activity diagram based on a standard UML activity diagram. It subsequently converts the UML activity diagram into an activity graph and applies dynamic programming methods to find the total number of paths and total conditional paths in the concurrent test cases. Different from our approach, the input in Yimman et al. [2017] is an UML activity diagram.

In the study by Malekzadeh and Ainon [2010], the authors describe the development of an automatic specification-based test case generator (ATCG) specifically designed for testing safety-critical software systems. The tool takes as input a document written in natural language, which contains the specification of the critical system, including the causes and associated effects. The ATCG tool constructs a cause-effect table based on the input document, capturing the relationships between causes and effects. From this table, a Boolean expression is formed by assigning a variable to each cause and combining them with appropriate boolean operators. A visual representation in the form of a Cause-Effect Graph (CEG) is automatically generated based on the boolean expression. Finally, test cases are automatically generated based on the constructed boolean expression. However, a limitation of this approach is that it generates test cases associated with a unique effect at each execution of the tool, meaning that it may not capture all possible variations or combinations of effects. While the input document in Malekzadeh and Ainon [2010] is also based on natural language like our approach, there is a difference in structure and focus. In our work, the input follows a structured specification of a use case, emphasising the interactions within the system. On the other hand, the input document in Malekzadeh and Ainon [2010] follows a cause-effect pattern, highlighting the relationship between causes and effects in the system behaviour.

A Controlled Natural Language (CNL) is introduced in Schnelte [2009], specifically for the automotive domain. Requirements written in this CNL are translated into a formal model that captures rich temporal behaviour. The generated test cases from this formal model are designed to handle non-deterministic timing behaviour, which is particularly relevant in the context of automotive systems. In contrast, our approach primarily focuses on mobile applications rather than the automotive domain. While time constraints and timing behaviour are important factors in the automotive domain, our current work does not explicitly incorporate them. However, we acknowledge the significance of time constraints and plan to incorporate them in future iterations of our approach.

In the study by Sarmiento et al. [2014], a tool called C&L is introduced, which implements a test case generation approach based on natural language (NL) requirements specifications. The C&L tool automatically translates the NL requirements descriptions into UML activity diagrams, which are internally represented as directed graphs. Test cases are then generated using graph search strategies applied to these activity diagrams. The C&L tool provides a graphical display that facilitates the visualisation and management of test scenarios, test elements, and test cases. This graphical interface enhances the understanding and representation

of the generated test artifacts. Overall, the C&L tool bridges the gap between NL requirements and the generation of UML activity diagrams and subsequent test cases. It provides a visual representation to aid in understanding and working with the test artifacts derived from the NL requirements.

In the study Santiago Junior and Vijaykumar [2012], a model-based test case generation methodology is presented, which involves automatically translating natural language (NL) requirements into Statechart models. The methodology requires the definition of a dictionary by the test designer to specify the application domain, allowing for a flexible vocabulary that aligns with the specific domain terminology. While this approach shares a similarity with ours in terms of using natural language descriptions as input, there are some notable differences. In the work described in Santiago Junior and Vijaykumar [2012], there are no imposed constraints on how requirements should be written; the primary focus is on defining a domain dictionary to facilitate the translation process. On the other hand, our approach provides a controlled natural language (CNL) specifically designed for writing use cases, promoting a more structured and controlled manner of expressing requirements. Furthermore, the work in Santiago Junior and Vijaykumar [2012] emphasises the methodology for bridging the gap between informal and formal requirements, aiming to enhance the precision and formality of the requirements. In contrast, while our approach also involves capturing and formalising requirements through CNL, our main focus is on generating test cases that accurately reflect the behaviour and interactions of the system. Overall, while both approaches involve translating NL descriptions into formal models, our approach provides a controlled natural language for writing use cases and places emphasis on generating test cases, while the work in Santiago Junior and Vijaykumar [2012] focuses on the definition of a domain dictionary and bridging the gap between informal and formal requirements.

In Jiang and Ding [2011], the author focuses on automating the translation of textual use case descriptions into Extended Finite State Machine (EFSM) models, which serve as the input for generating test cases. The test case generation process uses a statement coverage criterion to ensure that each statement in the model is tested. Similar to our approach, test cases are generated from use case textual descriptions. However, there are slight differences in the structure of the use cases between their work and ours. In the format described in Jiang and Ding [2011], the use case structure consists of several fields, including Use Case, System under Discussion (which represents the system itself and other subsystems), Primary actor, Scope, Precondition, Main scenario (presented in a step-by-step structure), Variation, and Extension.

While the overall objective of generating test cases from use case descriptions aligns with our approach, the specific structure and formatting of the use cases differ between the two approaches. Our focus is on a more structured and controlled natural language representation of use cases, which allows for a more concise and consistent specification of requirements.

7.4 TEST GENERATION FROM GENERATIVE AI AND LARGE LANGUAGE MODELS

Recent research (Dakhel et al. [2024], Siddiq et al. [2024]) has emphasized the potential of Large Language Models (LLMs) to generate software test cases with greater efficiency and broader coverage, indicating promising applications for integrating LLMs into the test generation workflow.

The use of Generative AI and LLMs in test generation relies on their advanced natural language processing capabilities to transform software requirements or plain language descriptions into test cases. By interpreting input prompts, LLMs can autonomously produce test cases that reflect the expected functionality of the software, covering a wide range of scenarios, including edge cases and interactions. This methodology proves especially useful for complex, concurrent, or GUI-based systems, where traditional manual testing methods may fall short in accounting for all possible interactions.

The study by Dakhel et al. [2024] explores the use of LLMs in automated test generation, incorporating mutation testing to identify potential failures missed by the initial set of test cases. Their approach tailors LLMs to better understand and process natural language descriptions, enabling the generation of flexible and adaptable test cases, particularly for complex systems, including those with concurrent states. While both this work and the proposed approach emphasize comprehensive test generation and improved fault detection, the proposed approach extends the concept by incorporating dependency analysis. This ensures the correct ordering of test steps, which is crucial for systems with intricate dependencies. Furthermore, it introduces quiescence handling, an essential feature for managing scenarios where no further output is expected, such as in concurrent systems. In contrast, mutation testing, as outlined in Dakhel et al. [2024], primarily focuses on addressing coverage gaps identified in the initial test cases and does not specifically account for these nuances in test execution order or quiescence.

The paper in Siddiq et al. [2024] investigates the use of LLMs, such as GPT, for the automatic generation of JUnit tests in software projects. The empirical study evaluates the

effectiveness, quality, and utility of tests generated by LLMs in comparison to manually written tests by developers. The findings indicate that while LLMs can produce valid tests with good code coverage and some error detection capabilities, the quality of these tests can be inconsistent, with certain tests proving less effective. The study suggests that LLMs hold promise as a tool for automating test generation, enhancing productivity, and increasing test coverage. However, human oversight is still necessary to ensure the completeness and effectiveness of the tests, particularly in complex scenarios. In conclusion, while LLMs show potential as a supplement to manual testing, they cannot yet fully replace the expertise of human testers. In contrast, the proposed approach focuses on generating tests for concurrent systems from natural language requirements, specifically targeting mobile applications. Unlike the JUnit-based method, which primarily addresses unit tests and does not account for concurrency, the proposed method excels in generating tests for complex, concurrent applications. In summary, the JUnit-focused study is suitable for code-level testing of individual functions, while the proposed approach is tailored for higher-level system testing in concurrent environments, particularly with GUI applications, where dependencies, order, and quiescence play crucial roles.

In conclusion, generative AI, particularly LLMs, holds significant promise for automating test case generation, but it also faces several notable challenges. One key limitation is the occurrence of model inaccuracies or "hallucinations," where LLMs generate incorrect or irrelevant test cases. Additionally, LLMs depend heavily on clear and precise input requirements, making them sensitive to ambiguities in the provided specifications. The computational cost of running advanced models also poses a barrier, especially when dealing with large-scale systems or complex test scenarios.

LLMs further struggle with capturing complex behaviours, particularly in concurrent systems, due to their limited capabilities in managing multi-threaded interactions and handling "quiescence" — scenarios where no output is expected. This can make them less effective in testing systems with intricate, non-deterministic behaviours. Furthermore, the use of third-party LLMs raises concerns about security and privacy, as sensitive data processed through these models could potentially be exposed.

7.5 CONCLUDING REMARKS

This chapter provides an overview of various approaches related to the three main contributions of this thesis: testing of concurrent systems, test generation from natural language

models, and test generation from formal models. Several dimensions, including objectives, application domain, input/output format, conformance relation, and tool support, are considered in the analysis of these approaches.

By considering these dimensions, the chapter aims to provide a comprehensive understanding of the different methodologies and techniques employed in each area of contribution.

The literature review reveals that the proposed approach in this work possesses unique characteristics concerning the use of natural language. While existing test generation approaches primarily rely on natural language for specifying the input model, the proposed approach surpasses this limitation. It extends the usage of natural language to encompass standardisation and defining model constraints. This broader application of natural language enables a more comprehensive and expressive representation of the testing process.

Considering the specific circumstances surrounding the approach, particularly its focus on concurrency and its reactive characteristics, we incorporate the concept of quiescence. Quiescence entails a state in which a system is unable to generate an output or alter its state unless it receives an input beforehand.

Furthermore, the proposed approach stands out as the only known natural language-based approach for test generation that explicitly allows the specification of concurrent behaviour. This capability to explicitly represent and handle concurrency in the natural language specifications sets the proposed approach apart from other existing methods.

Our approach does not currently provide specific constructs or mechanisms to represent and handle time-related properties or behaviours. This limitation means that our approach may not fully capture and address temporal dependencies, constraints, or sequences that are relevant in certain testing scenarios. The inclusion of time aspects is therefore identified as an aspect that will be pursued in future work to further enhance the capabilities and effectiveness of our approach.

8 CONCLUSION

We have introduced an approach for generating sound test cases for concurrent features. The theoretical foundation to establish the soundness of this approach has been presented, along with the incorporation of a dependent analysis tool to tackle any inconsistencies related to the order of executing steps. This integration ensures the generation of coherent test cases with enhanced consistency.

To account for quiescent behaviour, we introduced a new relation called cspio_q , building upon the approach described in Nogueira [2012], for generating sound test cases that may involve quiescence.

Furthermore, we have optimised our test generation strategy to offer a more efficient alternative that closely aligns with the automation and execution environment of our industrial partner. The optimisation performs interleaving of atoms, simplifying the process by directly extracting pertinent information without the necessity for intricate reverse engineering test cases into use cases. We have also connected the optimised approach to the formal test generation theory, providing a formal proof and demonstrating that the optimised approach generates identical test suites for any possible input, mirroring the behaviour of the original approach. By establishing this close alignment, we assure that the optimised approach remains a dependable and effective means of generating sound test cases, providing an essential bridge between theory and a real-world application.

Tool support has been introduced for every phase of the proposed strategy, and the architecture and interface details have been presented. As for industrial practices, the proposed approach offers significant contributions by streamlining the test case generation process for concurrent systems. By automating the generation of test cases, the tool drastically reduces the time and effort required compared to traditional manual methods. It not only speeds up the process but also increases efficiency by systematically suggesting combinations of test scenarios that would be difficult or impossible for humans to conceive, given the complexity of concurrent interactions. This capability enhances the overall coverage and effectiveness of testing, allowing engineers to identify potential defects in scenarios that may have otherwise been overlooked, ultimately improving the reliability and robustness of software systems in production environments.

We conducted an empirical evaluation to gauge the effectiveness of the tests generated

using the proposed approach. This evaluation involved measuring the number of uncovered bugs and the coverage of the tests produced by Motorola. We also collected the same measures for the tests generated by the proposed approach, allowing us to compare the efficiency of the two testing approaches.

A significant number of bugs (more than 18) have been uncovered by applying the given strategy. This demonstrates the effectiveness of our approach in enhancing the bug detection capability of the testing process, thus contributing to the overall quality and reliability of the system.

Moreover, the test coverage measurement using filtering of the execution log could show the tests generated with the proposed strategy provide superior coverage compared to the tests generated by our industrial partner. For instance, in the evaluation of the specified scenario, the coverage attained by the proposed approach amounted to 580 lines, corresponding to 77.85% of the total, whereas the test suite developed by Motorola covered 421 lines, representing 56.51% of the overall lines. This comparison highlights a notable difference in the extent of coverage achieved by each approach, with the proposed method demonstrating a significantly higher level of test case coverage. This suggests that our approach not only uncovers new bugs but also provides more extensive testing, encompassing a wider range of functionalities within the system.

Finally, as part of advancements in test case generation for concurrent systems, this work led to the publication of the paper Sound Test Case Generation for Concurrent Mobile Features, presented at 26th Brazilian Symposium on Formal Methods. This research contributed significantly to the field of formal methods by proposing a reliable framework for generating test cases for concurrent mobile features. By applying formal techniques, the approach guarantees the accuracy and consistency of the generated test cases, effectively tackling challenges like non-deterministic behaviour and the complex interactions between multiple processes. Additionally, another paper titled Combining Sequential Feature Test Cases to Generate Sound Tests for Concurrent Features was submitted to the Science of Computer Programming journal and is currently under review.

8.1 STUDY LIMITATIONS

Dependence on Natural Language Requirements Quality

One of the key limitations of the proposed approach is its reliance on the quality and completeness of the natural language requirements. The method assumes that these requirements provide a clear and accurate description of the system's behaviour, which is often not the case in real-world scenarios. In many instances, requirements are vague, ambiguous, or incomplete, which can lead to test cases that miss critical scenarios or fail to fully capture the behaviour of the system. Furthermore, if the requirements do not explicitly define all possible concurrent interactions, the generated tests might overlook important edge cases, impacting the thoroughness of the testing process. In practice, addressing these gaps requires additional steps, such as refining the requirements or incorporating domain knowledge to supplement missing details, which can be time-consuming and resource-intensive.

Selection of optimal test cases

A significant limitation of approach lies in the challenge of optimal test case selection. Due to the sheer volume of test cases generated to encompass all possible feature interactions, it is often infeasible to execute every single test within reasonable time and resource constraints.

The selection process inherently carries a risk of overlooking certain interactions or potential bugs. By focusing on a smaller set of test cases, there is a chance that some scenarios—particularly those involving complex or less frequent interactions—may be omitted from the testing suite. These untested paths could contain critical issues that only become apparent under specific conditions, potentially leading to undetected faults that impact product quality and user experience.

In essence, while selective testing improves feasibility, it introduces a trade-off between execution efficiency and thoroughness. Without a sophisticated mechanism for prioritising cases based on impact and likelihood of failure, the tool may inadvertently exclude cases that provide valuable insights into the system's robustness. This limitation underscores the need for further refinement in test selection algorithms, perhaps incorporating predictive analytics or risk-based prioritisation, to better align selection with coverage requirements and enhance the tool's ability to detect latent defects. A more detailed discussion is presented in the 8.2 Section.

Computational Overhead

Another limitation of the approach is the computational overhead introduced by generating

large volumes of test cases. As the complexity of the system under test grows, the number of potential test cases increases exponentially, which can be both time-consuming and resource-intensive. While the approach includes an optimization strategy for generating new tests by permuting test steps (atoms), the sheer volume of cases that may be produced still requires significant computational resources. This challenge becomes particularly evident when testing large-scale applications or systems with many components, where the execution of all generated tests might be impractical. Addressing this issue may involve further optimization techniques, such as smarter test case selection algorithms or techniques for prioritizing tests based on risk or likelihood of revealing defects.

8.2 FUTURE WORK

Evaluation

Although we have addressed the evaluation of the proposed approach to a significant extent, we plan to employ a code instrumentation technique. This involves modifying the source code of the applications under test by adding tags to enable tracking and filtering the execution of code segments specifically triggered by the executing tests. Additionally, performance metrics such as execution time and resource usage will be analyzed to assess the efficiency of the generated tests in real-world scenarios. This will allow us to gauge the overhead introduced by our approach and ensure it remains feasible for industrial-scale applications. Further, we aim to validate the usability of the proposed tool through a series of case studies and user feedback from industry practitioners. This validation will provide insights into the tool's practicality, ease of integration, and potential areas for improvement, reinforcing the robustness and effectiveness of our approach in supporting concurrent system testing.

Proposed approach generation cost

Further investigation is needed to provide a better assessment of the cost of generating test cases using the proposed approach as well as a detailed mathematical analysis, aiming to reduce execution time and increase efficiency in the interleaving process of sequences. Possible solutions for optimising test case generation include implementing pruning techniques to discard interleavings that do not contribute to the desired coverage, thereby saving compu-

tational time. Another promising strategy could involve memoization of previously processed intermediate interleavings, avoiding redundant computation of repeated combinations.

Generative AI to support test case generation

Another subject for further investigation is exploring the integration of generative AI solutions to support test case generation for concurrent systems. As AI advances in natural language understanding and content generation, it presents promising avenues to enhance our proposed approach. Generative AI could assist by interpreting requirements written in natural language and identifying complex interactions that might otherwise go unnoticed. Additionally, AI could enable the development of adaptive testing strategies that automatically adjust to cover new configurations and states in evolving concurrent systems. By leveraging these capabilities, generative AI has the potential to reduce engineering time and effort needed to create effective tests while also improving coverage and the detection of bugs, thereby strengthening system reliability under concurrent conditions.

Test Case Selection

The exhaustive interleaving generation is computationally infeasible for large test suites. Using optimisations can reduce the effort of producing tests for concurrent features. However, the inherent number of combinations still leads to the problem of better selecting a subset of tests for running. Exploring selection mechanisms is a matter for future investigations. In that sense, investigating AI solutions could also play a crucial role in intelligently selecting the most relevant subset of cases, optimizing test coverage while reducing execution time.

Mechanisation of proof

Our proof of soundness was manually developed. An important future direction is to mechanise the proof using a tool like CSP-Prover Roggenbach [2008]. Finally, we plan to apply our approach to further industrial scenarios.

ACKNOWLEDGEMENTS

We thank Fundação de Amparo a Ciência e Tecnologia do Estado de Pernambuco (FACEPE). We also thank the partnership CIn/UFPE-Motorola Mobility (a Lenovo company) for partially supporting the author. Particularly, we thank all Motorola's Testing Teams for providing feedback about this work and the opportunity to carry out an empirical evaluation.

REFERENCES

- Rafaela Almeida. *Automatic Test Case Generation for Concurrent Features from Natural Language Specifications*. dissertation, Federal Rural University of Pernambuco, 2019.
- Rafaela Almeida, Sidney Nogueira, and Augusto Sampaio. Automatic test case generation for concurrent features from natural language descriptions. In *Brazilian Symposium on Formal Methods*, pages 163–179, Salvador, Brazil, 2018. Springer.
- Wilkerson L Andrade and Patrícia DL Machado. Testing interruptions in reactive systems. *Formal Aspects of Computing*, 24(3):331–353, 2012.
- Gregory R Andrews and Fred B Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys (CSUR)*, 15(1):3–43, 1983.
- Larry Apfelbaum and John Doyle. Model based testing. In *10th International Software Quality Week 1997. QW'97.*, pages 296–300, San Francisco, California, USA, 1997.
- Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
- Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (LNCS)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- Ilene Burnstein. *Practical Software Testing: a Process-oriented Approach*. Springer Science & Business Media, New York, NY, 2006.
- Yizhen Cao and Yongbin Wang. Concurrent software testing method based on csp and pat. In *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, pages 641–644. IEEE, 2018.
- Gustavo Carvalho, Flávia Barros, Florian Lapschies, Uwe Schulze, and Jan Peleska. Model-based testing from controlled natural language requirements. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 19–35. Springer, 2013a.
- Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Test case generation from natural language requirements

- based on scr specifications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1217–1222. ACM, 2013b.
- Gustavo Carvalho, Augusto Sampaio, and Alexandre Mota. A csp timed input-output relation and a strategy for mechanised conformance verification. In *International Conference on Formal Engineering Methods*, pages 148–164, Queenstown, New Zealand, 2013c. Springer.
- Gustavo Carvalho, Flávia Barros, Ana Carvalho, Ana Cavalcanti, Alexandre Mota, and Augusto Sampaio. Nat2test tool: From natural language requirements to test cases based on csp. In *18th Brazilian Symposium on Formal Methods*, pages 283–290. Springer International Publishing, 2015.
- Ana Cavalcanti, Robert M Hierons, Sidney Nogueira, and Augusto Sampaio. A suspension-trace semantics for csp. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 3–13. IEEE, 2016.
- E.M. Clarke, O. Grumberg, D. Peled, and D.A. Peled. *Model Checking*. The Cyber-Physical Systems Series. MIT Press, 1999. ISBN 9780262032704. URL <https://books.google.com.br/books?id=Nmc4wEaLXFEC>.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, third edition*. Computer science. MIT Press, 2009. ISBN 9780262033848. URL <https://books.google.com.br/books?id=i-bUBQAAQBAJ>.
- Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468, 2024. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2024.107468>. URL <https://www.sciencedirect.com/science/article/pii/S0950584924000739>.
- Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 285–294, Los Angeles, CA, USA, 1999. IEEE.
- Filipe Marques Chaves de Arruda. *A Formal Approach to Test Automation based on Requirements, Domain Model, and Test Cases written in Natural Language*. PhD thesis, Federal University of Pernambuco, 2022.

- Gustavo Henrique Porto de Carvalho. *NAT2TEST: Generating Test Cases from Natural Language Requirements based on CSP*. PhD thesis, Federal University of Pernambuco, Recife, PE, 2011.
- Felype Ferreira, Lais Neves, Michelle Silva, and Paulo Borba. Target: a model based product line testing tool. *Tools Session of CBSoft*, 2010a.
- Felype Ferreira, Laís Neves, Michelle Silva, and Paulo Borba. TaRGeT: a Model Based Product Line Testing Tool. In *Proceedings of CBSoft 2010 — Tools Panel*, pages –, Salvador, Brazil, 2010b. Springer.
- Jannik Fischbach, Maximilian Junker, Andreas Vogelsang, and Dietmar Freudenstein. Automated generation of test models from semi-structured requirements. *CoRR*, abs/1908.08810, 2019. URL <http://arxiv.org/abs/1908.08810>.
- Marie-Claude Gaudel. Formal methods and testing: Hypotheses, and correctness approximations. In *International Symposium on Formal Methods*, pages 2–8, Newcastle, UK, 2005. Springer.
- <https://m3.material.io/>. Material design. Available at: <https://m3.material.io/>.
- <https://vuejs.org/>. Vue.js - the progressive javascript framework. Available at: <https://vuejs.org/>.
- <https://vuetifyjs.com/>. Vuetify - a vue component framework. Available at: <https://vuetifyjs.com/en/>.
- Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer*, 7:297–315, 2005.
- Mingyue Jiang and Zuohua Ding. Automation of test case generation from textual use cases. In *Interaction Sciences (ICIS), 2011 4th International Conference on*, pages 102–107. IEEE, 2011.
- Mehdi Malekzadeh and Raja Noor Ainon. An automatic test case generator for testing safety-critical software systems. In *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*, volume 1, pages 163–167. IEEE, 2010.

- PVR Murthy and Andreas Ulrich. Distributed gui test automation. In *2017 14th IEEE India Council International Conference (INDICON)*, pages 1–6, 2017. doi: 10.1109/INDICON.2017.8487560.
- Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test Generation from State Based Use Case Models. Technical report, Cin-UFPE, <http://www.cin.ufpe.br/~scn/reports/jss10Extended.pdf>, 2010.
- Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects of Computing*, 26(3):441–490, 2012.
- Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects of Computing*, 26:441–490, 2014.
- Sidney Nogueira, Hugo L. S. Araujo, Renata B. S. Araujo, Juliano Iyoda, and Augusto Sampaio. *Automatic Generation of Test Cases and Test Purposes from Natural Language*, pages 145–161. Springer International Publishing, Belo Horizonte, Brazil, 2016.
- Sidney Nogueira, Hugo Araujo, Renata Araujo, Juliano Iyoda, and Augusto Sampaio. Test case generation, selection and coverage from natural language. *Science of Computer Programming (under revision)*, 20(1):77–143, 2019.
- Sidney Carvalho Nogueira. *Test Generation and Compositional Conformance Verification from Input-Output CSP Models*. PhD thesis, Federal University of Pernambuco, Recife, PE, 2012.
- Jeff Offutt and Sunitha Thummala. Testing concurrent user behavior of synchronous web applications with petri nets. *Softw. Syst. Model.*, 18(2):913–936, apr 2019. ISSN 1619-1366. doi: 10.1007/s10270-018-0655-8. URL <https://doi.org/10.1007/s10270-018-0655-8>.
- Glaucia Boudoux Peres. *A Black-box Testing Technique for the Detection of Crashes Based on Automated Test Scenarios*. dissertation, Federal University of Pernambuco, Recife, PE, 2009.
- Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012. doi: 10.1109/IWAST.2012.6228988.

- Markus Roggenbach. Csp-prover - a proof tool for the verification of scalable concurrent systems. 2008. URL <https://api.semanticscholar.org/CorpusID:2754514>.
- A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 1998.
- A. W. Roscoe. *Understanding Concurrent System*. Springer-Verlag London Limited, Inc., London, U.K., 2011.
- Valdivino Alexandre Santiago Junior and Nandamudi Lankalapalli Vijaykumar. Generating model-based test cases from natural language requirements for space application software. *Software Quality Journal*, 20(1):77–143, 2012.
- Edgar Sarmiento, Julio Cesar Sampaio do Prado Leite, and Eduardo Almentero. C&I: Generating model based test cases from natural language requirements descriptions. In *Requirements Engineering and Testing (RET), 2014 IEEE 1st International Workshop on*, pages 32–38. IEEE, 2014.
- Matthias Schnelte. Generating test cases for timed systems from controlled natural language specifications. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 348–353. IEEE, 2009.
- Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–322, 2024.
- Ian Sommerville. *Software Engineering*. Pearson Education, [S.I.], 2004.
- Ian Sommerville. *Software engineering/ian sommerville.*—, 2011.
- Chang-ai Sun. A transformation-based approach to generating scenario-oriented test cases from uml activity diagrams for concurrent applications. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 160–167. IEEE, 2008.
- T.Gibson-Robinson, P.Armstrong, A.Boulgakov, and A.W.Roscoe. Fdr3 17 a modern refinement checker for csp. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, Grenoble, France, 2014. Springer.

- Sunitha Thummala and Jeff Offutt. Using petri nets to test concurrent behavior of web applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*, pages 189–198. IEEE, 2016.
- Jan Tretmans. Testing concurrent systems: A formal approach. In *Baeten J.C.M., Mauw S. (eds) CONCUR'99 Concurrency Theory. CONCUR 1999*, pages 46–65, Berlin, Heidelberg, 1999. Springer.
- www.android.com. Android n. Available at: <https://www.android.com/versions/nougat-7-0/>, a.
- LogCat www.android.com. Logcat — www.android.com, b. URL <https://www.android.com>.
- Sumana Yimman, Suwatchai Kamonsantiroj, and Luepol Pipanmaekaporn. Concurrent test case generation from uml activity diagram based on dynamic programming. In *Proceedings of the 6th International Conference on Software and Computer Applications, IC-SCA '17*, page 33–38, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348577. doi: 10.1145/3056662.3056699. URL <https://doi.org/10.1145/3056662.3056699>.
- Tingting Yu, Zunchen Huang, and Chao Wang. Contesa: Directed test suite augmentation for concurrent software. *IEEE Transactions on Software Engineering*, 46(4):405–419, 2020. doi: 10.1109/TSE.2018.2861392.