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

FILIPE MARQUES CHAVES DE ARRUDA

A Formal Approach to Test Automation based on Requirements, Domain Model, and Test Cases written in Natural Language

Recife
2022

FILIPE MARQUES CHAVES DE ARRUDA

A Formal Approach to Test Automation based on Requirements, Domain Model, and Test Cases written in Natural Language

Tese apresentada ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciências da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação.

Orientador: Augusto Cezar Alves Sampaio
Coorientadora: Flávia de Almeida Barros

Recife
2022

Catalogação na fonte
Bibliotecária Luiza Maria Pereira de Oliveira, CRB4-1316

A773f Arruda, Filipe Marques Chaves de
A Formal approach to test automation based on requirements, domain model, and test cases written in natural language /Filipe Marques Chaves de Arruda. – 2022.
134 f.: il., fig., tab.

Orientador: Augusto Cezar Alves Sampaio.
Coorientadora: Flávia de Almeida Barros
Tese (Doutorado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, Recife, 2022.

Inclui referências.

1. Automação de testes. 2. Análise de consistência. 3. Linguagem natural controlada. 4. Csp. 5.Alloy. 6. Modelo de domínio I. Sampaio, Augusto Cezar Alves (orientador). II. Barros, Flávia de Almeida (coorientadora). III. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2022- 174

Filipe Marques Chaves de Arruda

"A Formal Approach to Test Automation based on Requirements, Domain Model, and Test Cases written in Natural Language"

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

Aprovado em: 26/09/2022.

_____

Orientador: Prof. Dr. Augusto Cezar Alves Sampaio

BANCA EXAMINADORA

_____

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

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE

_____

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

_____

Prof. Dr. Tiago Lima Masssoni
Departamento de Sistemas e Computação / UFCG

_____

Prof. Dr. Márcio Eduardo Delamaro
Instituto de Ciências Matemáticas e de Computação / USP

I dedicate this thesis to my mother, father, and wife.

ACKNOWLEDGEMENTS

Seria absurdo insinuar que a conclusão desse trabalho é um mérito exclusivamente meu. De forma geral, compartilho igualmente as honras conquistadas com todos que contribuíram direta ou indiretamente na minha vida e carreira. Foi a partir da colaboração de todos, conhecidos ou não, que tive o privilégio de dedicar-me à pesquisa. Porém, como sou conhecido por ser esquecido, adianto que a lista de agradecimentos a seguir não é, nem de perto, completa. Espero que as pessoas que não estejam aqui nomeadas sintam-se também reconhecidas – obrigado pelo apoio!

Começo agradecendo a Deus, a quem geralmente expresso minha gratidão discretamente mas com franqueza. Também aos meus pais, Isabel e Nilton, os quais dedicam suas vidas ao trabalho árduo que permitiu os privilégios que hoje tenho. Suas crenças na educação como força transformadora me trouxeram até aqui. À minha esposa, Raíne, companheira por mais de uma década, pelo apoio incondicional até nos momentos mais apreensivos. Ao meu orientador, Augusto Sampaio, o qual admiro tanto como cientista como ser humano, pela atenção, apoio, e ensinamentos que influenciaram minha trajetória. À minha coorientadora, Flavia Barros, pelos comentários e revisões valiosas. Ao professor Alexandre Mota pelo apoio na concretização dos projetos. Também ao professor Gustavo Carvalho pela ajuda com o processamento de linguagem natural e pelos feedbacks sobre o trabalho.

A todos que contribuíram com o desenvolvimento das ferramentas e/ou experimentos: Marlom Jobsom, Gabriel Schneider, Rafael Rofner, Fabiana Almeida, Viviana Toledo, Andre Bazante, Samir Ferreira e Audir Filho.

A Sergio Soares e Waldemar Neto pelo apoio com a formatação do plano dos experimentos, e aos membros da banca da qualificação – Leopoldo Teixeira, Tiago Massoni e Breno Miranda – pelos comentários pertinentes que contribuíram para a construção da versão final da tese.

Ao IFPE – em especial a Marco Eugênio, Italo Lemos, Rodrigo Folha, Marlus Barbosa, Cleber Silva e Renata Andrade – por permitir concluir a pesquisa.

Por fim, agradeço à Motorola (a Lenovo Company) pela longa parceria e suporte financeiro.

"...it is well known that a vital ingredient of success is not knowing that what you're attempting can't be done." (PRATCHETT, 2009, p. 119).

ABSTRACT

Software testing is a costly and time-consuming activity. For this reason, there is a substantial effort both in the academy and industry to automate it as much as possible. There are several test generation strategies and theories based on formal specifications. Formalisms such as process algebra, transition systems, and so forth, allow a precise definition of the semantics of system requirements, which one can leverage to verify essential properties, such as soundness, and derive test cases automatically. In an industrial context, however, ad-hoc/manual strategies are far more common due to their convenience since natural language descriptions are, more likely, easier to understand. Still, the lack of formal rigor can generate inaccurate tests, as there is no verification mechanism to ensure relevant properties. Also, requirements specified by product owners tend to be abstract by design, and should not be changed by other stakeholders, such as test engineers down the line. Then, it is a challenge to generate concrete test cases while still guaranteeing that the original behavior is preserved. Thus, in this work, we promote the use of natural language descriptions with rigorously defined underlying semantics (transparent to the user). Regarding the scope of this work, we cover the entire traditional (direct engineering) testing process and artifacts, from requirements to automation scripts generated automatically. Requirements written in a controlled natural language are parsed, and their semantics are automatically modeled using the CSP process algebra. To deal with different abstraction levels, from requirements to concrete tests, we formalize the concept of a domain model, in which additional information (such as dependencies, compositions, etc.), also written in natural language, can be combined with the requirements while preserving the original behavior. Then, by considering the domain model, sound and consistent test cases are generated from the discovered scenarios using the *cspio* conformance relation. These test cases can then be linearized back to natural language to allow manual execution or directly translated into test scripts for automated execution. On the other hand, we should not ignore a recurring scenario in which there is a large number of test cases already generated by hand, potentially created with ad-hoc techniques, without using requirements as input to the generation process. Hence, we also provide assistance to automate and make legacy test cases consistent. Custom tools for test generation, consistency analysis, and automation, were developed to mechanize the entire process, and evaluated in the real-world setting of a partnership with Motorola, a Lenovo Company. As a result, we not only generated and automated the same test cases that were described by hand in retrospect from old requirements (with more than 90% precision and 80% text size reduction) but also discovered new scenarios and created, from new features, test cases approved to be used in production. The efficiency of the consistency analysis, carried out through the Alloy Analyzer, was also evaluated for real test cases. Although most analyses took less than 10 (ten) seconds, we developed an alternative implementation that exhibited a

massive decrease in the analysis time.

Keywords: test automation; consistency analysis; controlled natural language; csp; alloy; domain model.

RESUMO

Teste de software é uma atividade que demanda tempo e custos significativos. Por este motivo, existe um considerável esforço tanto na academia como na indústria para automatizar o máximo possível desse processo. Existem estratégias e teorias de geração automática de testes baseadas em especificações formais. Formalismos como álgebras de processo, máquinas de estado, entre outros, possibilitam uma definição precisa da semântica dos requisitos, permitindo que propriedades importantes, como *soundness* (consistência), sejam verificadas mecanicamente e que casos de testes sejam derivados. No contexto industrial, entretanto, estratégias de geração manual ou *ad-hoc* são comuns por serem mais acessíveis, já que artefatos descritos em linguagem natural são, potencialmente, mais facilmente entendidos por *stakeholders*. Mas a falta de rigor formal pode gerar testes imprecisos, pois não há mecanismos de verificação de propriedades como, por exemplo, *soundness*. Além disso, requisitos especificados por *product owners* tendem a ser inerentemente abstratos e não devem ser modificados por outros *stakeholders*, como engenheiros de testes, em etapas posteriores. Então, torna-se um desafio gerar casos de teste concretos e garantir ao mesmo tempo que o comportamento original da especificação seja preservado. Dessa forma, este trabalho promove o uso de linguagem natural na descrição dos artefatos, mas com uma semântica subjacente, transparente ao usuário, rigorosamente definida. Todo o processo de engenharia direta de casos de teste é explorado, dos requisitos até *scripts* de automação gerados automaticamente. Requisitos são analisados sintaticamente, e uma semântica é automaticamente gerada na álgebra de processos CSP. Para lidar com as diferentes granularidades de abstração, formalizamos o conceito de modelo de domínio, no qual informações adicionais (como dependências, composições, etc.), também descritas em linguagem natural, possam ser combinadas com os requisitos originais. Então, considerando o modelo de domínio, são gerados casos de teste consistentes e consolidados a partir dos cenários encontrados utilizando a relação de conformidade *cspio*. Esses casos de teste podem ser, então, linearizados em linguagem natural, permitindo execução manual, ou traduzidos diretamente para *scripts*, possibilitando uma execução automática. Por outro lado, na prática, há uma grande quantidade de casos de teste já gerados manualmente, possivelmente de uma forma *ad-hoc*, sem utilizar requisitos no processo de geração. Por isso, a estratégia também prevê suporte para automatizar e consolidar casos de teste legados. Ferramentas para geração de testes, análise de consistência e automação foram desenvolvidas para mecanizar todo o processo, sendo avaliadas no cenário real de uma parceria com a *Motorola, a Lenovo Company*. Como resultado, não somente foram gerados e automatizados os mesmos casos de teste, em retrospectiva, que foram criados manualmente a partir de requisitos pré-existentes, mas também novos cenários foram descobertos. Além disso, novos casos de teste, para novas features, foram gerados e formalmente aprovados pela empresa para serem usados em produção. A eficiência da análise de consistência, implementada utilizando o Alloy Analyzer, também é avaliada

para casos de testes reais. Apesar das análises, majoritariamente, demorarem menos de 10 (dez) segundos, nós desenvolvemos uma implementação alternativa que diminuiu massivamente o tempo de análise.

Palavras-chave: automação de testes; análise de consistência; linguagem natural controlada; csp; alloy; modelo de domínio.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BNF	Backus-Naur Form
C&R	Capture-and-Replay
CNL	Controlled Natural Language
CSP	Communicating sequential processes
cspio	CSP Input-Output Conformance
CSPm	Machine readable syntax for CSP
EBNF	Extended Backus-Naur Form
FSM	Finite-State Machine
GF	Grammatical Framework
GUI	Graphical User Interface
IOLTS	Input-Output LTS
IUT	Implementation Under Test
LTS	Labelled Transition System
MBT	Model-Based Testing
NLP	Natural Language Processing
SUT	System Under Test
TC	Test Case

CONTENTS

1 INTRODUCTION

Test artifacts may include features, plan, use cases, test cases, traceability information, and reports. Typically, to automate Test Case (TC) execution, test scripts are used. They are important resources to verify whether an implementation conforms to the system requirements. According to the literature and practitioners, the adoption of test automation tools or techniques reduces the cost of test cycles (RAFI et al., 2012). However, despite this global reduction, the initial cost for test design is higher with the use of automation practices and the maintenance can be problematic. Therefore, test scripts should be designed in a way to encourage reuse, being easier to maintain and to remain functional despite the constant changes on the System Under Test (SUT). This way, automation turns out to be a productive solution (GRECHANIK; XIE; FU, 2009).

Considering the particular context of mobile devices development, which is the application domain we focus on in this work, test script generation tends to be more complex than in the case of traditional systems. Note that, in this context, scripts are executed on less stable platforms/environments that suffer from other variable conditions, such as dynamic localization, sensors, distinct network providers, and heavy hardware dependency (CHANDRA et al., 2014). This increasing complexity, in addition to the lack of standardization and mechanized generation/analysis, raises issues on various artifacts created by different stakeholders in the testing process. We catalog next the main issues we identified as a result of a broad investigation on traditional software testing automation.

Abstraction gap and traceability – The traditional testing process produces and consumes different artifacts, such as Requirements, Use Cases, Test Cases, Test Scripts, etc. These artifacts have multiple abstraction levels and are, in general, handled by people with different roles with different technical backgrounds. Keeping trace of these artifacts, and how they relate when a change is made, is a hard task, both for time constraints and for the lack of cascading updates in the artifacts. Test automation strategies, in particular, are generally focused in low-level system instructions, which creates a gap for flexible/high-level strategies (ALEGROTH; NASS; OLSSON, 2013).

Additionally, most projects require that details are added in later stages. Therefore, a viable strategy should allow partially executable models. Besides, keeping implementation details from the abstract model helps to break down complexity.

Heterogeneous notations – The multiple abstraction levels usually have different notations that require different expertise in order to add information or make changes. For instance, use cases may be described in UML while test cases are described in natural language using a tabular format. Likewise, test objectives can be defined in a different formal notation from the one used to model the specification (UTTING; PRETSCHNER; LEGEARD,

2012). The lack of a unified (homogeneous) language contributes to the problem of consistently maintaining and updating the artifacts. There must be a compromise between flexibility/expressiveness and maintainability/readability, all considering the consistency of artifacts described in (possibly different) notations.

Inconsistency – Abstract requirements may lead to inconsistent tests. These underspecified tests, with respect to missing dependencies or the lack of input/output mapping to the implementation, hinder an effective automated execution, causing unexpected errors or invalid results. For instance, to automate the action of sending an email, one should make sure that the SUT is currently connected to the internet. If this required state is not verified, the execution might produce inconsistent results (depending on whether the internet connection was activated by unrelated actions beforehand). Additionally, inconsistent results may arise due to duplicate mappings to scripts for the same actions.

Execution time – A trivial strategy to guarantee that test scripts start their execution on a valid system state is to make them self-contained. In other words, each script executes all required setup methods (always assuming that the SUT is in the initial state). However, in doing so, the execution time of a test plan might be infeasible since the system must be reset for each test case. Lengthy execution times can defeat the purpose of regression suites since running them takes so much time that it is not feasible to execute them during regular development (TILLEY; PARVEEN, 2012). The correct system state should be ensured by using more efficient strategies, like, for instance, the execution of a test suite in such an order that a test cases benefits from the setup established by previous ones in the sequence.

Internationalization – Considering globalization, nowadays most companies have sites in different countries that do not share the same mother tongue. Specifications can already be ambiguous, especially when there must be off-site translations, contributing to misinterpretations. Mahmood and Ajila (2013) even find results showing that using native language to specify the system improves the correctness of the modeling.

Technology lock-in – Frameworks, platforms, and programming languages are in constant change, as well as business requirements. Tying a testing process to a specific technology might make the entire process obsolete before the effort to build it pays out. For instance, technology lock-in poses a critical threat to testing solutions in the cloud since a provider can suddenly increase prices or show technical issues (OLIVEIRA; MARTINS; SIMAO, 2017). A sustainable strategy should be portable and consider only open, accessible, and stable languages/frameworks.

Legibility – This concern can fall into the “Heterogeneous notations” issue, since novice testers might not be familiar with unusual complex notations. Therefore, an accessible language must be adopted to reduce ambiguity and allow non-experts to be productive.

However, even though a poor readability of TCs make the task of evolving or maintaining them harder, the readability is often neglected (GRANO et al., 2018).

Duplicate artifacts – One of the side-effects of not having efficient traceability is to end up finding duplicate artifacts across the project. Multiple strategies were proposed to mitigate or minimize this issue, but a formal strategy should envision this problem in a constructive way, aiming to avoid these duplications.

Regardless of the test automation strategy adopted, these are some of the issues that can hinder maintenance and productivity when dealing with test cases generated by hand from experience and intuition. The lack of a standardized automation process, in which every aspect of testing could be machine-readable, contributes to the emergence of these problems. To validate whether these maintenance and productivity problems can also be observed within an industrial context, we present and discuss next a participant observation.

1.1 IMMERSION IN SOFTWARE TESTING INDUSTRY

For several years we were involved in several testing activities performed in partnership with *Motorola, a Lenovo Company*. We not only had access to the artifacts, both used or created, but also participated *in loco* in multiple teams responsible for distinct steps of the testing process. This immersion helped us to understand the improvement opportunities in each step and how they are interconnected. It also served as an important input to formulate our research question, as well as for the design of the tools and experiments presented later. Next we briefly discuss our observations related to: test automation; manual execution; test generation; feature analysis; and, finally, feature definition.

1.1.1 Automation Team: Participant Observation

One of the main priorities in the testing process is to obtain fast software quality reports. Therefore, when a TC is often executed and is viable for mechanization, it becomes a prioritized candidate to be automated. However, there are always development and maintenance constraints. At first, teams focused solely on automating new TCs as much as possible. Developers were hired and automation frameworks were built to make the automation process faster, but it resulted in growing script errors due to different environment conditions and rapidly changing requirements. In this light, the teams adapted design patterns such as Page Objects (LEOTTA et al., 2013b) and dependency resolution to organize its codebase, resulting in a significantly better success rate. However, every change would now require meticulous reviews since there is no automatic traceability to the original requirement. Therefore, even a single change can cause errors in multiple TCs across different teams.

The main goals of this observation are to: 1) Be acquainted with the everyday tasks of an automation team; 2) Gather and document pain points; 3) Read documents and training resources; 4) Execute daily tasks; and 5) Get insights on where and how to apply potential solutions.

The chosen team was responsible for automating the sanity suite, which is a small set of tests frequently executed to report critical errors as quickly as possible. The team members were working at the site localized at CIn-UFPE, Recife, Brazil.

Observations and Discussions

1. There is a considerable learning curve for newcomer developers: they face a large amount of concepts and frameworks to become familiar with, even before automating a simple scenario. A lot of time was spent reading manuals for the multiple libraries.

2. The most time-consuming activity is to analyze suite executions: the auto pass rate is always less than the final pass rate (after revisions).

3. There is no mapping between errors and fixes on the codebase: this lack of traceability hinders automatic solutions for maintenance.

4. No timesharing data about the time spent automating new TCs *versus* maintaining old ones.

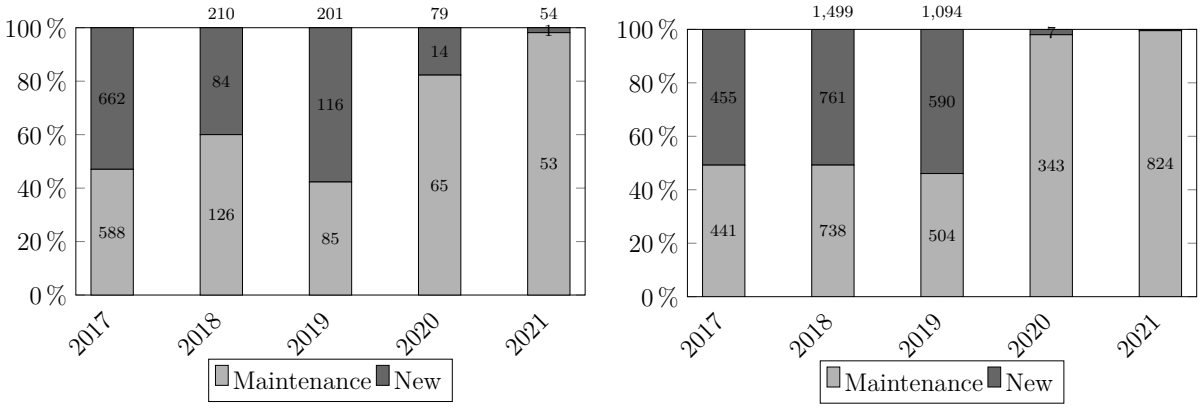
5. Team members are overloaded with maintaining the current test scripts. Thus, they have difficulties meeting the demand for creating new test scripts.

In an attempt to quantitatively validate the team members' narrative, we have collected data from internal projects about 1) the tasks created and resolved over the years; 2) the auto and final pass rates from official test execution results. Regarding the first topic, we have extracted all tasks associated with an issue in the management tool. The chosen projects were suggested by the team members because they are related to test automation. Figure 1 shows the distribution of the issue types over the years. The inferred issue categories (maintenance and new) are derived from the original issue types. Each project has different custom types, but even the same issue type can mean a different category in a different context. To infer the categories, we recurred to the team experience. For instance, issues marked as "Story" were put in the "New" category, while "Defect" issues were considered as "Maintenance".

In Figure 1 we can notice that the number of issues related to automating new TCs has been shrinking over the years in comparison to the total of maintenance-related issues, corroborating the team report.

Regarding the information about pass rates, we have extracted the results from an internal and custom tool that stores the results from all test executions that use the

Figure 1 – Distribution of inferred issue categories over the years



(a) Team A (b) Team B

Source: The author (2022)

internal test framework. From all these results, we have filtered only the ones marked as official and belonging to the test suite developed by the team we have observed. After processing the data, we discovered that, on average, the auto pass rate stands at 76% while the final pass rate at 82%. Thus, maintenance still is a staling issue, as reported by multiple developers. Some teams consider no longer automating new TCs since they can only afford to maintain the already automated ones. Although we analyzed a relatively small population (only one team) and might have an imprecise inference of the issues categories, the evidence from the team feedback and the issue management system ratify each other.

1.1.2 Further Direct Observations

Manual Execution The testing process still heavily relies on manual execution, since a large number of TCs are not feasible for automated testing. This decision considers technical and cost constraints but also a business choice. In some scenarios, an actual human interaction with the SUT gives more confidence to the test result. For instance, when one needs to test a GPS, the test has more credibility when actually driving around than manipulating the device sensors to simulate a trajectory. Because such tests need to be executed by human testers, TCs describe all required procedures and information in plain natural language. Additionally, there is no standard for the different teams across the globe to write them besides the basic structure (summary, setup, steps and expected results). This scenario leads to: 1) Different granularity: some tests describe step by step each screen interaction to get the test done, while others only have one step with the general idea for what should be tested; 2) Ambiguity: this is usually expected in descriptions written in plain natural language, but it is augmented both by the granularity issue and by the ambiguity inherited from the requirements. It also leads to wrong test results and slow execution by newcomer testers.

Test Generation TCs are usually generated from requirements or bug reports. Requirements could be extracted from internal documents or external sources. Assuming that all required information is gathered and understood, there are different ad-hoc strategies to create new TCs, but all of them are solely based on the test engineer's experience. The general idea is to find the relevant scenarios that should be tested and describe them in a spreadsheet, allowing the test steps to be further detailed. This spreadsheet is then uploaded to the actual tool in which the TCs are stored. This approach might lead to missing relevant test scenarios but, most importantly, it hinders test case maintenance. Since there is no traceability between requirements and TCs, it is hard to know which TCs are affected by a future update and it is even harder to know how to fix them. Because of that, all TCs must be thoroughly reviewed and updated accordingly.

Feature Analysis Feature analysis encompasses the process of analyzing a set of requirements regarding specific SUT and software versions. When an analysis is requested from a team, it must consider whether the test scenarios must be covered by the team and, in that case, if new tests must be created. Based on internal reports, information can spread across different documents and often requires tacit knowledge to understand it. It was not uncommon for engineers to search for the meaning of some requirements both in internal and external search engines.

Feature definition We also participated in the process of defining new functionalities. Because of its inherent creative nature, it is out of the scope of our work. During the discussions, different artifacts were created such as slides, prototypes, etc. Once the requirements are decided, however, they can be rewritten in compliance with any proposed standard to be used as input to the testing process.

1.2 A FORMAL YET ACCESSIBLE APPROACH

The apparently unrelated issues listed at the beginning of this chapter were observed in the immersion and can all be traced back to the lack of a mechanized analysis using precise notations and formal strategies to test case generation and automation. The seminal work of Gaudel (1995) argues that testing can be formal, too. From this perspective, there are several test generation strategies and theories based on formal specifications, such as *ioco* (TRETMANS, 1996) and *conf* (BRINKSMA, 1988). Formalisms such as process algebra, transition systems, etc., can precisely define the semantics of system requirements, which allows one to verify essential properties and derive test cases automatically. Closely related work, such as TaRGeT (FERREIRA et al., 2010) and NAT2TEST (CARVALHO et al., 2014), leverages formal languages and verification tools to mechanize the analysis. However, even when we have a rigorous formal approach, some maintenance issues still arise such as, for instance, execution inconsistencies and over-detailed models that need to be updated in

every development iteration. As an illustration, consider some examples and how they are tackled both in practical and formal approaches.

- When a screen element changes in a GUI-based application, the related test artifacts should be changed too. In most ad-hoc strategies, developers can adopt design patterns, such as Page Objects, to improve traceability, but the update is still mostly manual (which can be imprecise). When using formal models, the change is automatic. However, understanding the model and finding the correct component to update is complex for non-expert users. Besides, there are additional difficulties when different models are used for development and testing.
- It is common to have different notations for each step of a test automation process. Ad-hoc and formal strategies can use quite different languages, diagrams and logic. Thus, it is hard to make changes that deal with varied abstraction levels and notations.
- Most strategies are dependent on specialized tools or standalone solutions. This can be onerous when the underlying technology must be replaced.

In this light, this work pursues a strategy for the industrial context, particularly focused on the mobile device application domain, in a partership with Motorola, a Lenovo Company, that promotes the use of accessible notations, such as natural language descriptions, without giving up a formal theory. The entire traditional (direct engineering) testing process is covered, from requirements to automated scripts. To deal with different abstraction levels, the concept of a domain model arises, in which additional information (such as dependencies, cancellations, compositions, and instantiations) can be combined with the requirements to generate concrete scripts. On the other hand, the large number of test cases already generated by hand, possibly without using requirements as reference, should not be ignored. Hence, aside from supporting the test generation and automation from new requirements, we should also assist in the automation and consistency analysis of legacy test cases. Ultimately, by following this formal approach adapted to an industrial context, we provide evidence from evaluations and experiments that the above-mentioned problems can be minimized.

1.3 RESEARCH QUESTION AND OVERALL STRATEGY

As discussed in the previous sections, many issues in a testing process can arise from the lack of standardization, which hinders the automation of the entire process. Even though many strategies in the literature formalize the test generation process, they use notations that require specialized training or do not cover script generation for automated execution. Considering this context, the leading question that guides our research is the following:

How to generate sound, consistent and executable test scripts based on requirements written in natural language?

To answer this question we first discuss some related challenges such as 1) How to process natural language descriptions; 2) How to verify the soundness of derived test cases, and 3) What practical strategy can be used to generate tests that can be interpreted directly by a test driver without demanding over-detailed requirements. The proposed solutions for these challenges are listed next:

1. The input to the overall process is restricted to sentences that comply with a Controlled Natural Language (CNL), with a well-defined syntax and precise semantics interpretation; this allows development of deterministic parsing as well as a formal (semantic) model generation.

2. The Implementation Under Test (IUT) can be modelled by a Communicating sequential processes (CSP) process to use a conformance relation, *CSP Input-Output Conformance (cspio)* (NOGUEIRA; SAMPAIO; MOTA, 2014), for soundness verification.

3. The domain model, which must also comply with a Controlled Natural Language (CNL), can progressively describe test actions and dependencies that can be interpreted by a test driver, while not changing the original expected behavior.

Considering our industrial context, however, we cannot ignore the large number of test cases already generated by hand, potentially created with ad-hoc techniques. The dichotomy between the literature's best practices and the industry reality is also what drives this work. Therefore, we also try to answer the follow-up research question:

How to generate consistent and executable test scripts based on legacy test cases written in natural language?

An expected challenge is that legacy test cases might have no associated requirements and even have imprecise setups. Thus, soundness cannot be verified, but we can still address consistency via a domain model. Because these legacy TCs are not standardized, we proposed the following solution:

4. Freestyle natural language descriptions from legacy test cases can be matched by similarity with CNL descriptions.

Each of the previous challenges, in addition to their corresponding solutions and how they relate to each other, is discussed in Section 1.3.1. We briefly disclose our overall approach for test automation by the standardization of natural language descriptions with mechanized generation and analysis of the underlying semantics. First, we introduce

the major scenarios in which we apply our strategy (Section 1.3.1). Then, we present the main used artifacts (Section 1.3.2). After that, we show an overview of the test case generation process (Section 1.3.3). Lastly, we present how test automation can be carried out after a mechanized, sound, and consistent test generation (Section 1.3.4).

1.3.1 Scenarios

Particularly, we concentrate on UI/functional test cases, since the practical context of this work is mobile device testing with no access to the source code (black-box testing), involving a partnership with Motorola Mobility, a Lenovo company. To align with the industrial demand, we have considered two major scenarios.

The first scenario is a traditional direct test case engineering process, in which we parse the CNL requirements specifications and generate intermediate artifacts until we have the corresponding test scripts. This scenario assumes that all requirements are documented, accessible, and in compliance with our CNL. The other scenario deals with a more challeging context of legacy test cases manually generated in an *ad-hoc* way, without following a standard or without coming from explicitly defined requirements. In this case, because there might be no requirements from which to infer the expected behavior, it is not possible to reason about relevant properties such as soundness. However, since both scenarios can be observed in an industrial context, we address both of them: the first with a more rigorously defined strategy than the second. Figure 2 illustrates these scenarios.

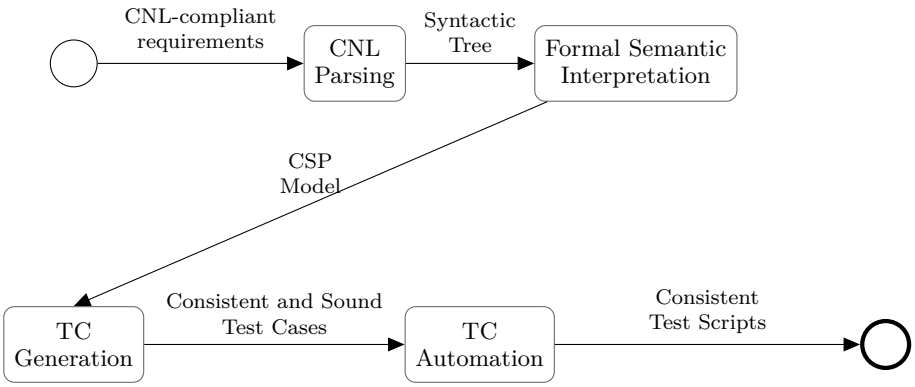
As seen in Figure 2a, we deal with the direct engineering scenario of automating test cases generated automatically from requirements written in compliance with a proposed CNL. The requirements in natural language are parsed to obtain the syntactic tree, which is used to define the underlying semantics in CSP process algebra. Then, the CSP model is used to generate sound and consistent test cases in compliance with the proposed CNL which can be, in turn, mapped to automation scripts automatically. Each task is discussed in the corresponding dedicated chapter, as detailed in Section 1.5.

The second scenario is illustrated in Figure 2b, in which test cases were already created in the past by hand (hence "legacy"). In this scenario, the test step descriptions are either matched using text similarity to CNL-sentences or they are defined on-the-fly, via a capture procedure. Then, an equivalent consistency analysis to that of direct engineering is carried out. Because there are no associated requirements, soundness is not addressed and only consistency is verified.

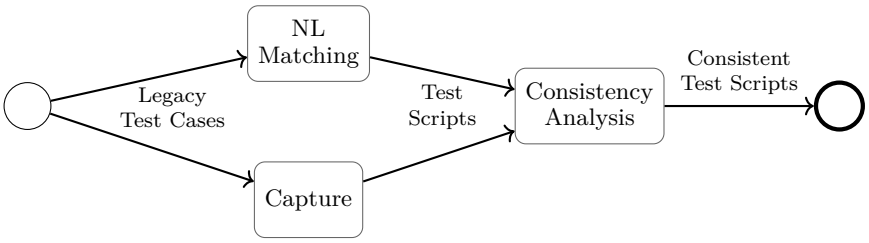
1.3.2 Artifacts

In a traditional software testing process, there are many artifacts involved. In most cases, software behavior is defined by a set of functionalities, each one having its own description. In our context, this description is part of a document named Feature. Each feature has a list of relevant requirements, which can be clearly described or scattered over many

Figure 2 – Process tasks



(a) Automation Process



(b) Automation of Legacy Test Cases

Source: The author (2022)

attachments with no clear distinction. From these descriptions, using their own interpretation, analysts prepare artifacts to test the SUT behavior. While some test analysts create use cases to ease the process of finding relevant scenarios, others may write test cases directly from the feature descriptions with no intermediate artifact.

Features

A feature describes, in our context, a system functionality. It may comprise multiple related requirements distributed over a dozen attached documents (such as slides, manuals and prototypes). Features are the origin for the development and testing processes because they hold all information about the software expected behavior. Because they are the main source of information, there are also additional annotations that help to categorize them. For instance, one may indicate the affected products, supported versions, product owner and other tags. Each feature can contain a limited number of requirements which, in composition, describes the SUT behavior. These requirements can be scattered over different related files, such as presentations, manuals, spreadsheets etc.

Requirements

The description of a feature may not be written in a standardized template. It is typically described in freestyle natural language with the help of figures, slides, and mockups. To keep the process simple, we manually obtain a list of individual requirements. Each requirement must be written using a single sentence. The composition of these sentences is what characterizes the expected feature behavior, and is the input for the test generation and automation strategy we propose. Evidently, there are some automatic strategies to scrap textual descriptions to extract the requirements and convert the language. However, these tasks are beyond our current scope, and thus they will be indicated as future work.

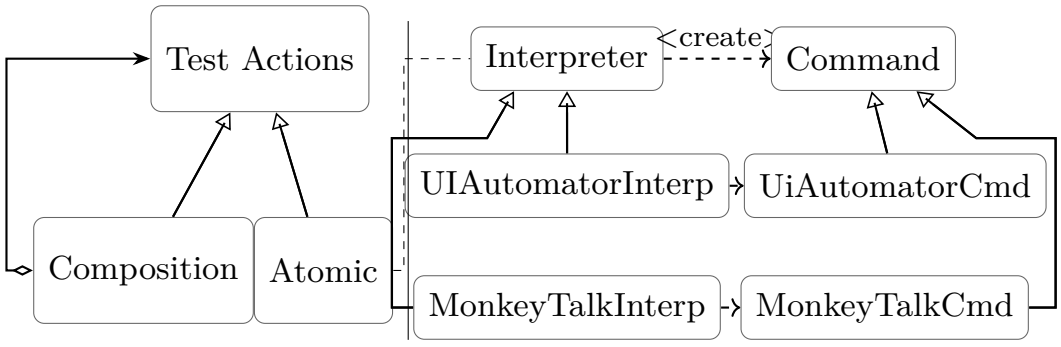
Test Cases

Test cases may be presented in different templates, but they usually have the same basic fields: initial setup, test steps and expected results. TCs are supposed to provide instructions, by interpreting the requirements, on how to check a given scenario. When a TC is written in natural language by hand, issues regarding ambiguity and imprecision often arise.

Test Actions

Test actions provide a suitable representation for the underlying abstractions of NL descriptions; they can be represented as recursive structures, inspired by the composite design pattern (GAMMA, 1995) illustrated in Figure 3 (a). They support abstraction layers that allow one to represent atomic operations, test steps, TCs or even test suites using the same structure. The relationship notation used in Figure 3 is based on UML relationships (PILONE; PITMAN, 2005).

Figure 3 – Overall architecture



(a) Composite Pattern (b) Factory for interpreters

Source: The author (2020)

Since our focus is on Graphical User Interface (GUI)-based test case automation, it is worth mentioning that the high-level descriptions of atomic operations are automatically derived from screen interactions, while in composite test actions these NL descriptions are TC titles or step descriptions, among others. Only atomic operations (that are predefined) are mapped into code-level scripts. Here we use an interpreter to a specific framework (Figure 3 (b)), which can be dynamically instantiated using the factory method pattern (GAMMA, 1995). To illustrate the framework agnosticism of our proposal, we consider two different frameworks (see Figure 3 (b)): *UIAutomator* and *MonkeyTalk*, with their respective interpreters.

Domain Model

Many issues arise from the abstraction gap between requirements and executable test scripts. Different roles in the testing process are involved on the tasks of providing details to abstract behavior descriptions. In practice, there is no single artifact or role responsible for describing the software behavior thoroughly and exaustively. Any Model-Based Testing (MBT) strategy may be destined for failure if it tries to encompass the entirety of a feature in a single artifact or by a single person. For this reason, we propose the adoption of a domain model: a live document in which further details and constraints about the SUT can be added without modifying the original behavior described by the requirements. The advantage is that a formal domain model can be used to generate executable test scripts and to execute verifications automatically, while the actual dynamics of who are able to edit this model and what details should be inserted can be freely defined by team conventions. To preserve the original behavior described by the requirements, there are restrictions on the type of information that can be added. The domain model can only express associations, i.e., relations that link actions to each other, such as dependencies, cancellations, composition, among others.

1.3.3 Test Case Generation

Instead of creating *ad-hoc* algorithms to generate test cases from the specification artifacts, we define formal semantics for these artifacts in the CSP process algebra, and reason about them using well-established techniques and tools. To reason about relevant properties that TCs should have, we use a testing theory for CSP models (NOGUEIRA; SAMPAIO; MOTA, 2014) and an associated conformance relation denoted *cspio*.

Notation

Typical MBT approaches rely on using a formal model to generate test cases. Therefore, in practice, testing teams that use these approaches usually have two different specification sources: the one written in natural language by the product owner; and a formal model

manually derived from it. This situation leads to inconsistencies in the testing process, specially when the specification changes, increasing maintenance costs and escaped defects. To avoid this problem, we propose an approach to derive a CSP model automatically from requirements written in the CNL we propose. This approach is based on extracting circumstances from the subordinate clauses found in the requirements and then creating test scenarios for each circumstance mutation, that are further translated into a formal model to generate intelligible and/or automated test cases. The generated CSP model is actually a formal semantics for the input requirements.

Consistency Analysis

Test scripts always depend on some conditions to execute properly. These conditions are present in different artifacts: environment configuration, test case setup and even test step preconditions. Regardless of the language or method used to define these conditions, there must be a resolution mechanism to interpret them and find a valid and optimal sequence of steps to be executed. In practice, those mechanisms are often ad-hoc implementations based on topological sorting that cannot solve specific dependency relations for testing and do not guarantee optimal solutions. We then propose a formal dependency solver that handles regular dependencies in addition to other useful relations for testing, such as cancellation, instantiation and consistency choice.

1.3.4 Test Case Automation

Test automation becomes a potentially easier task when tests are consistent and the domain model is complete. Basically, the strategy should match atomic actions to their corresponding execution script in a given test driver. Because all other actions are just a composition of more concrete ones, then each one is automatically implemented via its basic actions.

When dealing with legacy test cases, however, those premises cannot be assured. We can identify in the related literature several strategies to automate already created test cases, such as coding and Capture & Replay (Capture-and-Replay (C&R)). While TC automation via coding is slow and forces developers to acquire in-depth knowledge, C&R approaches are usually faster, but suffer from high maintenance costs later on due to poor reuse, as noted in a previously conducted empirical assessment (LEOTTA et al., 2013a). These drawbacks, associated with the use of an ambiguous language to specify requirements, negatively affect a direct mapping from test descriptions to scripts, demanding from testers the creation of a structured representation to enable an automatic and efficient mapping (OSTRAND; BALCER, 1988). On the other hand, practical experiences in testing have shown that forcing programmers to adopt new notations is not the best option, reinforcing the use of well-known notations and environments (BERTOLINO, 2007; GRIESKAMP, 2006). In this light, we offer strategies to improve automation speed and

minimize maintenance problems by improving reuse and by adopting a declarative approach for execution dependencies with the help of the domain model (delegating to an automatic analysis mechanism instead of manually creating test setups, for instance).

1.4 CONTRIBUTIONS

The main contributions of this work are the following:

- A single CNL to describe requirements, domain model and test cases.
- A denotational semantics in Machine readable syntax for CSP (CSPm) for requirements and domain model written in compliance with our CNL.
- A formal support for further specification, *via* domain model, aside from the original requirements, while preserving the underlying behavior.
- A formal definition for test consistency, a property that ensures that test cases, formerly abstract, can be directly executed in an implementation without inconclusive or inconsistent results.
- A flexible strategy for consistent test automation that accepts both CNL-compliant and legacy test cases written in natural language.
- Tools implementing both the direct engineering from requirements (*SmarTest*) and the automation of legacy test cases (*Force IDE* and Kaki).
- Empirical evidence, through experiments and evaluations, demonstrating the practical application of our approach in an industrial context.

1.5 DOCUMENT ORGANIZATION

In the following chapter, we propose a CNL and present its grammar, parsing rules, and the framework chosen for implementation. Chapter 3 introduces CSP and the semantic rules for translating a syntactic tree, obtained after parsing requirements and domain model that comply with the CNL, into a CSP model. In Chapter 4, we introduce the *ioco* testing theory, together with its CSP adaptation (cspio). Then, we discuss how to generate sound and consistent test cases by means of CSP refinement assertions. With the resulting traces, we show how to linearize them back to natural language and how to map them into automated scripts. In addition, we also present in Chapter 5 how to automate legacy test cases while still making sure that the resulting test scripts are consistent. Chapter 6 presents the tools built to implement our strategy and provides some evidence (observations and experiments) of their efficacy. Finally, Chapter 7 discusses related work, while Chapter 8 presents our conclusions, summarizes the work and discusses next steps.

2 PARSING CNL-COMPLIANT REQUIREMENTS

The pursuit of an integrated testing process eventually has to face the following question: *which notation should be adopted?* As it is further discussed in Chapter 7, there are several proposed frameworks which adopt *ad-hoc* notations to represent the requirements and the generated artifacts. However, one major downfall of adopting such notations is to find (or train) specialists. Thus, the answer to our initial question may indicate the use of a widely known notation: our written natural language. It is ubiquitous and expressive, allowing to: refer to other entities for further description (Abstraction gap and traceability); express all kinds of artifacts (Heterogeneous notations); use mature mechanisms for translation (Internationalization); be easily interpreted by humans (Legibility).

Yet, because of their ambiguous nature, natural languages may not be always deterministically interpreted by an algorithm. Then, there is no straightforward way to define a meaningful mapping between textual descriptions and their semantic representations. This lack of precision typically leads to, for instance, an abstract gap between sentences and the actual corresponding execution code – for example, different test engineers may describe the same action using different sentences (regarding vocabulary and syntactic structure). Considering the simple act of downloading an e-mail attachment, some writing alternatives are as follows:

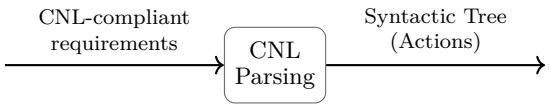
- Tap on attachment icon at the header portion of the message.
- The user should tap on the icon at the header.
- Step: Tap icon; Container: header; Actor: User.

Besides the ambiguities that may arise, which cripples the sentence interpretation by a human tester, it becomes difficult for any strategy to find an accurate and deterministic match. The NL flexibility may result in different stored actions that convey the same meaning, hindering the potential reuse, impairing maintenance and providing wrong test results.

In this light, we propose the use of an CNL, allowing the users to write artifacts in English while certifying that a standard is being followed, leading to a deterministic machine interpretation. Figure 4 presents the process task (extracted from Section 1.3.1) that is detailed in this chapter.

In what follows, we discuss the aspects considered when building the CNL and how it was implemented. Section 2.1 shows the main requirements that a CNL must comply with to provide concise writing and reading. In Section 2.2 we briefly discuss the frame theory used as inspiration to model both grammar and semantic representation. In Section 2.3 we present the syntax of our CNL together with some sample statements. Then,

Figure 4 – Process Task: CNL Parsing



Source: The author (2022)

Section 2.4 introduces the framework used to implement our strategy and discusses some implementation details. Finally, Section 2.5 presents some final remarks.

2.1 GENERAL REQUIREMENTS FOR A CNL

By observing a large number of requirements, test cases and their corresponding test steps in our industrial context, it was possible to formulate a writing pattern which covers the different writing styles found. From this pattern we created a standard (CNL), which is summarized in Section 2.3. However, to elaborate a concise CNL there are some concerns that must be assessed. Even though CNLs are known to be easier to understand, users may find hard to write statements that comply with the syntactic restrictions. According to Kuhn (2013), there are some approaches that could be taken to ease this task:

Error messages - It consists of a basic and straightforward approach, however it does not always provide useful messages for most users (DIMITROVA et al., 2008).

Conceptual authoring - It consists of providing to the user menu operations or well-formed sentences with some gaps to fulfill (HALLETT; SCOTT; POWER, 2007).

Predictive editors - It shows all possibilities of fulfilling a sentence while it is being built, allowing a natural and guided writing (BERNSTEIN; KAUFMANN, 2006).

In our work, predictive editor (along with error messages) is the most fitting approach since users are familiar with search engines and IDEs that provide this functionality. However, to allow an efficient implementation of this approach, some grammar requirements must be met (KUHN, 2013):

Concreteness - Sentences that comply with a CNL should be concretely read and interpreted by computer programs, with syntax trees being automatically built and ambiguities spotted and mitigated beforehand.

Declarativeness - A given CNL grammar should be declarative in the sense that it does not depend upon a specific *modus operandi*. It should keep a clear separation between its syntax and the parser algorithms that process it.

Lookahead Features - The design/construction of CNLs should consider their efficiency when implementing lookahead features. Whenever an unfinished sentence is given, the CNL should allow the retrieval of multiple valid endings.

Anaphoric References and Scoping - CNLs should support expressions that refer to previously mentioned entities (such as "it"), besides defining their scope. Even though this problem is treated by semantic and discourse analysis interpreters, it should also be regarded syntactically.

Dynamic lexicon - It should be possible to change or extend the dictionary dynamically with no programming intervention.

The fulfilling of these requirements are pursued and discussed in the next sections. While some of them are solved by the adopted semantic foundation, others were outsourced to an specialized framework for natural language design and parsing (Grammatical Framework - Section 2.4.1)

2.2 FRAME STRUCTURE

To deal with the *Concreteness* and *Declarativeness* requirements, we adopted a framework for knowledge representation that allows computer interpretation. The chosen schema is heavily built upon the concept of a *frame*, which is a structure to store data about a previously known situation, as defined in (MINSKY, 1975):

> "A frame is a data-structure for representing a stereotyped situation [...] We can think of a frame as a network of nodes and relations. The 'top levels' of a frame are fixed and represent things that are always true about the supposed situation. The lower levels have many terminals—'slots' that must be filled by specific instances or data. Each terminal can specify conditions its assignments must meet. (The assignments themselves are usually smaller 'sub-frames'.) Simple conditions are specified by markers that might require a terminal assignment to be a person, an object of sufficient value, or a pointer to a sub-frame of a certain type [...]."

These frames contain prefixed *slots* that, when filled, represent an instance of a specific situation. Each slot holds a different purpose. Consequently, there can be a specific rule (or set of rules) for filling each slot. As an example, a rule can be a restriction on a specific set of elements (enumeration), a regular expression, among others. For instance, a slot related to birthday information can only be filled by a valid date. Furthermore, each slot must have an associated default value, which is used when not all slots of a frame can be filled by the available information. In practical terms, we can think of a pre-filled frame whose slots content is overridden by concrete instances, when available.

Our work was also inspired by the linguistic approach to semantic representation known as Case Frames (FILLMORE, 1968), which center around the verb. Each verb defines its own frame, since verbs complements may vary (e.g., intransitive verbs do not require

any complement, differently from transitive verbs). Yet, as this theory aims to provide semantic interpretation, the slots do not represent syntactic classes (such as subject and object). Instead, each slot holds some information about the action represented by the verb. In the simplest case, the verb indicates an action performed by an agent towards a patient. For instance, *John* (agent) *bought* (verb) *a book* (patient).

Because our purpose is to ultimately represent test actions, our frames should convey elements that resemble test steps. In our context, the "agent" slot defaults to the user, since our frames aim to describe how a user (tester) can interact with the SUT. The verb slot is named as 'operation', and its immediate complement is always the 'patient' (see Table 1).

Table 1 – Frame example

Required (Static)			Extra (Dynamic)		
Agent	Operation	Patient	Sender	Receiver	Title
(User)	Send	Email Message	fmca@cin.ufpe.br	acas@cin.ufpe.br	Smartest

Source: The author (2022)

In our scenario, in which test case automation is a primary goal, we adapted the Frame theory by dividing the slots into two categories: Required and Extra. The former must be present in every frame, acting as a unique identifier, being static in the sense that it cannot be changed without changing the intention of an action. The latter, instead, does not define the action intention, but all other dynamic properties and modifiers of a situation. This separation allows us to use the required slots as unique identifiers (when mapping to script methods) and pass the dynamic slot values as arguments.

2.3 SYNTAX

Our main goal when designing the CNL was to be able to write any test artifact using the same subset of natural language and following the same rules. This decision has the advantage of: facilitating the communication among stakeholders, since they already know the universal notation; and bypass the need for converting the test case descriptions once they are generated from the requirements. To provide a general idea of which sentences comply with our CNL, we present in Listing 2.1 an overview of the grammar (excerpt) using the Extended Backus-Naur Form (EBNF) notation.

Listing 2.1 – Grammar syntax using the EBNF notation

$$\langle \text{Sentence} \rangle \models \langle \text{Circumstance} \rangle? \ \langle \text{Statement} \rangle \tag{2.1}$$

$$\langle \text{Circumstance} \rangle \models \langle \text{SingleEvent} \rangle? \mid \langle \text{Recurrence} \rangle \mid \langle \text{Conditional} \rangle \tag{2.2}$$

$$\langle \text{Statement} \rangle \models \langle \text{Task} \rangle \langle \text{SimultaneousTask} \rangle? \tag{2.3}$$

$$\langle \text{SimultaneousTask} \rangle \models \texttt{at the same time} \ \langle \text{Task} \rangle \tag{2.4}$$

$$\langle \text{Task} \rangle \models \langle \text{Action} \rangle \langle \text{Modifier} \rangle * \mid \langle \text{Conjunction} \rangle \mid \langle \text{Disjunction} \rangle \tag{2.5}$$

$$\langle \text{Modifier} \rangle \models \langle \text{Destination} \rangle \mid \langle \text{Sender} \rangle \mid \langle \text{Iterable} \rangle \tag{2.6}$$
$$\langle \text{KeyValueModifier} \rangle \mid [...]$$

$$\langle \text{KeyValueModifier} \rangle \models \texttt{with} \mid \texttt{using} \ \text{“}\langle \text{SlotValue} \rangle\text{”} \ \texttt{as} \ \langle \text{SlotKey} \rangle \tag{2.7}$$

$$\langle \text{Action} \rangle \models \langle \text{Agent} \rangle? \langle \text{Modality} \rangle? \langle \text{Polarity} \rangle? \tag{2.8}$$
$$\langle \text{Operation} \rangle \langle \text{Patient} \rangle \langle \text{PredicativeQualifier} \rangle?$$

$$\langle \text{Actor} \rangle \models \langle \text{Agent} \rangle \tag{2.9}$$

$$\langle \text{Patient} \rangle \models \langle \text{Reference} \rangle \tag{2.10}$$
$$\mid \langle \text{Quantifier} \rangle? \langle \text{AttributiveQualifier} \rangle? \langle \text{Object} \rangle$$
$$\mid \langle \text{PatientConjunction} \rangle \mid \langle \text{PatientDisjunction} \rangle$$

$$\langle \text{Qualifier} \rangle \models \langle \text{AttributiveQualifier} \rangle \mid \langle \text{PredicativeQualifier} \rangle \tag{2.11}$$

$$\langle \text{PatientConjunction} \rangle \models \texttt{and} \ \langle \text{ListPatient} \rangle \tag{2.12}$$

$$\langle \text{PatientConjunction} \rangle \models \texttt{or} \ \langle \text{ListPatient} \rangle \tag{2.13}$$

$$\langle \text{ListPatient} \rangle \models \langle \text{Patient} \rangle \mid \langle \text{Patient} \rangle \langle \text{ListPatient} \rangle \tag{2.14}$$

$$\langle \text{Conjunction} \rangle \models \langle \text{Action} \rangle \ \texttt{and} \ \langle \text{Action} \rangle \tag{2.15}$$

$$\langle \text{Disjunction} \rangle \models \langle \text{ExclusiveDisjunction} \rangle \mid \langle \text{InclusiveDisjunction} \rangle \tag{2.16}$$

$$\langle \text{ExclusiveDisjunction} \rangle \models \langle \text{Action} \rangle \ \texttt{either} \ \langle \text{Action} \rangle \tag{2.17}$$

$$\langle \text{InclusiveDisjunction} \rangle \models \langle \text{Action} \rangle \ \texttt{or} \ \langle \text{Action} \rangle \tag{2.18}$$

$$\langle \text{SingleEvent} \rangle \models (\texttt{when} \mid \texttt{after}) \ \langle \text{Condition} \rangle \tag{2.19}$$

$$\langle \text{Recurrence} \rangle \models \langle \text{PreRepetition} \rangle \mid \langle \text{PostRepetition} \rangle \tag{2.20}$$

$$\langle \text{PostRepetition} \rangle \models (\texttt{until} \mid \texttt{till}) \ \langle \text{Condition} \rangle \tag{2.21}$$

$$\langle \text{PreRepetition} \rangle \models \texttt{while} \ \langle \text{Condition} \rangle \tag{2.22}$$

$$\langle \text{Conditional} \rangle \models (\texttt{if} \mid \texttt{in case} \mid \texttt{given}) \ \langle \text{Action} \rangle \tag{2.23}$$

$$\langle \text{Modality} \rangle \models \langle \text{RequiredModality} \rangle \mid \langle \text{AbilityModality} \rangle \tag{2.24}$$

To simplify the presentation of the grammar, we begin by illustrating simpler sentences that comply with intermediate production rules. From these, we add more complex configurations until we reach the starter symbol. For instance, consider the production rule 2.8. It states that an action is required to have an Operation and a Patient. Thus, the following basic sentence would be successfully recognized:

A message is sent.

The word "message" would be the Patient while "send" would be the Operation. Additionally, an action can have Agent, Modality, Polarity and a Predicative Qualifier. To illustrate a sentence with all these slots, we have:

The user must not send a message early.

In this case, we incremented the initial sentence by adding the "user" as Agent, a Required Modality (represented by the modal verb "must"), the Negative Polarity and a Predicative Qualifier (adjective "early"). Also, the patient slot can be further incremented by using one of the rules in 2.10. For instance, we could add an anaphoric reference by replacing "message" for the pronoun "it". Also, we could add an attributive adjective for the object in question, or even a quantifier. Finally, we could use coordinating conjunctions to join two patients and form a more complex one (noun phrase coordination). We illustrate these possibilities in the following sentence:

Two notifications and a small message are sent.

In this example, we have a coordinate patient by the coordination of two other patients. Since they are linked by an "and", then we consider this as a conjunction (instead of disjunction). In the first element of the conjunction, we have "two" as the number of messages (quantifier). The second element has an attributive qualifier associated to it (adjective "small").

When observing the initial production rule 2.1, we highlight that all sentences must have a statement, which eventually is described by an action or coordinating actions. The statement is, in practical terms, the task that should be performed. When parsing test cases, each sentence represents a step that should be executed on the SUT. When the sentence is a requirement, it usually describes what must be the observable behaviour of the SUT after receiving an input or when a condition is met. This way, sentences can have a Circumstance, which can be a Single (or recurrent) Event or a Condition. Each Circumstance has a corresponding preceding conjunction, namely "when", "if" and "until" (and their synonyms). For instance, the following sentence has a circumstance:

When the button is pressed then a message is sent.

Ultimately, both Circumstances and Statements are described by means of Actions, allowing us to use a single structure for conditions and tasks, which is easier for the user to grasp. Additionally, since conditions must be transformed into actions to be executed (to ensure, in test cases, that the conditions are met), this unified solution is fit for that purpose. In this particular example, we have a Circumstance, namely a Single Event,

represented by the Action “button is pressed”. The statement is described by the Action “message is sent”, as previously discussed.

More complex configurations can be found by exploring the other production rules, but the above examples represent the core language we defined. Implementation details will be discussed in Section 2.4).

2.3.1 Building a Frame

We use the parsed data to fill out the required and build the extra slots of a frame, as discussed in Section 2.2. The production rule 2.5 contains all information needed to fill out the frame slots. The first non-terminal leads to the production rule 2.8, in which the Agent, Operation, and Patient slots are extracted. Then, from the rule 2.6, all extra slots are progressively filled out. It is worth mentioning that the extra slots are not restricted to the ones already established in the grammar, such as “Sender”, “Destination” etc. For instance, consider the following sentence:

Send a message using **“Thesis” as Title** *and* **“high” as Priority**.

The new slots “Title” and “Priority” were created and filled out on-the-fly, similarly to variable naming in programming languages.

2.4 CNL IMPLEMENTATION

The CNL requirements and Backus-Naur Form (BNF) both give an idea of what sentences comply with our standard. However, other technical and functional issues arise when implementing these constraints. The following sections present our implementation choices and how these issues were solved. First, we introduce the framework used to define the grammar. Then, we show how we use its API for parsing and linearization. Finally, we discuss the pre- and post-processing tasks, besides optimization techniques adopted to improve performance.

2.4.1 Grammatical Framework

The Grammatical Framework (GF) is not only a theoretical framework but also a special-purpose programming language for the description of natural languages (ANGELOV, 2011). It was designed to have an out-of-the-box support for the complexities found in natural languages (RANTA, 2011). While it provides the special tools for the design and implementations of such languages, it also incorporates computational resources created by a wide range of experts for 26 (twenty-six) languages, including complete morphology and grammar rules for more than 20 (twenty) languages (ANGELOV, 2011).

Every GF grammar is composed by one abstract syntax and one or more concrete syntaxes. Abstract syntaxes model a specific application domain (similarly to ontologies),

whereas concrete syntaxes are language-dependent and thus the latter encode a particular idiom. Because there is a separation between the syntaxes, the abstract one can be used as interlingua between multiple concrete languages (ANGELOV, 2011). This notion of abstract and concrete syntaxes is similar to the concept of deep and surface structures from Chomsky (1971) in which deep structures are underlying constructs that unify the surface forms. We can also build upon this linguistic theory to unify the different ways of describing an interaction (for instance, active or passive voice) with a single frame.

Building the CNL Grammar

We start by enumerating the abstract types that constitute a Sentence. These types correspond to the Frame slots previously defined, and they are expressed as categories in the GF. For instance, we can have the following categories:

cat Sentence; Operation; Object; Patient;

Following, it is possible to enumerate the elements of each category as a parameterless function. Each function is a placeholder that can be concretized later. To illustrate, we list below some operations.

fun Press, Hold, DoubleClick: Operation;

Objects can also be defined as functions with zero arity. An Object differs from a Patient only to reflect the nuances of natural language, such as inflections.

fun Widget, Button, Notification: Object;

Once we enumerate the Objects, we need a function that transforms them into a Patient. Below we present an example of grammar rule for nouns number inflection (single or multiple). Remind that nouns are Object slots in our Frames.

fun Single, Multi : Object → Patient;

Finally, we define a function that joins Operation and Patient to get a complete Sentence. This function must also be implemented by the concrete grammar.

fun Action: Operation → Patient → Sentence

Once we have the function “Action” defined, it can be called by passing the requirement parameters, similarly as in any other programming language. For instance, to build a sentence with the “Press” operation and multiple “Button” objects, we have:

Action Press (Multi Button)

So far, we have defined only the abstract grammar, similar to method signatures and class definitions in object-oriented programming. In order to provide a mapping between these concepts and actual strings, we must tell GF how to *linearize* them. We must define how a category must be represented internally and, then, how each function is concretized:

lincat Operation = {s: Str};
lin Press = {s = “pressed”};
lin Hold = {s = “held”};
lin DoubleClick = {s = “double clicked”};

Since Object linearization must be in the plural form when they are multiple, the definition considers a parameter “Number” (singular or plural), which can be used to change the string based on the number of objects:

param Number = Sg | Pl;
lincat Object = Number => Str;
lin Widget = table {Sg => “widget”; Pl => “widgets”; };
lin Button = table {Sg => “button”; Pl => “buttons”; };
lin Notification = table {Sg => “notification”; Pl => “notifications”; };

Finally, we linearize the function that builds a sentence. To keep it simple, we illustrate here only sentences with passive voice with the modal verb “must”.

lincat Sentence = {s: Str};
lin Action op pat = {s = “the” ++ pat.s ++ “must be” ++ op.s};

Two files must be created so GF can compile the grammar. Next we show the abstract grammar file (Listing 2.2) and the concrete grammar file (Listing 2.3) by putting together all previous excerpts.

Listing 2.2 – Abstract grammar – simple example

```
abstract Smartest = {
    cat Sentence; Operation; Object; Patient;
    flags startcat = Sentence;

    fun Press, Hold, DoubleClick: Operation;
    fun Widget, Button, Notification: Object;
    fun Single, Multi : Object -> Patient;
    fun Action: Operation -> Patient -> Sentence;

}
```

Listing 2.3 – Concrete grammar – simple example

```
concrete SmartestEng of Smartest = {

    param Number = Sg | Pl;

    lincat Operation = {s: Str};
    lin Press = {s = "pressed"};
    lin Hold = {s = "held"};
    lin DoubleClick = {s = "double clicked"};


    lincat Object = Number => Str;
    lin Widget = table { Sg => "widget"; Pl => "widgets" };
    lin Button = table { Sg => "button"; Pl => "buttons" };
    lin Notification = table { Sg =>"notification"; Pl => "notifications" };

    lincat Patient = {s: Str};
    lin Single obj = {s = obj!Sg};
    lin Multi obj = {s = obj!Pl};

    lincat Sentence = {s: Str};

    lin Action op pat = {s = "the" ++ pat.s ++ "must be" ++ op.s};
}
```

After joining all definitions and rules in the GF files, we can then call the GF command line to compile them.

```
$ gf SmartestEng.gf

- compiling Smartest.gf...   write file Smartest.gfo
- compiling SmartestEng.gf...   write file SmartestEng.gfo
linking ... OK

Languages: SmartestEng
Smartest>
```

Once the files are compiled, GF provides a command-line interface to interact with the defined CNL. For instance, we can ask to generate a random tree in the current abstract syntax:

```
Smartest> gr
Action Hold (Single Button)
```

The output (abstract function) can then be linearized into a string (following the rules we implemented earlier):

```
Smartest> l Action Hold (Single Button)
the button must be held
```

Not only an abstract tree can be linearized into a string, but the opposite can be achieved by parsing an enquoted string.

```
Smartest> p "the button must be held"
Action Hold (Single Button)
```

Resource Grammar Library

The mechanism of decoupling the nuances of a language from its abstraction allows us to have a common abstract syntax as *interlingua*. Then, each concrete grammar can define how each of these abstract elements are written, and even add specific categories. In addition to having a declarative style, writting multilingual grammars turned out to be possible (RANTA, 2009).

The development of domain specific application grammars, such as the one defined in this work, usually reuses a subset of the natural language syntax and lexicon to reduce ambiguity. To ease this burden, GF provides a Resource Grammar Library (RGL), which covers comprehensive morphologies and syntactic structures from more than 20 languages (GRUZITIS; PAIKENS; BARZDINS, 2012). In this light, instead of defining from scratch the inner workings of a natural language, we can reuse the mature syntactic and morphologic paradigms, detailed by specialists, already present in the RGL.

For instance, to create a full-fledged sentence using RGL, we could use the mkS function with the following signature:

$$\text{mkS} \quad (\text{Tense}) \rightarrow (\text{Ant}) \rightarrow (\text{Pol}) \rightarrow \text{Cl} \rightarrow \text{S}$$

Source: The author (2022)

This function must receive a tense (conditional, future, past or present), an anteriority (anterior or simultaneous), polarity (positive or negative), and a declarative clause to be adapted. Below we have an example of use:

```
RGL> l mkS conditionalTense anteriorAnt negativePol (mkCl she_NP sleep_V)
"she wouldn't have slept"
```

A clause, in turn, also has an overloaded function called mkCl which usually receives a noun phrase and a verb (in addition to other complements, such as adverbs and adjectives):

$$\text{mkCl} \quad \text{NP} \rightarrow \text{V3} \rightarrow \text{NP} \rightarrow \text{NP} \rightarrow \text{Cl}$$

Source: The author (2022)

We can illustrate the use of this function with the following example, which uses a three place verb:

```
RGL> l mkCl she_NP send_V3 it_NP he_NP
"she sends it to him"
```

Finally, we can refactor the concrete grammar presented in Listing 2.3 by reusing the RGL elements now presented. Listing 2.4 shows the linearizations by means of RGL syntactic elements. The abstract types, such as Operations and Patients, can be directly mapped to verbs and noun phrases, respectively. The functions borrowed from RGL (mkS, mkCl, etc.) handle the morphological inflections and additional details.

Listing 2.4 – Grammar refactored using RGL

```
concrete SmartestEng of Smartest = {

    lincat Sentence = S, Operation = V, Object = N, Patient = NP;
    lin Press = press_V, Hold = hold_V, Widget = widget_N, Button = button_N,
        Notification = notification_N;

    lin Single obj = mkNP a_Det obj;
    lin Multi obj = mkNP the_Det obj;


    lin Action op pat = mkS presentTense positivePol (mkCl pat must_VV (passiveVP
        mkV2 op))
}
```

API

In the previous section we discussed how to define our grammar using GF. To actually embed this gramar into a tool, we used its API through the available Java binding to the C runtime. After all sources are compiled successfully, a PGF (Portable Grammar Format) file, which is a machine readable format for GF, is created (GF, 2022). Listing 2.5 shows how to read this file programatically and how to access the previously defined concrete language.

Listing 2.5 – Reading a PGF file using the GF Java Binding

```
PGF pgf = PGF.readPGF("Smartest.pgf");
Concr smartestEng = pgf.getLanguages().get("SmartestEng")
```

Having access to the concrete language, we can use it to parse or linearize expressions. For instance, Listing 2.6 illustrates how to parse a sentence and access the syntactic tree. First, the sentence is provided as a string, together with the start category defined in the grammar. Then, a list of all possible abstract trees are returned, sorted by decreasing probability, since a single sentence can be parsed in multiple ways. The parser implementation is lazy, returning each tree as soon as it is ready, instead of executing a full seacrh. (GF, 2022).

Listing 2.6 – Parsing a sentence using the GF Java Binding

```
Iterable<ExprProb> iterable = smartestEng.parse(pgf.getStartCat(), "a message must
    be sent");
ExprProb exprProb = iterable.next();
```

```
System.out.println(ep.getExpr());
// Sent_Stmt (Stmt_All Positive (Action_noAgent (Modal_Required Must) Send_CatOp (
    Pat_Obj UndefinedDeterminerSingular Message_CatObj)))
```

When an action is identified at the syntax tree provided by GF, an analysis process is executed, and the action is converted back to a text. This is achieved by passing the same expression to the linearization function of the concrete language, as in Listing 2.7.

Listing 2.7 – Linearizing a expression using the GF Java Binding

```
String exprStr = "Sent_Stmt (Stmt_All Positive (Action_noAgent (Modal_Required Must)
     Send_CatOp (Pat_Obj UndefinedDeterminerSingular Message_CatObj)))"
Expr expr = Expr.readExpr(exprStr);
System.out.println(smartestEng.linearize(expr));
// a message must be sent
```

In our scenario, not only we need to parse a sentence but also to manipulate the parse tree in order to linearize it in different ways. For instance, a test step must always be in the imperative mood, while requirements may be presented in the indicative mood. Even though the action (identified by its frame slots) remains the same, its linearization may differ depending on the artifact. A simple mechanism for manipulating the expressions is detailed in Listing 2.8. In short, we change only the leading functions, while maintaining the original elements, to linearize the parse tree in a different tense.

Listing 2.8 – Manipulating the abstract tree output by GF

```
String impExprStr =  exprStr.replace("Sent_Stmt", "Sent_Imp").replace("Stmt_All", "
    ImpStmt")
// Sent_Imp (ImpStmt Positive (Action_noAgent (Modal_Required Must) Send_CatOp (
    Pat_Obj UndefinedDeterminerSingular Message_CatObj)))

Expr impExpr = Expr.readExpr(impExprStr);
System.out.println(smartestEng.linearize(impExpr));
// send a message
```

2.4.2 Dynamic lexicon

Because the native GF API does not allow dynamic types, in the sense of creating new categories on-the-fly, we had to create an external mechanism to dinamically extend the initial vocabulary. We considered creating a comprehensive and fixed grammar to avoid dinamically updating the lexicon, but this option was not feasible due to performance bottlenecks while parsing.

The implemented external mechanism was based on a template engine: we render a new grammar file (source code) by passing the new identified terms as input data. We defined the rules on how the output file should be generated based on these parameters. Listing 2.9 shows a code snippet using the syntax of the Freemarker template engine (APACHE, 2022). It receives all identified operations on the parameter “operations” and, following, each operation “op” is rendered as an “Operation”.

Listing 2.9 – Template for the grammar

```
<#ftl encoding="utf-8" auto_esc=false>
abstract Smartest = {
    [...]
    cat <#list operations as op>${op.getCatId()}<#sep>, </#list> : Operation;
    [...]
}
```

Listing 2.10 provides a sample output generated from the template above, assuming that three operations were identified (Send, Press and Check). To give sample output that could be generated from the template above and assuming that three operations were identified (Send, Press and Check), we can check the Operations, patients, qualifiers and other elements are dynamically generated using this method. It allow users to extend the dictionary without recompiling the tool as a whole.

Listing 2.10 – Sample output based on the grammar template

```
abstract Smartest = {
    [...]
    cat Send_Op, Press_Op, Check_Op: Operation;
    [...]
}
```

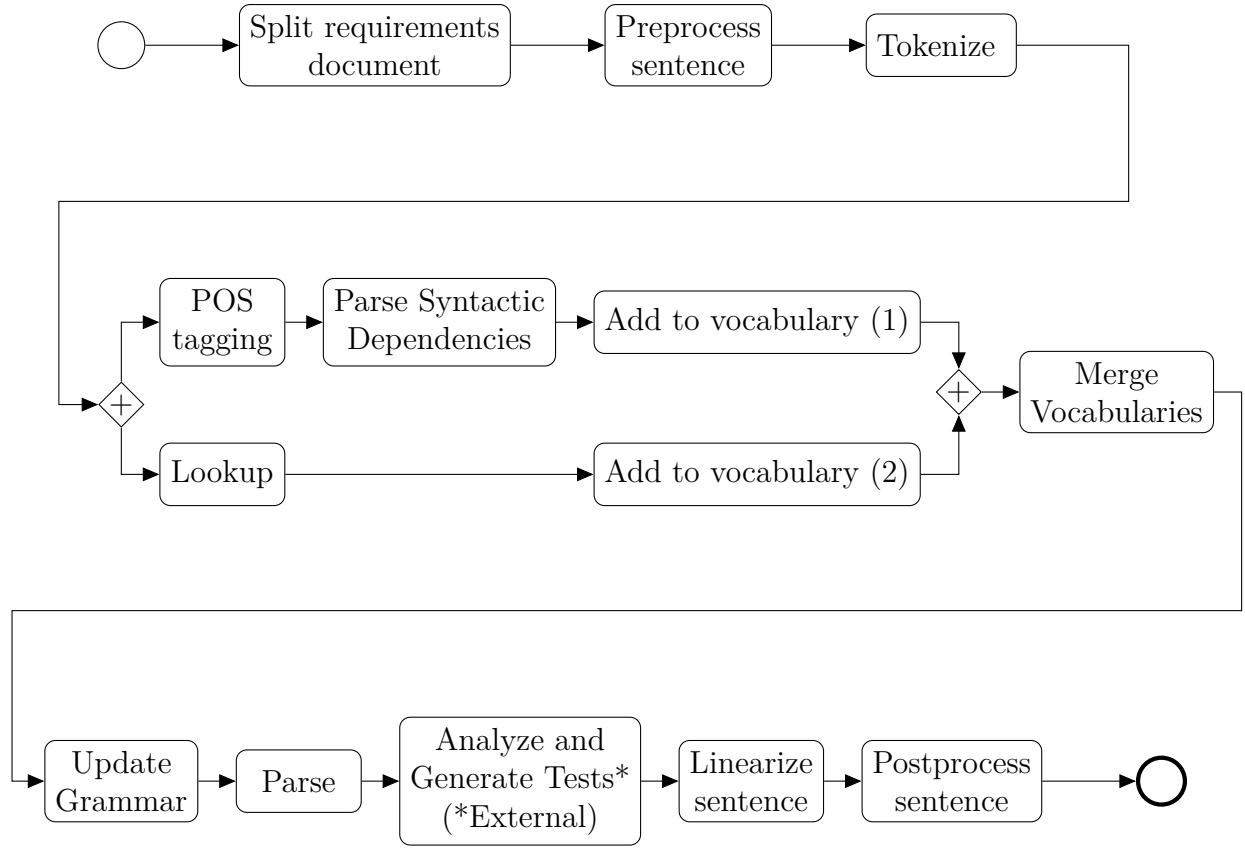
2.4.3 Processing and Optimizations

When dealing with entire text documents as input, there are some functionalities that are not covered by GF. Since GF deals mostly with sentences, we created our own document structure to define how the sentences are interconnected. Besides the general structure, we also created a pipeline of pre- and post-processing to modify the sentences supplied to or from GF.

We already have a minimal and fixed dictionary to recognize the most common operations and patients for toy examples. However, real world examples will contain words that are not covered by this basic dictionary. For this reason, we implemented a mechanism to automatically identify the vocabulary from the given sentences without human intervention. All these processing tasks are illustrated in Figure 5 and discussed further.

First, the requirements document is split into sentences, which is an easy task, since these documents are divided into sections that, in turn, contain bullet lists. There is a main section named “Requirements”, in which each requirement is specified using a single sentence in each bullet item. Next, each sentence is preprocessed: captions, annotations and quote marks are replaced by placeholders that are later overwritten at the post-processing step. Following, each sentence is tokenized and its words are individually analyzed. From here, the processing is split in two parallel flows: the first tries to determine the syntactic categories while the latter adds all already known categories associated to each word. The first flow is carried out by CoreNLP (MANNING et al., 2014) and the inferred vocabulary is stored (1). The second flow is a simple lookup for all syntactic categories associated to

Figure 5 – NL processing tasks



Source: The author (2022)

a given word in the WordNet lexical database (MILLER, 1998), also storing the extracted vocabulary (2). This supposedly "redundant" strategy creates a vocabulary with not only the traditional categories associated to the word, but also the unusual ones that only make sense when occurring in a odd phrase. Because our domain deals with acronyms and idiosyncratic terms, human intervention was reduced.

Then, the vocabularies (1) (2) are merged, which triggers a grammar update to include the new frame slots (nouns as patients and verbs as operations, for instance). With the updated grammar, then a sentence can be parsed. The intermediate representation obtained after parsing is used in the semantic analysis and test generation (both described in later chapters). When this representation is required to be read in English (e.g., test steps for manual TCs), it is linearized back to natural language using the GF framework (as previously shown). Finally, annotations that were extracted at the preprocessing stage are written back, when required.

This is a cumbersome pipeline and it affects the time to parse the dozens of sentences usually present in a requirement document. To minimize this impact, two points of caching were created: 1) before the vocabulary identification; and 2) before updating the grammar. It is common to analyze the requirement documents multiple times while it is

being written. Thus, instead of processing the same sentences again, only the new requirements are actually handled. Since the vocabulary identification (specially by CoreNLP) is demanding, this alone already gives a performance boost. Another demanding step is the GF grammar update. Because GF has no native support for dynamic lexicon, the entire grammar must be compiled after each change. In this light, this compilation is triggered only when the vocabulary has been *de facto* changed from the previous analysis of the current user / feature.

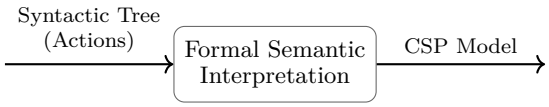
2.5 FINAL REMARKS

The *Lookahead* requirement discussed in Section 2.1 was outsourced to Grammatical Framework, which allowed us to use its API to develop a linear algorithm (on top of the existing ones) to suggest sentences. A polynomial parsing complexity can be achieved when a subclass context-free GF is used (LJUNGLÖF, 2004). The requirement related to *Anaphoric reference and scoping* is partially managed as a structural concern when converting the frames into test actions. Here, the scopes are defined by the composite pattern, and references are implicitly linked by their semantic interpretation within the current context (discussed in the following chapters). Finally, the *Dynamic lexicon* requirement is resolved, as discussed in Section 2.4.2, since the lexicon can be actively changed by the user, which triggers a full grammar update in background.

3 FORMAL SEMANTICS FOR CNL-COMPLIANT SPECIFICATIONS

In this chapter, we define a precise behaviour for CNL-compliant requirement models via mapping into CSP models. More specifically, we map actions and circumstances into CSP events and processes. This approach leverages model checking to automate the conformance verification and TC generation, without requiring the development of ad-hoc algorithms (NOGUEIRA; SAMPAIO; MOTA, 2014). Also, it is easier to reuse and extend previous work by means of process composition. Figure 6 illustrates the input/output transformation detailed in the next sections.

Figure 6 – Process Task: Formal Semantic Interpretation



Source: The author (2022)

First, in Section 3.1 we introduce the basics of the CSP algebra and its machine-readable language for automated verification using the FDR tool[1]. Then, in Section 3.2, we present the denotational semantics of the requirements written in natural language (represented by a syntactic tree, at this point) as CSP events and processes. Finally, in Section 3.3 we present the domain model elements and their semantics.

3.1 CSP

Communicating Sequential Processes (CSP) is a formal notation used to model concurrent systems using algebraic, denotational and operational approaches to reason about their behaviors. The core abstractions of this notation are processes and events. An event is the basic element of the modelling: it represents the occurrence of any given situation. For instance, the action of sending a message can be represented by a single event *send.message*. Of course, this action could be represented by a more detailed sequence of events, as discussed later. The chosen abstraction level depends on what behavior is relevant to model. The other basic elements of CSP are the fundamental processes *Stop* and *Skip*, which model the absence of communication (deadlock) and sucessful termination, respectively.

In addition to those core elements, one can use a rich repertoire of algebraic operators to model the system behavior. *Prefix* is a simple but important operator: it takes an event and a process, yielding a new process such as, for example:

$$send.message \rightarrow Skip$$

[1] https://cocotec.io/fdr/

The resulting process communicates the event *send.message* and, then, behaves as *Skip*. As discussed before, we could also represent the action of sending a message by describing more detailed events, such as the sequence: *open.app*, *write.message*, *press.button*. In this case, we could model this sequence with the following process:

$$P = open.app \rightarrow write.message \rightarrow press.button \rightarrow Skip$$

It means that P communicates *open.app* and then behaves as the unnamed process $write.message \rightarrow press.button \rightarrow Skip$, and so on. We define a process P2, which is equivalent to P, by splitting those steps into separate processes, and then cobine them together using the sequential composition operator (;), as in:

$$S1 = open.app \rightarrow write.message \rightarrow Skip$$
$$S2 = press.button \rightarrow Skip$$
$$P2 = S1;\ S2$$

Processes can also have distinct execution flows by synchronizing on the event that the environment chooses to communicate. For instance, a system may allow a user to submit a message by pressing a button or making a swipe gesture. This behavior may be described by the following process:

$$P = open.app \rightarrow write.message \rightarrow (press.button \rightarrow Skip \ \Box \ swipe.right \rightarrow Skip)$$

The external choice operator ($\Box$) gives opportunity for the environment to communicate either event (*press.button* or *swipe.right*). Then, P synchronizes with the chosen event, resolving the choice deterministically. On the other hand, the internal choice operator ($\sqcap$) does not allow the environment to control which event will be communicated.

$$P = send.message \rightarrow (messagesent \rightarrow Skip \sqcap dataerror \rightarrow Stop)$$

In this example, the choice decision is internal to the process. Because of this unpredicatable behavior, the choice is said to be resolved nondeterministically.

For two processes to run concurrently, there are some operators that can be used. The interleaving operator, for instance, is used for concurrent tasks that do not communicate with each other.

$$P \ ||| \ Q$$

These processes run independently from each other. Meanwhile, when two processes need to communicate (i.e. synchronize), one can use the parallel operator (P |[X]| Q), where X is a set of events in which P and Q synchronize. For instance, we model a user and a device using two communicating processes that synchronize on key events.

$$USER = look \rightarrow mouse.down \rightarrow mouse.up \rightarrow USER$$
$$DEVICE = mouse.down \rightarrow enable.effect \rightarrow mouse.up \rightarrow disable.effect \rightarrow DEVICE$$
$$\text{USER} \ [|\, mouse.down, mouse.up \,|] \ \text{DEVICE}$$

The processes USER and DEVICE run independently until one of the specified events in the synchronization set is reached. Then, only when the same event is also reached by the other process, they synchronize and continue their independent behaviors.

CSP also offers the standard *if then else* operator, as in the example below:

```
COND(action, Allowed) =
  if (action ∈ Allowed)
      then
    action → Skip
  else
    Stop
```

Another useful operator is hiding, which hides events to avoid synchronization from other processes, preventing the environment from influencing the events behaviors. For instance, by hiding the events *open_app* and *send_message*:

$$P = open\_app \rightarrow send\_message \rightarrow Skip$$
$$Q = P \setminus \{open\_app, send\_message\}$$

The process Q will behave, from the perspective of other processes, exactly as the process *Skip*.

Machine Readable CSP

In order to allow the formal notation to be interpreted by automated tools, CSP offers an alternative machine readable notation. This specific version of the language is known as CSP_M. The most popular tool that analyses models in this language is FDR (Failures and Divergences Refinement). This is the tool that we use in the next sections to reason about our modelling and verify the refinement assertions. For future reference, Table 2 shows the equivalent operators of CSP in CSP_M.

Another subtle difference is that, in CSP_M, events are created through channels that must be explicitly declared, possibly with the corresponding types of the communicated values. For instance:

$$channel\ send\_message,\ open\_app$$
$$channel\ delete\_id : Int$$

Typeless channels define a single event, while the typed ones define collections of events. In this case, each event of the *delete_id* channel transmits an Int value. The enumeration of all those events can be represented by the following set $EVENTS$, which can also be represented by the expansion $\{|\ delete\_id\ |\}$:

Table 2 – CSP_M handbook

CSP syntax	**CSP_M syntax**
$\rightarrow$	->
;	;
$\Box$	[]
$\sqcap$	|~|
$\|\|\|$	$\|\|\|$
$\|[\ S\ ]\|$	[|S|]
$\setminus$	$\setminus$

Source: The author (2022)

$$EVENTS = \{delete\_id.1, delete\_id.2, ...\}$$

Finally, CSP_M also offers indexed (replicated) versions of some operators that generalise the binary form to allow the composition of an arbitrary number of processes combined using a same operator. For instance, $||| \ x : \{1..3\} \bullet P(x)$ is equivalent to $P(1) \ ||| \ P(2) \ ||| \ P(3)$. These replicated operators are generally disposed on the format $op \ \langle statements \rangle \bullet P$, in which op is the operator, $statements$ the list of statements and P the process being defined by means of the statements. Another indexed operator we use in our work is the replicated external choice operator. An example would be $[]y : \{a, b, c\} \bullet Q(y)$ which is equivalent to $Q(a)[]Q(b)[]Q(c)$.

CSP semantics and refinements

Refinement relations are used to formally verify whether an implementation is correct regarding to a specification, i.e, one can prove that an implementation satisfies the specification properties if the given implementation is a refinement of the corresponding specification. To establish refinement relations between processes, CSP offers three well-established semantic models: traces, failures, and failures-divergences (CAVALCANTI; GAUDEL, 2007).

The traces model denotes a process P by the sequence of events it may perform. $traces(P)$ is a subset of the reflexive transitive closure of $\alpha P \ \cup \ \{\checkmark\}$, where αP is the alphabet of P (set of events that P can communicate), and ✓ is a special event that indicates that a process has sucessfully terminated. For instance, the process P

$$P = a \rightarrow Skip \ \Box \ b \rightarrow Skip$$

has the corresponding set $traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, \checkmark\rangle, \langle b\rangle, \langle b, \checkmark\rangle\}$. This means that, initially, P has not comunnicated any event ($\langle\rangle$), and then it can communicate either a or b and terminate. Analyzing a similar process $Q = a \rightarrow Skip \sqcap b \rightarrow Skip$, we would verify that Q is a refinement of P ($P \sqsubseteq_T Q$), since $traces(Q) \subseteq traces(P)$. This happens

because the traces model is not capable of differentiating an external from an internal choice, since they produce the same sequence of events.

The failures model is more elaborate in the sense that it is able to distinguish between internal and external choice, and can be used to check for deadlock. The intuition is that it identifies the sequence of events a process can communicate but also the refusals at each point of execution. Furthermore, the failures-divergences model includes the characteristics of both traces and failures, but also identifies livelocks. Considering the context of our work, however, traces is enough as a semantic model, since we are only concerned with sequences of events of a process that compose the test scenarios of interest.

3.2 SEMANTICS OF REQUIREMENTS

Because we aim to define a formal semantics for the proposed CNL requirements in CSP, in order to be able to reason about soundness and consistency we must adopt an appropriate mapping that reflects the dynamics of interacting with a SUT to observe its responses. In this light, we represent the requirements in terms of input and output events. A proper testing theory is discussed in detail in Chapter 4.

3.2.1 Actions to Input/Output Events

Test actions are the building blocks of test steps, domain models and requirements. Then, since all artifacts are just different kinds of compositions of these actions, we need to provide a mapping between the test action elements and CSP processes. Due to the testing theory we adopt, there must be a distinction between input and output events in a specification. Test actions alone, without context, have not a direct mapping: only when considering a requirement they can be accurately interpreted. The intuition for interpreting a requirement is simple: every action from a requirement circumstance (see Equation 2.2) is mapped to an input event, while the actions within a statement (Equation 2.3) are translated into outputs. The reasoning behind this is that we must interact with the system to fulfill the circumstance and, then, we must check whether the actions specified in the statement clause happened. There are, however, some exceptions to this rule that are discussed in the next sections.

3.2.2 Denotational semantics

Here we present the semantic rules that translate the specifications into CSP processes and events. It is worth mentioning that, for simplicity, we assume that all necessary channels and auxiliar sets are declared. The initial and more abstract artifact is the feature document, in which the requirements are listed. Each requirement, in turn, can be represented by one or more normalized sentences. The metalanguage expressions used to define the semantic rules are underlined and highlighted with a different color (gray).

We also use meta keywords to define auxiliary and local functions (*let...within*) and other variables (*where*). The symbol $\widehat{=}$ denotes a function definition.

Rule 1. Feature ⟦ feature: Feature ⟧ =

```
let list(current, remaining) ≙
  if #remaining = 0
    ⟦ current ⟧
  else
    ⟦ current ⟧ [+ α_current ∩ α_head remaining +] list(head remaining, tail remaining)
within
  REQ = list(head sentences, tail sentences)
where
  sentences = normalize(feature.sentences)
```

Rule 1 shows that a feature is composed by a sequence of sentences. Each sentence is then semantically interpreted as described in Rule 2. Finally, the final specification, called *REQ*, is defined by using the synchronizing external choice operator between all individual sentences. Each sentence usually represents a single requirement, but there are cases that it can represent more than one. This is why we have to normalize them by searching the sentence for disjunctions. For instance, the sentence "The message must be sent or the popup should be shown after the button is pressed" would be broken down and combined together into two separated normalized sentences: "The message must be sent after the button is pressed" and "The popup should be shown after the button is pressed". Then, each individual (now normalized) sentence can be semantically defined.

Rule 2. Normalized Sentence ⟦ sentence: Sentence ⟧ =

```
let
  convert(elements) ≙
    join(map(elements, λ elem: ⟦ elem ⟧), → ) within
  convert(sentence.circumstances) → convert(sentence.statements) → Skip
```

Rule 2 details the semantics of a sentence. The event sequence for these processes will be obtained by converting the circumstances and, then, the actions to events (all prefix for the *Skip* process). The *convert* function maps all elements (Circumstances or Statements) into their semantic representations and, then, joins them together using → as separator.

Rule 3. Circumstance ⟦ circumstance: Circumstance ⟧ =

```
convertAction(circumstance.task.action, Input)
```

Rule 3 and Rule 4 both access the corresponding (annotated) action and pass it as argument to the *convertAction* function defined in Rule 5. The only difference is that circumstance actions will be marked as inputs while statements will be converted to outputs.

Rule 4. Statement ⟦ stmt: Statement ⟧ =

```
convertAction(stmt.task.action, Output)
```

We already mentioned in Section 2.2 that the static part of the action frame is used as a unique identifier. In this light, every action is translated to an event using the rule defined next.

Rule 5. Action convertAction(action: Action, type: Input|Output): event =

```
let
  prefix(type: Input|Output) ≙
    if type is Input then i_ else o_
  getId(action: Action) ≙
    action.operationId ⌢ action.patientId ⌢ action.modifierIds
within
  prefix(type) ⌢ getId(action)
```

Rule 5 describes a semantic function that translates any given Action into a CSP event depending on the additional parameter "type" that determines whether the resulting event should be an input or an output. As discussed in the previous section, this partitioning is appropriate for the kind of GUI testing of our application domain. If an action should be an input, it will have the prefix $i\_$, otherwise it will be marked as an output with $o\_$. The *getId* is a function that gives a unique identifier for the action by concatenating its slot values.

3.2.3 Example

To illustrate the semantic rules presented in the previous section, we show some concrete examples of how natural language sentences are translated into CSP processes and events. First, consider the feature below:

A message must be sent after the option is enabled
When the option is enabled the main screen should be shown

This feature contains two requirements, each being a separate sentence. After parsing the sentences, we have the syntactic tree represented as the following structured data:

$$\begin{aligned} sentence_1 = \{&circumstances : [\{operation : enable, patient : option\}], \\ &statements : [\{operation : send, patient : message\}]\} \end{aligned}$$

$$\begin{aligned} sentence_2 = \{&circumstances : [\{operation : enable, patient : option\}], \\ &statements : [\{operation : show, patient : mainscreen\}]\} \end{aligned}$$

By applying Rule 1, assuming that $sentences = [sentence_1, sentence_2]$, we have the partial *REQ* process as below:

$$\text{REQ} = \underline{[\![sentence_1]\!]} \; [+ \;\; \underline{\alpha_{sentence_1} \cap \alpha_{sentence_2}} \;\; +] \; \underline{[\![sentence_2]\!]}$$

The semantics of each sentence is obtained by applying Rule 2, which calls the function *convert* from Rules 3 and 4, as below:

$$\begin{aligned} \text{REQ} = \underline{\mathsf{convert}(sentence_1.\mathsf{circumstances})} \rightarrow \underline{\mathsf{convert}(sentence_1.\mathsf{statements})} \rightarrow \mathit{Skip} \\ [+ \;\; \underline{\alpha_{sentence_1} \cap \alpha_{sentence_2}} \;\; +] \\ \underline{\mathsf{convert}(sentence_2.\mathsf{circumstances})} \rightarrow \underline{\mathsf{convert}(sentence_2.\mathsf{statements})} \rightarrow \mathit{Skip} \end{aligned}$$

Then, each sentence is translated into a separate CSP process. The last step is to resolve the intersection between the alphabet of events for both sentences, as illustrated next:

$$\begin{aligned} \text{REQ} = \; &\text{i\_enableoption} \rightarrow \text{o\_sendmessage} \rightarrow \mathit{Skip} \\ &[+ \;\; \underline{\alpha_{sentence_1} \cap \alpha_{sentence_2}} \;\; +] \\ &\text{i\_enableoption} \rightarrow \text{o\_showmainscreen} \rightarrow \mathit{Skip} \end{aligned}$$

Finally, *REQ* is defined as the resulting process of applying the synchronizing external choice operator over both processes, as shown below:

$$\begin{aligned} \text{REQ} = \; &\text{i\_enableoption} \rightarrow \text{o\_sendmessage} \rightarrow \mathit{Skip} \\ &[+ \; \{i\_enableoption\} \; +] \\ &\text{i\_enableoption} \rightarrow \text{o\_showmainscreen} \rightarrow \mathit{Skip} \end{aligned}$$

It is worth mentioning that disjunctions may be present in the sentences. As discussed in Rule 2, disjunctions must be identified, separated and then combined into different requirements but reusing the parts in common. The following sentence illustrates a case in which a disjunction is present.

When the message is sent then the main screen should be shown or a network error should be shown

Since this requirement includes a disjunction, we remove it by normalizing the sentence into two separate ones, as shown below:

When the message is sent then the main screen should be shown
When the message is sent then a network error should be shown

$$normalizedSentence_1 = i\_sendmessage \rightarrow o\_showmainscreen \rightarrow Skip$$
$$normalizedSentence_2 = i\_sendmessage \rightarrow o\_shownetworkerror \rightarrow Skip$$
$$REQ = normalizedSentence_1 [+\{i\_sendmessage\}+] normalizedSentence_2$$

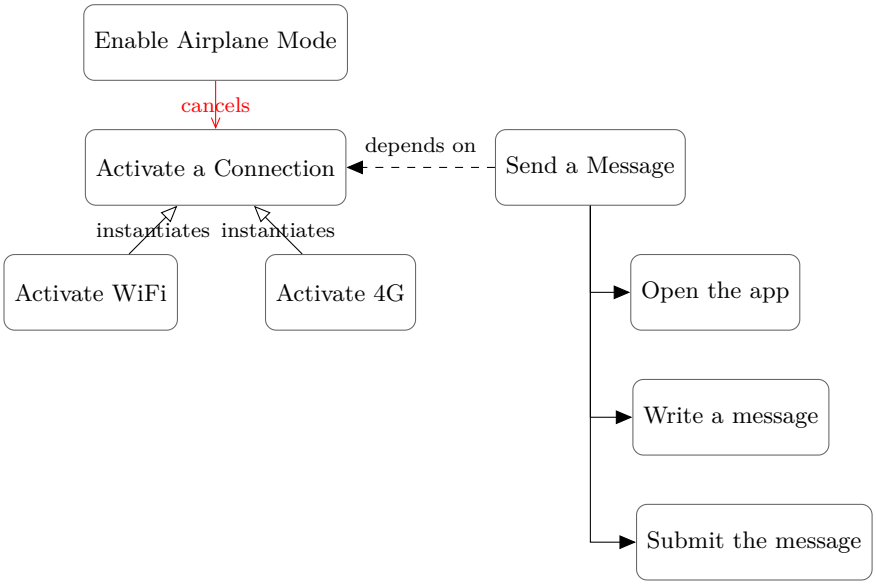
Because the disjunction is present only in the statement part, then the extra requirement has the same circumstances (hence *i_sendmessage* in both requirements). The final *REQ* process is described as a synchronized external choice between these two generated requirement processes. The resulting process accepts the event *i_sendmessage* and then presents a choice between the events *o_showmainscreen* and *o_shownetworkerror*, which is equivalent to $REQ = i\_sendmessage \rightarrow (o\_shownetworkerror \rightarrow Skip \,\Box\, o\_shownetworkerror \rightarrow Skip)$ using the traces semantics.

3.3 SEMANTICS OF DOMAIN MODELS

While requirements describe what should be implemented and tested, domain models give details on how. It is especially relevant for test engineers to give concrete implementation details when creating test cases so manual testers and the automation team can execute the tests without guessing undocumented behaviors. These concrete details can be expressed through associations between actions which together form a domain model. Figure 7 illustrates a sample domain model which describes associations related to “Sending a message”. The dependency between “Send a message” and “Activate a connection” means that one must make sure that the connection was activated before sending a message. On the other side, “Activating the Airplane Mode” cancels any action of “Activating a connection”. Thus, if airplane mode was activated after activating a connection, one must activate a connection again before trying to send a message. The instantiation relation represents concrete and alternative ways to execute an action. For instance, to “Activate a connection”, one can “Activate the WiFi” or “Activate the 4G”. Finally, “Open the

app", "Write a message", and "Submit the message" describe the ordered steps to "Send a message". It is worth mentioning that these actions, in turn, can also be further detailed.

Figure 7 – Domain model example



Source: The author (2022)

In spite of being an informal common practice, formalizing the domain model is new and allows an automated verification to check that additional details do not change the intended behavior described by the requirements. Taking into account that the requirements are already semantically defined in CSP, we also model the domain in terms of CSP events and processes. The obvious advantage is to analyze them both in a uniform way using a single language and tool. Next, we present the semantic rules for the domain model.

Rule 6 gives an overview of the main events and processes used to model the domain. We define the *DOMAIN* process that uses a replicated external choice operator over all main inputs, followed by a recursive call. *domain_main_inputs* includes only the main inputs, i.e., the first (source) input event of a relation. For instance, by envisioning the domain model as a graph, a dependency between two actions could be represented by the edge $i\_sendmessage \rightarrow i\_turnwifion$ ("send message" depends on "turning the wifi on"), in which the first node would be the main input.

Rule 6. Domain ⟦ domain: Domain ⟧ =

$DOMAIN = []main\_input : domain\_main\_inputs \bullet EXECUTE(main\_input);\ DOMAIN$
$EXECUTE(x) = (INJECT(x)[]NOTINJECT(x));\ CANCELS(x)$
$INJECT(x) = DEP(x);\ COMPOSITE(x);\ INJECTED(x)$
$COMPOSITE(x) = execute.x- > INSTANTIATE(x);\ x- > executed.x- > SKIP$
⟦ domain.dependencies ⟧

⟦ domain.cancellations ⟧
⟦ domain.details ⟧
⟦ domain.instantiations ⟧
⟦ domain.consistencyChoices ⟧

Then, we have the definition of *EXECUTE* which, in words, describes the steps of how to consistently execute an action by considering its details. Its definition begins by presenting a choice between two distinct parameterized processes *INJECT* and *NOTINJECT*. It introduces the possibility of modelling what happens when the execution is consistent and when it is not (Rule 11). The latter is specially important when testing for setup error scenarios. By diving on the *INJECT* definition, we identify a sequential composition of *DEP*, *COMPOSITE* and *INJECTED*. The processes *DEP* and *INJECTED* are placeholders for describing what should be executed before and after any given action. *DEP* is only defined for actions that have dependencies. In its definition, discussed in Rule 7, the actions that should be executed before x will be declared. *COMPOSITE* is a process for detailing on how to execute the input x. Its definition shows the auxiliar events *execute* and *executed*, to mark exactly before and after the action execution. Other processes can add events related to the execution of x by synchronizing with these auxiliar marks. If x is an abstract input, for instance, the concrete actions will be recursively listed as presented in Rule 9. It is worth mentioning that, between these auxiliar events, instead of just synchronizing with x we call the process *INSTANTIATE* (Rule 10), which is a placeholder for the cases when an abstract action can be executed in different but equivalent ways.

After the definition of these core processes, we have the characterization of all actual dependencies, cancellations, details, instantiations and consistency choices. Starting with the dependencies, Rule 7 shows that, for all dependencies present on the domain model, we define an "instance" of *DEP*, pattern matching with the given event *main*. The definition of *DEP* shows that there are two alternatives: the setup is already executed or it should be executed. We rely on the FDR analysis mechanism to give the shortest trace, thus not including re-execution of dependencies when it is not necessary. The *VERIFY_DEP* makes clear that if the dependency was already executed (via *EXECUTE*) then it will synchronize with the auxiliar event *isexecuted*. Finally, *DEP*(_) is defined for all the other cases when an action does not have dependencies.

Rule 7. Dependencies ⟦ dependencies: DependencyList ⟧ =

```
for main, deps @ dependencies
  DEP(main.id) = for dep in deps
    (VERIFY_DEP(dep.id) [] EXECUTE(dep.id));
DEP(_) = SKIP
```

```
VERIFY_DEP(x) = isexecuted.x -> SKIP
```

Cancellations on the other hand, as defined in Rule 8, describe which events should be re-executed when the given *main* is executed. This is specially important for incompatible actions. For instance, when an account is logged out, then we can not send any message until we log in again. Thus, *i_logout* cancels the previous action *i_login*. In order to actually instruct the consistency analyzer to re-execute cancelled actions, the parameterized event *cancels* synchronizes with another process (detailed in Chapter 4) that keeps record of the active actions.

Rule 8. Cancellations ⟦ cancellations: CancellationList ⟧ =

```
for main, canceled_events in cancellations
  CANCELS(main.id) = for cancel_evt in canceled_events
    cancels.cancel_evt.id ->
  SKIP
CANCELS(_) = SKIP
```

Expressing the concrete steps to execute an action can be done by defining its details, as in Rule 9. The intuition is to convey an abstract action by the sequential composition of its steps which, in turn, are also actions (or inputs) that can be further detailed too. The inspiration for this interpretation is the composite design pattern. The *DETAILS* process is instantiated for each composition described in the domain model and there is a base case for the actions that are already concrete.

Rule 9. Details ⟦ details: DetailsList ⟧ =

```
for main, composition in details
  DETAILS(main.id) = for detail_evt in composition
    EXECUTE(detail_evt.id);
DETAILS(_) = SKIP
```

Instantiations are similar to details, but they define an action in terms of choices instead of a sequence of steps. It is important in particular for defining equivalent ways of executing a procedure. For instance, when asked to activate an internet connection, you may turn on the wifi or activate the cellular data. In the case of the cellular data being

already active because of previous steps, we can procede to the next action. However, if we have depended directly on turning the wifi on, instead of depending on the more abstract action of activating the internet connection, we would have turned it on unnecessarily. The Rule 10 shows the process *INSTANTIATE* that is defined by means of an replicated external choice between the equivalent events. In the case of an action not having different instances, it defaults to the *DETAILS* process defined above.

Rule 10. Instantiations ⟦ instantiations: InstantiationList ⟧ =

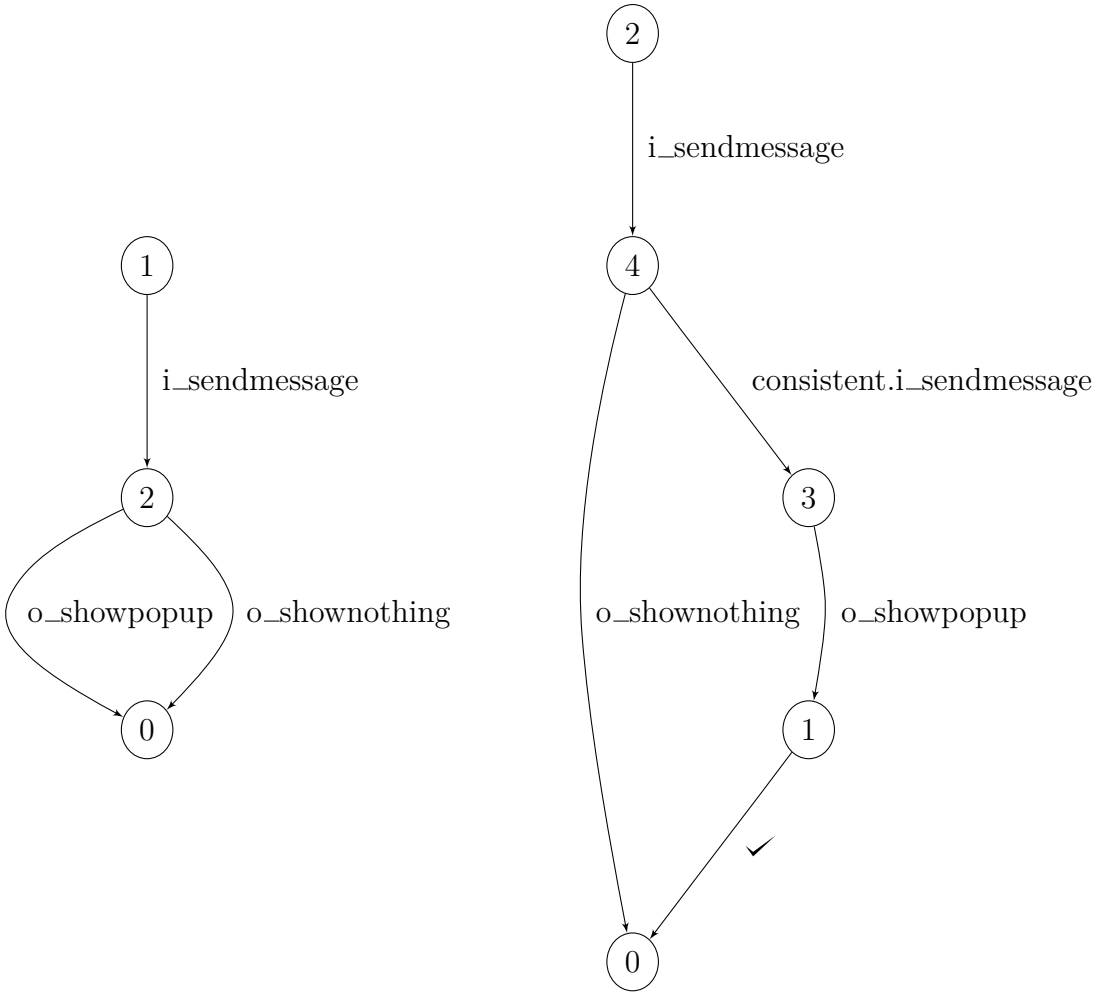
---

for main, instances in instantiations
  $INSTANTIATE$(main.id) = [] instance: instances @ EXECUTE(instance.id)
$INSTANTIATE(x) = DETAILS(x)$

Lastly, we can model consistency choices. These alternatives allow us to model not only the success scenarios (in which all setups were correctly executed) but also what should happen when there is a consistency problem. Additionally, this mechanism can allow us to minimize the number of inconclusive results by modelling explicit prefixes on previously nondeterministic processes. Figure 8 illustrates an specification with nondeterministic traces (a) and the same specification with the consistency marker to make it clear that a popup should be shown only when the execution of sending a message is consistent.

Rule 11 first shows the declaration of the *consistent* channel that receives as parameter all events with consistency choices. The auxiliar set *consistency_success_events* only holds all possible events from this channel for later analysis. Then, for each consistency choice we define the *NOTINJECT* and *INJECT* processes. The former does nothing but synchronizes with the plain execution of the given *main*. The latter "overrides" the *INJECTED* process to add the auxiliar event *consistent*!*main* to mark a consistent execution (since the *INJECTED* process will be executed only when the *INJECT* process is chosen). Additionally, we define the indexed process *CHOICE* that identifies which event should be executed when the consistency is honored. Finally, the *CHOICES* process list all indexed choices with an external choice operator. It will later be put in parallel with the requirement process to supplement the additional info.

Figure 8 – Consistency choice



Source: The author (2022)

Rule 11. Consistency Choices ⟦ consistencyChoices: ConsistencyChoiceList ⟧ =

channel consistent: {list(getMainList(consistencyChoices))}
consistency_success_events = |consistent|

for main, choice in consistencyChoices
  $NOTINJECT$(main.id) = SYNC(main.id)
$NOTINJECT(x) = INJECT(x)$
$SYNC(x) = execute.x -> x -> executed.x -> SKIP$
main, choice @ consistencyChoices
  $INJECTED$(main.id) = $consistent$!main.id $->$ $SKIP$
  $CHOICE$index = main.id -> $consistent$!main.id -> choice.id -> SKIP
$CHOICES$ = for main, choice in consistencyChoices $CHOICE$index []

3.3.1 Example

To illustrate the semantic rules for domain models presented above, we show a concrete example of how a structured text can be parsed into a domain model. Listing 3.1 shows the same domain model illustrated in Figure 7 but described via structured text divided into sections (one section for each association type), in which every sentence complies with our CNL.

Listing 3.1 – Domain model example in structured text

```
# Details
  * Send a Message
    * Open email app
    * Write a new message
    * Submit the message
# Dependencies
  * Send a Message
    * Activate a Connection
# Cancellations
  * Enable Airplane Mode
    * Activate a Connection
# Instantiations
  * Activate a Connection
    * Activate WiFi
    * Activate 4G
```

Then, each sentence is parsed to obtain the syntactic tree and the corresponding frame slots. Using these slots, we infer an identifier used later to generate CSP events. For simplicity, we illustrate below how the previous domain model can be parsed in terms of their inferred identifiers.

$$
\begin{aligned}
details = \{&i\_sendmessage : [\{id : i\_openapp\},\\
&\{id : i\_writemessage\}, \{id : i\_submitemessage\}]\}\\
dependencies = [\{&source : \{id : i\_sendmessage\},\\
&target : \{id : i\_activateconnection\}\}]\\
cancellations = [\{&source : \{id : i\_airplanemode\},\\
&target : \{id : i\_activateconnection\}\}]\\
instantiations = \{&i\_activateconnection :\\
&[\{id : i\_activatewifi\}, \{id : i\_activatefourg\}]\}
\end{aligned}
$$

By applying Rules 7-10, we get the partial CSP model as follows:

$$DEP(i\_sendmessage) = (VERIFY\_DEP(i\_activatedata)[]EXECUTE(i\_activatedata))$$

$$DEP(\_) = Skip$$

$$CANCELS(i\_activateairplanemode) = cancels.i\_activateconnection \rightarrow Skip$$

$$CANCELS(\_) = Skip$$

$$DETAILS(i\_sendmessage) = EXECUTE(i\_openapp);\ EXECUTE(i\_writemessage);\ EXECUTE(i\_submitmessage)$$

$$DETAILS(\_) = Skip$$

$$INSTANTIATE(i\_activateconnection) = EXECUTE(i\_activatewifi)\ []EXECUTE(i\_activatefourg)$$

$$INSTANTIATE(x) = DETAILS(x)$$

Since the domain model and requirements are defined in terms of CSP processes, we can now leverage generation mechanisms built upon refinement checkers to generate sound and consistent tests. A fitting generation mechanism is discussed in the next chapter.

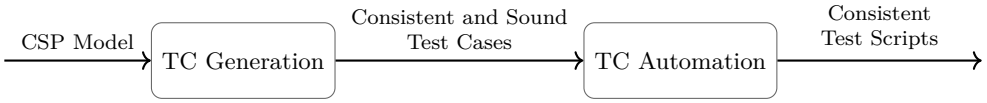
4 SOUND AND CONSISTENT TC GENERATION AND AUTOMATION

When generating test cases and their corresponding scripts, one must ensure that both artifacts actually produce reliable results regarding the system conformance to the requirements. Because test completeness is not feasible, we can not assume that a System Under Test (SUT) is correct only because some test cases passed. However, if we guarantee that whenever a test fails, then the SUT is not conformant to the requirements, we say that this particular test is sound. In other words, there can be false negatives, but we can prevent false positives by creating sound test cases. The approach to generate sound test cases is to define a formal conformance relation between implementation and specification, from which we derive valid test cases.

Furthermore, the level of abstraction present in these generated test cases is directly related to the one from the specification. Because useful specifications rarely have implementation details, derived test cases are abstract and cannot be directly executed on an implementation. In this light, to allow the generation of executable scripts, we adopt domain models as means to add execution details without changing the core behavior from the specification. Domain models, combined with requirements, allow the generation of consistent test cases. Consistent test cases can be executed as-is by a test driver because their dependencies and details are in accordance with the corresponding domain model.

The task of generating test cases from a Communicating sequential processes (CSP) model, as presentend in the previous chapter, is illustrated in Figure 9. Because the CSP Model also includes a domain model, a consistency analysis can be performed, ensuring that the generated test cases are consistent. By using an already established test generation strategy, we map CSP traces into legible test cases for automation. Also, a formal notion of conformance, based on traces refinement, ensures sound test cases.

Figure 9 – Process Task: TC Generation



Source: The author (2022)

This chapter is structured as follows. Section 4.1 presents some background on the approach we adopt for sound test case generation, starting with Section 4.1.1, on testing theory and conformance verifications regarding model based testing. After discussing the *ioco* and *CSP Input-Output Conformance (cspio)* relations, Section 4.1.2 introduces the Abstract Test Generator (ATG) and its strategy to generate sound test cases. In Section 4.1.3 we show how to combine the requirements with the domain model to ensure consistent test cases. Section 4.1.4 presents the refinement assertions to find traces that

will be converted into test cases. Finally, Section 4.2 shows how to automate the output test cases: mapping consistent test cases into scripts by annotating atomic actions in code with Controlled Natural Language (CNL)-compliant descriptions.

4.1 TEST GENERATION

In this section, we cover the steps involved in generating test cases. First, we present the strategy from Nogueira, Sampaio and Mota (2014) that allows one to create sound test cases from counterexamples of refinement verifications considering the *cspio* conformance relation. Then, we discuss how to check for consistency by considering the requirements model combined with the domain model, both in CSP. Finally, after showing how to extract sound and consistent test cases from the requirements and domain model, we discuss how to linearize (translate) these test cases back to English. They can then be subjected to manual execution or automatically translated into scripts to automated execution, as presented in Line 4.2.1.

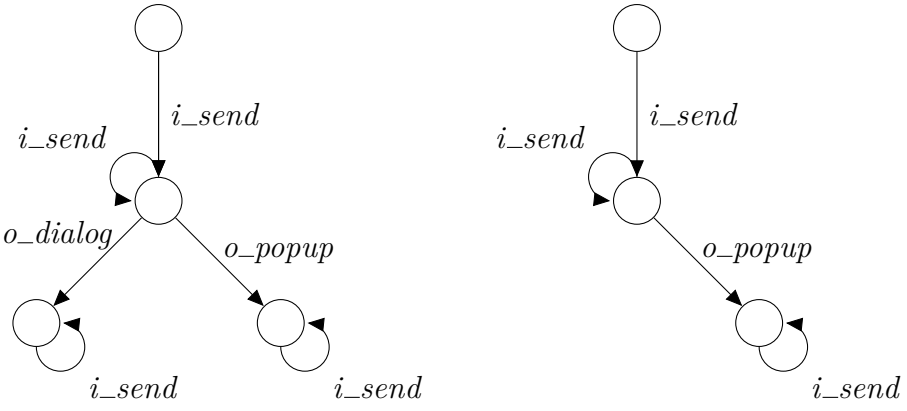
4.1.1 Testing Theory and Conformance Relations

Since we aim to generate sound test cases, we adopted a model-based strategy to automate the process of test case generation from requirements models. Concerning a formal account of MBT, conformance testing has the objective of checking whether an Implementation Under Test (IUT) satisfies its specification according to some defined relation (conformance relation). An inherent assumption is that the IUT can be modeled in a known formalism (testing hypothesis). There are several conformance relations (TRETMANS, 1996) mainly based on formal notations like Finite State Machines (Finite-State Machine (FSM)) and Labelled Transition Systems (Labelled Transition System (LTS)), but also on more abstract notations like the process algebra CSP (SAMPAIO; NOGUEIRA; MOTA, 2009).

Based on a conformance relation, test cases can be automatically generated from the input test model using algorithms which ensure that the generated tests satisfy relevant properties, such as *soundness*. Informally, soundness means that if an IUT fails a test case, then the IUT does not conform to the model from which the test was generated.

The *ioco* (Input-Output Conformance) (TRETMANS, 1996) is one of the most widely used conformance relations. It is based on Input-Output LTS (IOLTS) that in turn are a special class of LTS. IOLTS events, unlike LTS, are partitioned into input and output events. Taking into account the hypothesis that an implementation can be modelled using an IOLTS, another constraint must hold: all inputs must be enabled for all states in the implementation. This property is known as input completeness. Since the test case interacts with the implementation, the latter should be able to handle any input in order to always reach a verdict, avoiding unwanted deadlock.

Figure 10 – LTS - ioco



(a) iolts1 (b) iolts2

Source: The author (2022)

Figure 10 shows two IOLTS, *iolts1* and *iolts2*, both modelling different device behaviors. Because the events are partitioned into inputs and outputs, we adopt $i\_$ for representing inputs and $o\_$ for outputs. Considering that the topmost node in both graphs is the start node, we can interpret that in *iolts1*, after performing the input *i_send*, we should observe a *o_dialog* or a *o_popup*. While in *iolts2*, after the same input, we expect only the output *o_popup*. It is worth mentioning that the input is always enabled in all states, illustrated by self-pointing arrows in all nodes (*i_send*). In order to reason whether they conform to each other, we must formally describe the relation:

Definition 4.1.1 (ioco)**.** $i \; ioco \; s \mathrel{\widehat{=}} \forall \, \sigma \in straces(s) \bullet out(\Delta_i, \sigma) \subseteq out(\Delta_s, \sigma)$

where

$out(X, \sigma) = \{e : O_\delta \mid \sigma \frown \langle e \rangle \in straces(X)\}$

Definition 4.1.1 states that an implementation i conforms to a given specification Δ_s when the set of observed outputs after "performing" any suspension trace (σ) in i is a subset of the observed outputs after the same trace (σ) in Δ_s. Suspension traces, in addition to the original concept of traces, include the observation of quiescence (CAVALCANTI et al., 2016). The symbol Δ_x, where x is any LTS, represents x behavior after adding δ in every state that manifests quiescence. Likewise, the symbol δ annotated on the set O indicates that it includes quiescence. Quiescence represents the lack of observable behavior, as in deadlocks, livelocks, and output locks. Considering this definition, we can establish that *iolts2* ioco *iolts1* but the opposite is not true, since *o_dialog* is not an output observable in *iolts2* after the trace $\langle i\_send \rangle$: $out(iolts1, \langle i\_send \rangle) \nsubseteq out(iolts2, \langle i\_send \rangle)$

To allow an automated conformance verification and further test case generation via model checking, without relying on ad-hoc algorithms, we will use a process algebraic characterization of the ioco relation in CSP called *cspio*.

4.1.1.1 *cspio* Conformance Relation

In this section we present a formal definition in CSP of a conformance relation *cspio*, which is based on *ioco*. This new relation also assumes that the events are partitioned into inputs and outputs and that the IUT can be modelled as a CSP process. Similar to *ioco*, any given IUT conforms to a specification S if, after performing the same available traces, the outputs from IUT are a subset of the S outputs. To automate the verification of this conformance relation, it is encoded as the following CSP refinement check:

Definition 4.1.2 (cspio verification)**.** $S \sqsubseteq_T (S \triangle ANY(\Sigma_{Io}, Stop)) \,[|\, \Sigma_{IUT} \,|]\, IUT$

The intuition of the right-hand side of the refinement check (Definition 4.1.2) is to offer all possible specification traces for the implementation to synchronize and verify the outputs. The direct comparison between S and IUT would not work, since IUT can have, by definition, traces not present in S, because it can accept extra inputs or offer additional outputs after a trace that does not belong to S (CAVALCANTI et al., 2016). Then, $(S \triangle ANY(\Sigma_{Io}, Stop)) \,[|\, \Sigma_{IUT} \,|]\, IUT$ masks IUT traces that should not be verified by *cspio*. It blocks inputs not accepted by S, and the interruption of S on $ANY(\Sigma_{Io}, Stop)$ allows synchronization of output events from IUT that S does not produce, terminating the process. If this interruption happens, then the refinement would be false, as expected. If the resulting process does not produce different outputs from the same inputs, then the refinement is valid and IUT *cspio* S.

4.1.2 Abstract Test Generator

There is a well-established approach, implemented by the tool called Abstract Test Generator (ATG), which provides a guided test generation based on the *cspio* conformance relation (NOGUEIRA; SAMPAIO; MOTA, 2014) and CSP traces semantics. The general idea is to exercise the specification, obtain relevant scenarios as counterexamples of refinement verification, and generate sound test cases from them. In the next subsections, we briefly present the core elements of this strategy: test scenario, test purpose and test case.

4.1.2.1 Test Scenario

From any specification S we can exercise its paths to obtain traces that satisfy a given property, which can be, for instance, a successful termination. These properties can even depict selection criteria, which can guide the generation to search only for relevant scenarios. These criteria can be described as test purposes (Section 4.1.2.2). Regardless of the criteria, the scenarios are generated from counterexamples of refinement assertions. To demonstrate the approach, consider $MARK = \{accept.n\}$ with $n \in \mathbb{N}$ the set of all possible mark events that can be used to indicate a target that must be reached when searching for scenarios. The idea is to create a new process S' by adding these mark events

on the original model S. Then, by using the mechanism of refinement checking, we can obtain the scenarios from the counterexamples of $S \sqsubseteq_T S'$, which does not hold since there should be no trace $ts \in traces(S)$ and marker $m \in MARK$ in which $ts \frown \langle m \rangle \in traces(S)$. Then, all counterexamples should be in the format $ts \frown \langle m \rangle$, with ts being the test scenario we want by placing the marker m.

For instance, considering the following specification:

$$S = i\_turnon \rightarrow o\_turnon \rightarrow i\_send \rightarrow (o\_message \rightarrow Skip \,\Box\, o\_popup \rightarrow Skip)$$

We can add a marker $accept.1$ after o_popup if we are interested in testing only when a popup is shown. Then, we have the following annotated specification:

$$S' = i\_turnon \rightarrow o\_turnon \rightarrow i\_send \rightarrow$$
$$(o\_message \rightarrow Skip \,\Box\, o\_popup \rightarrow accept.1 \rightarrow Skip)$$

By checking the refinement $S \sqsubseteq_T S$' we get as counterexample the trace:

$$\langle i\_turnon, o\_turnon, i\_send, o\_popup, accept.1 \rangle$$

This is a valid path for testing. More generally, the generation is guided by the concept of test purpose, which provides auxiliary functions to mechanize the task of adding the mark events. Nevertheless, it is worth mentioning that scenarios are not test cases. In this example, even though we want to test only when a popup is shown (o_popup), if we eventually observe the output $o\_message$ when executing the test on the IUT, it should not be considered a failure. To generate a test case, one needs to consider the verdict, which, in the example considered above, is inconclusive, since $o\_message$ is an allowed output of the specification in this context. This will be further discussed in Section 4.1.2.3.

4.1.2.2 Test Purpose

Even though successful termination is a relevant criteria when searching for scenarios, as discussed before, there must be a more robust way to indicate more complex selection criteria. In this light, Nogueira, Sampaio and Mota (2014) adopt the concept of test purpose, which is a partial specification that describes the desired aspects for the generated tests. Because it is also defined as a CSP process, it actually denotes the traces that the scenarios must include. Definition 4.1.3 formalizes the concept:

Definition 4.1.3 (Test Purpose)**.** *With TP standing for test purpose and S for specification, both being CSP processes:*

$$\forall\, t \in traces(TP) \bullet (t \in RUN(\alpha_S)) \vee (t = t' \frown \langle m \rangle \wedge m \in \alpha_{MARK} \wedge t' \in RUN(\alpha_S))$$

A test purpose defines a trace that is present on the specification, besides allowing extra mark events.

There are some auxiliary processes that can help build test purposes. These processes have the prefix ATG, which indicates that they are part of the corresponding module. The process $ATG{::}ANY(evtset, next)$ offers any event present in the set $evset$ and behaves as $next$ when it synchronizes. The definition is $ATG{::}ANY(evset, next) = \Box\ evt : evtset \bullet evt \rightarrow next$. On the other hand, the process $ATG{::}UNTIL(\alpha, evtset, next)$ synchronizes with any event from α until an event from $evtset$ is communicated, which will make the process behave as $next$. It is defined as follows: $ATG{::}UNTIL(\alpha, evtset, next) = RUN(\alpha_S - evtset) \triangle ATG{::}ANY(evtset, next)$. Finally, $ATG{::}ACCEPT(\{n\}) = accept.n \rightarrow Stop$ simply adds a marker with the given natural number n. To illustrate their use, consider the following test purpose:

$$TP = UNTIL(\alpha_S, \{o\_popup\}, ACCEPT(1))$$

When analyzing $S\,|[\,\Sigma_{Si} \cup \Sigma_{So}\,]|\,TP$, with S from Section 4.1.2.1, we get an identical process to S', which provides the same scenario as counterexample for the refinement verification: $\langle i\_turnon, o\_turnon, i\_send, o\_popup, accept.1\rangle$.

4.1.2.3 Test Case

A test case is a process that interacts with the implementation and communicates an event representing the verdict for whether the implementation conforms to the specification (regarding the *cspio* conformance relation). In CSP terms, this interaction/execution is characterized by the parallel composition $EXEC = IUT\,|[\,\Sigma_{I_{IUT}} \cup \Sigma_{O_{IUT}}\,]|\,TC$, being IUT the process that represents the implementation while TC the process that describes the test case.

TC has in its alphabet a new event type $v \in VER \bullet VER = \{pass, fail, inco\}$, which represents that a test passed, failed or was inconclusive, respectively. It is worth mentioning that, by definition, test outputs in fact stimulates the implementation, while the response from the implementation is input for the test case. In this sense, the alphabet of TC is dual to the one from IUT, in the sense that TC outputs are inputs to IUT and test outputs are inputs to the implementation.

To check the result of a test execution over an implementation, we only need to verify which verdict is present. This check can be done with the refinement $EXEC \setminus (\Sigma_{I_{IUT}} \cup \Sigma_{O_{IUT}}) \sqsubseteq_T v \rightarrow Stop$. In other words, by executing the test case and hiding the inputs and outputs, the only events left are verdicts. Thus, when the refinement holds, then $t \frown \langle v\rangle \in traces(EXEC)$, which means that v is a possible verdict.

An important property that should be pursued when generating test cases is that these Test Case (TC)s should not generate false fails. This property, known as soundness, guarantees that when a test execution reaches a fail verdict then the implementation, for sure, does not conform to the specification. The Definition 4.1.4 formally defines a sound test case in CSP:

Definition 4.1.4 (Soundness). $\langle fail \rangle \in traces(EXEC \setminus (\Sigma_{I_{IUT}} \cup \Sigma_{O_{IUT}})) \Rightarrow \neg(IUT\ cspio\ S)$

To actually build a sound test case from a test scenario, we have to make sure that it records all output events that the specification communicates at each step of the given scenario. The process of building the sound test case is iterative and begins with a default annotated trace *atrace* with the format $\langle (ev_i, out_i) \frown (accept.n, \{\}) \rangle \bullet 1 <= i <= \#ts$ where ev_i is the i^{th} element of *ts* and out_i the corresponding set of output events after performing the trace until ev_{i-1}. The actual function $ATG::TC\_BUILDER$ is defined in Listing 4.1.

Listing 4.1 – ATG Test case builder

```
channel pass, fail, inconclusive -- not in Alfa
FAIL = fail → Stop
PASS = pass → Stop
INCO = inconclusive → Stop

TC_BUILDER(ialfa,oalfa,<(accept.i,{})>) = PASS
TC_BUILDER(ialfa,oalfa,s) = SUBTC(ialfa,oalfa,head(s));TC_BUILDER(ialfa,oalfa,
    tail(s))
SUBTC (ia,oa,(ev,outs)) =
  if(member(ev,ia)) then
    ev → Skip
  else
    ev → Skip
    []
    ANY(diff(outs,{ev}), INCO)
    []
    NOT(oa,union(outs,{ev}), FAIL)
```

Listing 4.1 first declares the verdict events and auxiliary processes *PASS*, *FAIL* and *INCO*. *TC_BUILDER* has a base case for when the *atrace*'s last element is reached, which yields the *pass* verdict. Until then, it recursively calls *SUBTC* for each element. Then, if the current event is an input, it just communicates the event – and it is guaranteed to synchronize, since the implementation is input enabled. On the other hand, if the event is an output, it offers three choices: a) communicates the event, which will synchronize if the *IUT* yields the expected output; b) communicates any event from the set *outs*, except the current one, and mark as inconclusive; and c) for the case when none of the previous events synchronize, it communicates all other possible outputs and marks as *FAIL* if any synchronizes. After building the *TC* process with *TC_BUILDER* and evaluating *EXEC* (assuming the trivial implementation being *S* itself) following Definition 4.1.4, if we get a counterexample, then we add the event to the corresponding out_i set and rebuild the test case until it is sound.

4.1.3 Consistency Analysis

The domain is modeled with CSP events and processes as described in Section 3.3. However, the actual processes that carry out the consistency analysis are discussed next. The

general idea is to combine the original requirements in parallel with the domain to build a more detailed specification, which is then used to generate test cases that are consistent in that there are no missing steps necessary for their executions by a test driver.

For a test case to be consistent, all rules expressed in the associated domain model must be satisfied. As presented in the previous chapter, these rules can encompass dependencies, compositions, and other relations. For instance, consider the following requirement:

$$REQ = i\_activatedata \rightarrow i\_sendmessage \rightarrow o\_showpopup \rightarrow Skip$$

It defines a new requirement describing that a popup should be shown when a message is sent with the data activated beforehand. For the purpose of this demonstration, assume that *i_activatedata* and *o_showpopup* are atomic actions interpreted by a test driver. To make the tests (derived from this specification) executable, then one can define the following domain rules for *i_sendmessage*:

$$DETAILS(i\_sendmessage) = i\_openemail \rightarrow i\_writemessage \rightarrow i\_submitmessage \rightarrow Skip$$
$$DEP(i\_submitmessage) = i\_activatedata$$

The first rule defines that *i_sendmessage* is composed of, in this case, three atomic actions. The composition resolution is performed by the process $EXECUTE(i\_sendmessage)$, which is defined for any action present in the domain model (Section 3.3). The second rule establishes that each time *i_submitmessage* is executed, *i_activatedata* must be also executed before and it is still active. To keep track of which actions are active at each step, we define the process *EXECUTION_HISTORY*. The intuition is that other processes can try to synchronize with it to check whether an input is active at the moment. For instance, when evaluating the dependencies of *i_submitmessage* in this case, it would detect that *i_activatedata* was already executed and, thus, there is no need to execute it again.

The *EXECUTION_HISTORY* process described in Listing 4.2 applies the replicated interleave operator that parametrises the process *LOOP* with all events that occur in the domain model. The *LOOP* process is responsible to keep track of which actions were already executed. When an input event is performed on the *REQ* process or by the *DOMAIN* itself, then it synchronizes with the markers *execute* and *executed*. When it occurs, it allows other processes to also synchronize with the marker *isexecuted* (*LOOPEXECUTED*), in order to avoid re-executing an action unecessarily. Finally, *LOOPCANCEL* allows processes to synchronize with *cancels* event if the given action was cancelled. In this case, it recurses back to the process *LOOP* which enforces processes to execute the action again, if needed, since the event *isexecuted* is not available anymore.

Listing 4.2 – Domain model - execution tracking

```
EXECUTION_HISTORY = ||| in : domain_events @ LOOP(in)
LOOP(x) = execute.x -> executed.x -> LOOPEXECUTED(x) [] LOOPCANCEL(x)
LOOPEXECUTED(x) = isexecuted.x -> LOOPEXECUTED(x) [] LOOP(x) [] LOOPCANCEL(x)
LOOPCANCEL(x) = cancels.x -> LOOP(x)
```

Listing 4.3 presents the augmented specification that is used to generate concrete test cases: the resulting process of putting the requirements, domain model and execution history together. The first process, *CONSISTENCY_ANALYZER*, is the result of combining the *DOMAIN* from Section 3.3 with the *EXECUTION_HISTORY* process discussed above. With this process it is possible to reason about consistency since we combine the effects of the rules from the domain model with the expected execution progress. *REQ_MARKED*, in turn, is another intermediate process that adds information about consistency choices directly on the original requirements. Finally, *SPEC* is the process used later as the specification for test case generation. Instead of capturing only the original and abstract requirements, it now holds more detailed events to generate concrete and consistent test cases.

Listing 4.3 – Consistent specification

```
CONSISTENCY_ANALYZER = DOMAIN [|aux_domain_events|] EXECUTION_HISTORY
REQ_MARKED = REQ [+ inter(req_events, choice_events) +] CHOICES
SPEC = (REQ_MARKED [|union(inter(req_events, domain_events),
    consistency_success_events)|] CONSISTENCY_ANALYZER) \ all_aux_events
```

Because we actually add behavior to the initial requirements, we must ensure, however, that the original behavior is preserved. It is important to have a guarantee that any information about the domain does not interfere with the core behavior. While it could be valuable for other scenarios, it is important in our industrial context to ensure that any information added by test analysts (or any other stakeholders) does not tamper with, for instance, third party requirements or critical functionalities. Listing 4.4 requires not only that the new specification be a refinement of the original requirement, but also that they are in fact equivalent. The refinement, in both directions, is checked after hiding the extra and auxiliary events from both processes.

Listing 4.4 – Check for behavior changes

```
SPEC_SIMPLE = SPEC \ domain_only_events \ details_events \ instantiations_events \
    consistency_success_events
REQ_SIMPLE =  REQ \ all_aux_events \ consistency_success_events
assert REQ_SIMPLE [T= SPEC_SIMPLE
assert SPEC_SIMPLE [T= REQ_SIMPLE
```

The *cspio* conformance relation assumes that an implementation always accepts any input (from its alphabet) and an output can always be observed after the input. Because of this input and output-enabledness constraint, we must guarantee that the process that models the system requirements meets this constraint. The *MEALY* process from Listing 4.5 when put in parallel with the specification process, enables the alternation of inputs and outputs.

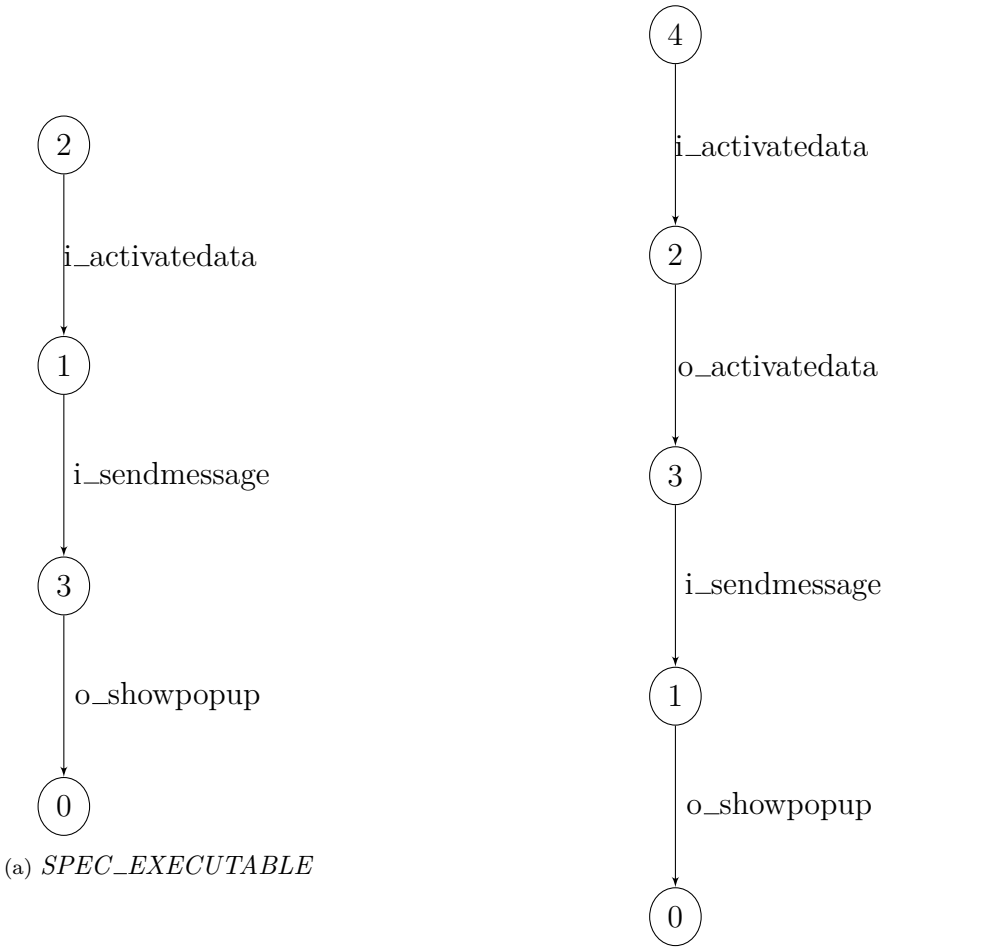
Listing 4.5 – Forcing alternation between inputs and outputs

```
ANY_IO = [] x: union(req_events, domain_events) @ x -> (OUT(x) [] ATG::ANY(
    req_outputs, SKIP))
MEALY = ANY_IO; MEALY

for input in allInputs
    OUT(input) = getOutputFrom(input) -> SKIP

OUT(_) = SKIP
```

Figure 11 – Alternating input and output events



(a) *SPEC_EXECUTABLE*

(b) *SPEC_EXECUTABLE_MEALY*

Source: The author (2022)

The *MEALY* process is defined as a sequential composition, beginning with the process *ANY_IO*, which in turn offers any event for synchronization and then communicates a corresponding trivial output for any input x having an associated $OUT(x)$ definition. $OUT(x)$ is dynamically generated for any input, providing an alternation between inputs and outputs before deriving tests. The intuition is that, when one makes an action, say "Send a message", the trivial output is that the "Message was sent". So, there should be an *OUT* process for each input of the specification or the domain model. In general terms, only the prefix $i\_$ is replaced by $o\_$. The *MEALY* process can be recursively called

until an output from the original requirements is reached. When reached, the process may stop, as stated by $ATG::ANY(req\_outputs, STOP)$.

The result of the parallel composition of *MEALY* with a process that captures a fragment of a requirements model is illustrated in Figure 11. *SPEC_EXECUTABLE* from Figure 11a has two consecutive inputs, which is not a proper specification. To ensure that there is an alternation between inputs and outputs, we add trivial responses for every input after its execution, by applying a parallel composition between processes *SPEC_EXECUTABLE* and *MEALY*, resulting in *SPEC_EXECUTABLE_MEALY*, as shown in Figure 11b.

These processes are also exemplified in Listing 4.6. In Line 1, *SPEC_EXECUTABLE* is created by hiding the events that were, in any way, modified by the domain model. For instance, if the input *i_sendmessage* is detailed by the sequence $\langle i\_write, i\_submit \rangle$, then the abstract action *i_sendmessage* should be ignored and must not appear in the traces. Line 2 defines the process *SPEC_EXECUTABLE_MEALY* yielded by the parallel composition of the previous *SPEC_EXECUTABLE* with the *MEALY* process. To avoid an unnecessary output *o_sendmessage* to be added when there is already an expected output (*o_showpopup*) from the requirements, we use the prioritise operator defined as $prioritise(P, \langle S_1, ..., S_n \rangle)$. The resulting process behaves similarly to P but prevents any event present in $S_i \bullet 1 < i <= n$ from communicating when any event from $S_j \bullet j < i$ is possible. Thus, when the parallel composition is passed as a parameter to this *prioritise* operator, the original outputs from the requirements are chosen over the extra outputs, thus avoiding unnecessary auxiliary outputs.

Listing 4.6 – MEALY

```
1 SPEC_EXECUTABLE = SPEC \ detailed_events \ instantiated_events \ all_aux_events
  SPEC_EXECUTABLE_MEALY =
3     prioritise(SPEC_EXECUTABLE [|union(req_events, domain_events)|] MEALY,
                 <req_outputs, diff(all_outputs, req_outputs)>)
```

4.1.4 Generation Mechanism

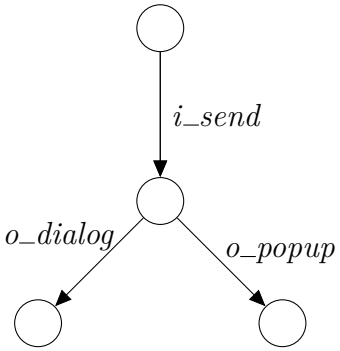
The generation of test cases follows the rules described in Section 4.1.2.3 with some modifications on the elements but not in the strategy itself. Listing 4.7 describes the steps of finding test scenarios. First, at Line 1, we create the test purpose *TP*1 which matches any event until *actionOutput* is reached. In our context, the generation is performed for each action of all requirements. Then, a test purpose is defined for each of these actions (which are converted to output as in Rule 4), supplied as the parameter *actionOutput*. Finally, we perform the refinement check in Line 2 to find a scenario from *SPEC_EXECUTABLE_MEALY* that fulfills *TP*1.

Listing 4.7 – Finding scenarios

```
  TP1 = ATG::UNTIL(req_events, {actionOutput}, ATG::ACCEPT({1}))
2 assert SPEC_EXECUTABLE_MEALY [T= SPEC_EXECUTABLE_MEALY [|req_events|] TP1
```

A test scenario is built from the counterexample found by the assertion in Line 2. The scenario is then annotated to include all possible outputs for each step. This annotation is important to mark inconclusive results, i.e., outputs that are not relevant for the current scenario, but are possible outcomes defined in the specification (SAMPAIO et al., 2014). For instance, consider a specification illustrated in Figure 12 and a given scenario $sce_1 = \langle (i\_send, \emptyset), (o\_popup, \emptyset) \rangle$.

Figure 12 – Sample IOLTS specification



Source: The author (2022)

An annotated trace $atrace_1$ must cover the possibility in which an IUT outputs *o_dialog* during execution. Then, $atrace_1 = \langle (i\_send, \emptyset), (o\_popup, \{o\_dialog\}) \rangle$. Details on how to obtain an annotated trace are shown in (NOGUEIRA; SAMPAIO; MOTA, 2014). The annotated trace is then supplied to *TC_BUILDER*, already discussed in Section 4.1.2.3, to build a sound test case as illustrated in Listing 4.8.

Listing 4.8 – Building the test case

```
TC = ATG::TC_BUILDER(input_events, all_outputs, atrace)
```

Since the process *TC_BUILDER* is sound, as proved in Sampaio et al. (2014), the resulting test case is always sound and can then be linearized back to natural language. Each input event becomes a test step, while the corresponding output event is the expected result. Inconclusive outputs can be annotated in the comments. As we have seen in Rule 5, we can trace back the action (and its syntactic tree) using the *id* from the trace. With its syntactic structure, we can linearize the action back to English, as seen in Chapter 2. We can also control the mood and tense of the sentences: test steps are in imperative mood while expected results are linearized using the indicative form and past tense. In the sequel we present an example and discuss how to automate these sound and consistent test cases in natural language.

4.1.5 Remarks on finding scenarios

As discussed in Sections 4.1.2.1 and 4.1.2.2, test cases are created based on the premise of reaching the expected outputs described in the requirements. To achieve this goal,

we define the *Statement* extracted from the sentence as a relevant test purpose for each requirement in a feature. Consequently, each requirement has, in the end, only one corresponding test case. The advantage is that it proved to be more intuitive for the test analyst when debugging the generation process. The evident disadvantage, however, is that it may not cover the "negative" scenarios, in which one should test things that should not happen. For instance, consider the following requirement:

Only when data is active and the button is pressed then a success message should be shown

Applying the proposed generation strategy, we would only get a single scenario $\langle i\_activatedata, o\_a$ because it is the only scenario clearly described in the requirement. However, in many cases, when the condition is not met, the action in the statement should not happen. Thus, we allow the generation of these additional scenarios when the requirement describes a necessary condition or event, i.e., "only if" or "only when". Considering the requirement above, we would get three implicit requirements in addition to the original one:

When data is active and the button is pressed then a success message should be shown
When data is not active and the button is pressed then a success message should not be shown
When data is active and the button is not pressed then a success message should not be shown
When data is not active and the button is not pressed then a success message should not be shown

The total number of scenarios depends on the number of circumstances in the original requirement. It can be calculated by the combination $\binom{n}{k}$ with n as the number of circumstances and $k = 2$, since a circumstance has only two states: it either happens or it does not. There are some cases in which the additional requirements are too obvious or trivial, or even do not make sense. In these cases, the analysts can remove the test cases from the selection after generation. More on test case selection and prioritization is discussed in Section 8.1.

4.2 TEST CASE AUTOMATION

In this section we cover the task of automating test cases. Figure 13 illustrates the task in which test cases passed as input are mapped into test scripts, for the purpose of automated execution.

Figure 13 – Process Task: TC Automation

Source: The author (2022)

Besides receiving sound, consistent, and CNL-compliant test cases, we also deal with a more uncontrolled scenario in which legacy test cases, written in freestyle natural language, are passed as input for automation (as previously illustrated in Figure 2b). Since

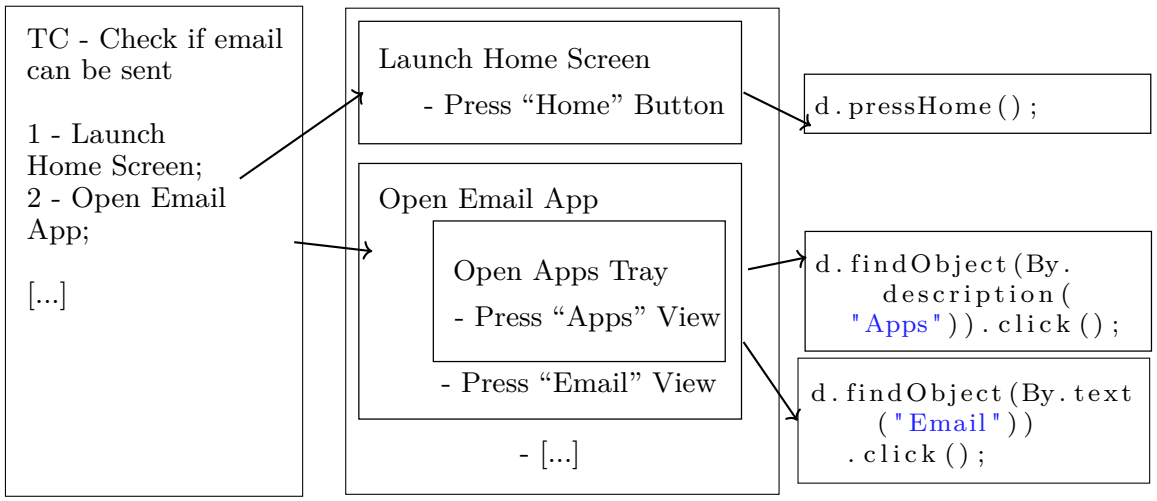
it is a business decision to: a) convert legacy requirements to use the new CNL or b) automate them as is, we designed two approaches (for the team to choose) to produce test scripts. In this light, the remainder of this section presents a strategy that can be used to automate test cases that are already sound and consistent with an annotated codebase, while Chapter 5 shows an alternative approach that maps freestyle descriptions into consistent test scripts in order to minimize the effort to automate legacy test cases or reuse legacy code.

4.2.1 Composite Actions

One must create a mapping between frames and concrete instructions to allow test drivers to interpret sentences as executable scripts. The mapping size should be kept at a minimum to reduce maintenance and to support fast transitions when adopting new frameworks and programming languages.

For that matter, we can leverage the compositionality of test actions, as introduced in Section 1.3.2, to map only a small number of (atomic) actions to scripts, and define other actions as a composition of those atomic actions. For instance, consider the TC illustrated in Line 14 (a) to check whether an e-mail can be sent. The test action representing this TC can be structured as a composition of other test actions (each representing a step). Each step, in turn, could also be composed of several screen interactions (expressed as atomic test actions), as presented in Line 14 (b). We potentialize the reuse possibilities and provide the code script by only interpreting the atomic actions, as seen in Line 14 (c).

Figure 14 – Test case automation using hierarchical test actions



(a) Test case written in CNL (b) Test action structure (c) Coded atomic actions

Source: The author (2022)

As discussed in the previous chapter, the output of the proposed test generation strategy are sound and consistent test cases that comply with our CNL. Because these test

cases are already consistent, there is no need to insert supplementary setup or reorganize the script since all dependencies and details defined in the domain model are already resolved. Only the mapping to execution methods (which the test adapter recognizes) is lacking. For this matter, we present here a straightforward strategy: direct code mapping. Table 3 illustrates a sample consistent test case and it will serve as reference input for the remainder of this section. It represents a single test case with 2 (two) setup actions and 4 (four) test steps with their corresponding expected results.

Table 3 – Sample consistent test case

Setup	**Step description**	**Expected results**
* Deactivate wifi * Email is logged in	Open email app	Email app was opened
	Write a new message	Message was written
	Submit a message	Message was submitted
	Check the popup	The popup is shown with 'Wifi is deactivated' as message

Source: The author (2022)

CNL-based Code Annotations

This straightforward strategy consists in mapping CNL-compliant sentences into the corresponding scripts. This direct association is possible due to the action representation parsed from the sentence. Following the frame theory described in Section 2.2, we can assign a unique identifier to the fixed frame slots (such as operation and patient) and then map each singular frame to the corresponding script method. We implemented this mapping mechanism using a proprietary framework adopted within the industrial context of our research. Because it is not publicly available, we instead present an analogous implementation using the Java programming language and the UIAutomator framework (ANDROID, 2015) for Android UI Testing. Listing 4.9 shows an example of this sentence-to-script mapping.

Listing 4.9 – Sample Java script mapping

```
@Smartest("press a button")
public void pressButton(String identifier, String description, String text) {

    UiSelector selector = new UiSelector().className("android.widget.Button");

    if (identifier != null) {
        selector = selector.resourceId(identifier);
    }
    if (description != null) {
        selector = selector.description(description);
    }
    if (text != null) {
        selector = selector.text(text);
    }

    UiDevice mDevice = UiDevice.getInstance(getInstrumentation());
    mDevice.findObject(selector).click();
}
```

In Listing 4.9, Line 2, we have a method declaration that is mapped to a sentence by the annotation in Line 1. The parameters have the same name as the slots the associated frame can have. Line 17 shows a method call on the variable *mDevice* that holds the API access to functions related to user interactions with the connected device. The API for the interaction with a device may differ from one automation framework to another. Listing 4.10 illustrates the different ways to reference the method declared above. Each leading line shows the sentence that must be executed and the following line displays the corresponding method found for the given sentence.

Listing 4.10 – Sample script executions

```
> execute("Press a button")
pressButton(null, null, null)
> execute("The button must be pressed with 'Apps' as description")
pressButton(null, "Apps", null)
> execute("Press the button using 'Send' as text and 's_btn' as identifier")
pressButton("s_btn", null, "Send")
```

Because the sentences are CNL-compliant, it does not matter how it was written since they will always be mapped to their corresponding method and the arguments. This mapping must be made for all atomic actions. Since all other actions are compositions of these basic actions, the automation effort is kept at a minimum. The choice of which actions should be atomic is entirely dictated by the team. In our scenario, there is a 1:1 mapping from each API native method available to a single action described in text. For instance, Listing 4.11 illustrates the definition of action composition whose atomic actions must be executed to perform the composite action. The indentation means that the action is a step of the parent action and so forth.

Listing 4.11 – Domain model example for action details

```
# Details
* Open email app
    * press the button with 'Apps' as text
    * press the button with 'Email' as text
* Write a new message
    * press the button with 'Compose' as description
    * write keyboard message with 'foobar' as text
* Submit the message
    * press the button with 'Send' as description
```

Then, assuming that the atomic actions are mapped considering the domain model described above, we can generate a code script that, when executed, gives a pass/fail result. A sample generation is shown in Listing 4.12. Analyzing the script, Line 5 defines an annotation that holds the test case identification for reporting. Line 6 shows another annotation that gives a concise description about what is being tested to help debugging. Lines 7 and 8 declare the dependencies needed to be executed beforehand. This mechanism must be implemented by the test driver. The method calls inside the method declaration correspond to the atomic actions that must be executed. For instance, lines 11 and 12 represent the atomic actions that compose the action of opening an email, as detailed in Listing 4.11.

Listing 4.12 – Sample TC script

```
1
@Inject SmartestDriver smartest;
3
@Test
5 @Tc("TCID-1234")
@Summary("Email message can't be sent when wifi is deactivated")
7 @Dependency("Wifi is deactivated")
@Dependency("Email is logged in")
9 public void testCase1234(){
    // Open email app
11  smartest.execute("press the button with 'Apps' as text")
    smartest.execute("press the button with 'Email' as text")
13  // Write a new message
    smartest.execute("press the button with 'Compose' as description")
15  smartest.execute("write keyboard message with 'foobar' as text")
    // Submit the message
17  smartest.execute("press the button with 'Send' as description")
    smartest.execute("check the popup with 'Wifi is deactivated' as text")
19 }
```

This direct mapping works well when the code is already annotated with CNL-compliant sentences. However, in projects that have a massive legacy code base, from which we can only extract the method name and associated comments, we must adopt a more flexible strategy of text-to-code mapping. This mapping can be done with search by text similarity and is the subject of the next chapter.

Even though finding test cases that describe inconclusive results is unusual in our

industrial context, we detail how the code mapping can handle this kind of verdict. Since the generation strategy annotates the inconclusive results, we only need to define how to implement the verification. The intuition is to execute the step, and if it fails and there is an annotated inconclusive result, the raised exception is caught and the action that represents the other possible event is performed. Then, the final verdict depends on the result of this last action: if the action fails and there is no other possible event, then the whole test fails; else, the test is marked as inconclusive. Listing 4.13 shows a sample implementation of an inconclusive verdict verification.

Listing 4.13 – Implementation of inconclusive verdict verifications

```
1 // ...
  public void testCase3456(){
3     // ...
      try{
5         smartest.execute("check the popup with 'Message was sent' as text");
      } catch (AssertionError | UiObjectNotFoundException e) {
7         smartest.execute("check the popup with 'No credit available' as text");
          Assumptions.assumeTrue(false, "An inconclusive verdict was found");
9     }
  }
```

Assuming a test case step such as $\langle ..., (o\_sendmessage, \{o\_nocredit\}) \rangle$, we first map the output *o_sendmessage* to code in Line 5. Then, in case its execution fails during verification, we test the other possible event (*o_nocredit*) of the current step in Line 7. If the corresponding action executes properly, it is a valid result but not the one we expect. In this case, we set the status as inconclusive in Line 8.

5 AUTOMATION OF LEGACY TEST CASES

As mentioned in Chapter 1, we also tackle a challenging industrial testing scenario in which the textual documents provided as input are test cases written in freestyle natural language. Since, in such cases, there are no requirements to rely on, it is not possible to verify whether the legacy tests are sound. Nevertheless, we can still verify consistency, given a domain model.

Additionally, because there is no standard to write these test cases, it is not possible, in general, to define a meaningful mapping between textual descriptions and automation code scripts. Due to the natural language imprecision, typically there is an abstraction gap between the NL test steps and the interface provided by automation frameworks. Thus, code scripts become scattered because there are no straightforward means to match and reuse test cases already automated.

We address this problem by applying the notion of *Test Action*, as discussed in the previous chapter. Because an abstract action is composed of progressively more concrete ones, it increases the chance of finding already automated actions, thus improving reuse. However, since there is no guarantee that the corresponding description complies with our CNL, reuse and mapping are performed by text similarity instead of frame structure correspondence.

5.1 ACTION REPRESENTATION FOR FREESTYLE TEST CASES

Legacy test cases often contain conditional expressions used for controlling the execution flow. Because these legacy TCs were created by hand with no consistency analysis, they usually have to check for multiple conditions to assert their execution is in the right state before verifying the verdict. To allow for a conditional control flow, we extend the behavior of test actions by allowing one extra interpretation: ordered choice. In most cases, to execute a test action, an interpreter executes all its subactions consecutively. However, if a test action is marked as an *OR Action*, then its subactions are treated as an ordered choice: the first one with precondition *true* is executed, and the other subactions are ignored. Thus, an *Or Action* allows some flexibility in the execution flow according to a given condition, such as different platforms and application versions.

For instance, there are different approaches to viewing the phone applications that were recently opened, depending on the platform. In some platforms, there is a specific button to accomplish this task. However, there are platforms in which it is necessary to press the *home* button twice. Considering this scenario, we can capture the flow of the *Open Recent Applications* action as an *OR Action*. The action itself would be composed of two different actions: *Press the Recent Apps Button* and *Open Recent Apps with Home*

Button. Because their parent is an *OR Action*, instead of executing both of them, the interpreter sequentially evaluates their preconditions and then call the first action whose precondition is true. Preconditions could check the presence of a specific button or analyze the current platform.

It is worth mentioning that, although actions can have an additional interpretation for freestyle test cases, it retains its structure introduced in Section 4.2.1. Another inherited property is that each action still has an associated frame.

5.2 MATCHING AND REUSE

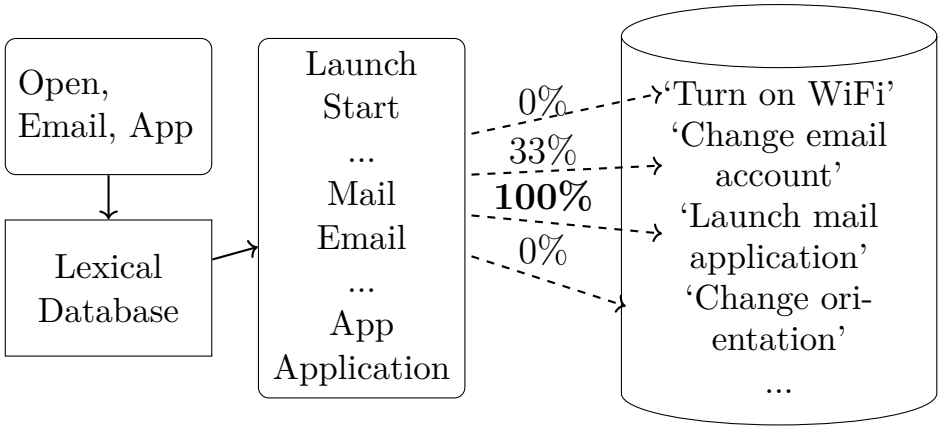
At the very beginning of an automation process, there are no previously created TCs, so there is no opportunity for reuse. However, as TCs are progressively being automated (either manually or via Capture-and-Replay (C&R)), test actions are gradually stored in a database and can be retrieved and reused in automating new TCs. Therefore, instead of capturing interactions all over again for every new TC, we employ an algorithm to match test steps written in freestyle English with the stored test actions, mitigating C&R issues regarding reuse—as noted in the empirical assessment conducted in Leotta et al. (2013a). These test actions may be organized and composed in any order to create yet more complex ones.

The matching process, detailed in Arruda, Sampaio and Barros (2016), is divided into three main operations: sentence tokenization; synonyms retrieval from a lexical database (currently, we use WordNet® Miller (1995)); and finally ranking of the test actions using synonym equivalence and string proximity.

The latter operation is executed for all stored action textual descriptions, to find the best similarity level. Finally, if no test action found is similar enough (for a given threshold), the user must implement a specific test action for the test step being processed. For instance, the matching process for the TC step "Open Email App" is illustrated in Figure 15.

In the tool that implements the overall C&R process, detailed in Section 6.1.2, the default matching score threshold is 50% but the user can use a different setting. Of course, there might always be a risk that the user considers a matching unsuccessful and eventually includes new sentences in the database that convey the same meaning as existing sentences. The evaluation of the tool, mentioned in Section 6.1.2, presents a significant result in terms of test actions reuse, and implementation effort. Even with these results, we still faced some problems regarding TC's dependency management and consistency due to ambiguous descriptions. This is one of the motivations for adopting a CNL for the descriptions of the stored actions. In this light, even though legacy TC descriptions are not standardized, we can match these descriptions via text similarity and require stored action descriptions to be written in compliance with our CNL. For the CNL semantics, we developed a consistency and dependency-checking strategy to

Figure 15 – Matching process



Source: The author (2020)

represent the test cases as valid test action sequences, according to these notions, as presented in previous chapters. Since there is no guarantee that these legacy TCs have associated requirements, we cannot verify soundness. However, given a domain model, one can ensure consistency, as detailed in the next section.

5.3 CONSISTENCY ANALYSIS OF LEGACY TEST CASES

When dealing with legacy test cases, it is difficult to verify whether the sentences define consistent artifacts without a proper semantic analysis mechanism. Such a mechanism should map them into an intermediate formalism to reason about context-sensitive properties, such as the consistency of a sequence of test actions.

As discussed in chapters 3 and 4, we encoded consistency analysis together with a conformance relation using a process algebra (CSP). However, unlike the generated tests from the previous chapter, legacy test cases have no associated formal semantics. Thus, soundness cannot be stated, let alone verified. Additionally, because the consistency analysis was performed at the specification level, it does not apply to legacy test cases directly.

In this light, we provide another encoding for domain models to perform consistency analysis for legacy test cases. Model finders are good candidates for semantic analyzers because they can point out inconsistencies and suggest valid alternative configurations. In our context, a model finder would serve different purposes, being of great value. When a test case is inconsistent because, for instance, some actions are missing, a model finder would be able to both identify the problem and also produce configurations that include the missing actions. Thus, a model finder has an enormous practical impact, as inconsistent test cases are not only identified but automatically made consistent. A notorious model finder is the Alloy Analyzer (JACKSON, 2012), which uses the Alloy modeling language.

Several other alternatives could be used for this purpose, including a process algebra

such as CSP and refinement verifications to implement the consistency analysis. The same can be done using theorem provers for model-based languages like Z (WOODCOCK; DAVIES, 1996) and B (SCHNEIDER, 2001). Although solvers other than Alloy are also possible alternatives, we decided to use Alloy for its simplicity and expressiveness. Particularly its modeling language allows specifying action hierarchies; this significantly simplifies the representation of the domain model, as we illustrate in the next section. Another important aspect is the bounded verification, which allows us to easily restrict the scope size of the analysis, thus mitigating state explosion when reordering tests.

5.3.1 Alloy and the Alloy Analyzer

Alloy (JACKSON, 2002) is a declarative modeling language. It was inspired by other formal notations, such as Z (SPIVEY; ABRIAL, 1992), VDM (JONES, 1990), and OCL (RICHTERS; GOGOLLA, 1998), and provides ways to describe models succinctly, in terms of sets and relations. Listing 5.1 presents a snippet that describes a simple model in Alloy.

Listing 5.1 – Alloy Model - Example

```
sig NaturalNumber {succ: one NaturalNumber,
                   predc: lone NaturalNumber}

assert PeanoAxiomAdapted {
  ∀ x,y,z: NaturalNumber  •
      x in y.^predc and y in z.^predc ⇒ z in x.^succ
}

check PeanoAxiomAdapted
```

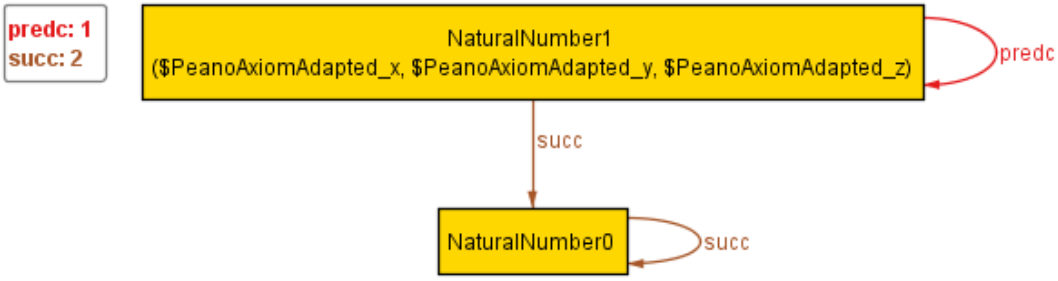
Lines 1 - 2 A signature has a similar meaning as types/classes in Object-Oriented Programming. However, a signature defines a set; in this example, a set named `NaturalNumber` has "fields", which are actually relations to sets. The field `succ` and its quantifier `one` indicate that a `NaturalNumber` has only one successor; `lone predc` defines that it can have zero or one predecessor.

Lines 4-7 An assertion is defined to verify the described property in the given model. In this case, an adaption of a well-known Peano's axiom for arithmetic: $\forall\, x, y, z \in N(x < y \land y < z \Rightarrow x < z)$.

Line 9 Checks the assertion by considering a default finite scope of 3 (three) elements. That is, Alloy will examine all examples that have up to 3 (three) natural numbers. In this case, there is a counterexample illustrated in Figure 16. It shows the graphic visualization provided by the Alloy Analyzer for the analysis output. The atoms are represented as rectangles, and the relations among them are displayed as line

connectors. A `NaturalNumber` that is not the predecessor of its successor may seem counterintuitive, but it is precisely to find these non-obvious modeling issues that Alloy is useful.

Figure 16 – Alloy Analyzer output - Counterexample found



Source: The author (2020)

To resolve this modeling problem, we present the Listing 5.2, which provides an extra predicate and a fact for the model shown in Listing 5.1.

Listing 5.2 – Alloy Model - Fixing the model incrementally

```
[...]
pred Ordering (x,y: NaturalNumber) {
   x.succ = y ⇒ y.predc = x
}
fact {
   ∀ x,y: NaturalNumber • Ordering[x,y]
}
[...]
```

Lines 2-4 The predicate holds if the parameters `x` and `y` satisfy the constraints listed in the body. It says that if a given `NaturalNumber x` has a successor `y`, then `x` should be the predecessor of `y`.

Lines 5-7 The *fact* construct behaves as an invariant: for all pairs of natural numbers, the `Ordering` predicate applies.

With these additions, the `check` command of the previous model presents no counterexample.

5.3.2 Detailed semantics

Before presenting the predicates that capture the semantic mechanism to ensure test case consistency, we define, via semantics in Alloy, key concepts that so far have been only

discussed in conjunction with requirements. These concepts are presented through top-down definitions of the syntactic elements, rules that define their semantics in Alloy, and some examples.

A domain model includes the relevant actions, frames, and how they relate to each other. As already explained, an action can be either atomic or composite. An atomic action is an abstraction of a concrete instruction (an implementation) to be directly executed on an SUT.

A composite action is a procedure abstraction, formed of test actions (atomic or composite); it is recursively interpreted and evaluated on an SUT. The elements of a composite action are as follows.

P_{re} is the action precondition; it consists of the set of basic actions ($P_{re} \subset AtomicAction$) used to check on-the-fly whether the composite action can be successfully executed (the focus here is on automation/execution rather than on specification);

sequence is a sequence of actions, $sequence \in seq\, Action$, which defines the composite action; and

P_{ost} is the composite action postcondition; it is a set of basic actions ($P_{ost} \subset AtomicAction$) used to check whether the desired effect is achieved after the execution, in which case the composite action is marked as **passed**.

Note that, for composite actions, the notion of pre- and postcondition is expressed as (executable) basic actions, rather than as the more usual logical style expressed in terms of propositional or predicate calculus. Particularly, only atomic actions that do not modify the state of the system and return a boolean value can be used in pre- and postconditions.

A frame, derived from a sentence in natural language, represents a specific situation, as discussed in Chapter 2. The definition of frames uses some additional sets: *Operation* is the set of all possible operations, *Patient* is the set of all possible patients, and *Extra* is the set of all extra slots. These sets correspond to the same notion in the frame theory presented in previous sections.

In the current context, we associate an action directly with a frame. Thus, our frames are a semantic representation of the sentence associated with the action; the actual elements of the action are not represented in the frame. The semantic reasoning considers well-formedness and associations among frames, representing a linguistic perspective that is not action-dependent.

With frames, we can define the well-formedness of individual actions as well as establish relationships among these actions. We are able to guarantee, for instance, that a sequence of test actions is consistent in the sense that it is self-contained and can be successfully executed without any missing actions. As already discussed in Chapter 3, a domain model describes these relations. In this context, we only consider two kinds of relationships: an

action can depend on another action (for instance, the action *Send Message* depends on a previous occurrence of *Activate Connection*), and an action can cancel another action (for example, the action *Activate AirPlaneMode* cancels the effect of *Activate Connection*).

Therefore, an association is a relation between frames whose semantics is distinct in case it characterizes a dependency (an action requires another action to be executed beforehand) or a cancellation (the current action cancels the effect of another action executed before). The first frame characterizes the source frame of the relation. The type establishes the association type (dependency or cancellation). Lastly, the second frame determines the target frame of the relation.

5.3.2.1 Well-formedness Conditions

The metamodel described previously is able to generate invalid models, as it captures only context-free structures. Therefore, we present some well-formedness conditions that specify which models are valid.

Considering a given generated model $dm : DomainModel$, $F_{set} \subseteq dm.frames$, $As_{set} \subseteq dm.associations$, $Ac_{set} \subseteq dm.actions$, then we have the following well-formedness conditions:

1. $\forall f_1, f_2 : Frame;\ t : Type \mid (f_1, t, f_2) \subset As_{set} \bullet f_1, f_2 \subseteq F_{set}$
2. $\forall f_1, f_2 : Frame \mid f_2 \subset f_1.dependencies \bullet \neg(f_1 \subset f_2.\hat{}\,dependencies)$
3. $\forall f : Frame \mid f \subset F_{set} \bullet f.action \subset Ac_{set}$
4. $\forall a : Action \mid a \subset Ac_{set} \bullet \{a.pre + a.sequence + a.post\} \subset Ac_{set}$

The first condition determines that all frames from a given association must also be present in the domain model. The second condition dictates that there must not be circular dependencies in a valid domain model. The third condition enforces that every action mapped from a frame must also be listed in the domain model. The last condition determines that every action that composes another action must be in the domain model too.

5.3.2.2 Semantic Rules

The semantics is inductively defined on the structure of a domain model. Each rule has the form $[\![s :< SyntacticElement >]\!] : AlloyModel$, where $[\![.]\!]$ is the semantic function. In general, the body of each rule includes Alloy elements and meta notation used to structure the semantic definition. The meta notation is underlined.

The first rule defines the semantics of a domain model using functions to handle its actions, frames, and associations. Each of these functions is defined by specific rules.

Rule 12. Domain Model ⟦ domain: DomainModel ⟧: AlloyModel =

```
actionSequence(domain.actions)
frameSequence(domain.frames)
defineAssociations(domain.associations)
```

Rule 13 defines the abstract signatures for action elements and uses a recursive function to define all individual actions.

Rule 13. Action Sequence actionSequence(actions: Seq(Action)): AlloyModel =

```
abstract sig Action {
    pre: set Condition,
    sequence: seq Action
    post: set Condition
}
abstract sig CompositeAction extends Action {
    sequence: seq Action
}
sig Id, Variable, Slot extends String {}
sig Value in String + Int {}
abstract sig AtomicAction extends Action {
    parameters: set Variable -> Value,
    device: one Device
}
defineActions(actions)
where
    defineActions(actions: Seq(Action)): AlloyModel ≙
    if #actions > 0 then
        ⟦head actions ⟧
        defineActions(tail actions)
```

Rule 14 verifies first whether the action is atomic or composite. In case it is atomic, it also defines:

parameters is a set of input variables for the chosen command;

device is the destination target where the atomic action must be executed.

When it is a composite action, the function lists its properties

Rule 14. Action ⟦action: Action ⟧: AlloyModel =

```
if action.isAtomic
  then
  abstract sig Actionaction.name extends AtomicAction {} {
    parameters = { join(map(params, λ param: param.var -> param.value, , ) }
    device = action.device
  }
  else
  one sig Action action.name extends CompositeAction {} {
    pre = {listA(action.pre)}
    sequence = {listA(action.subactions)}
    post = {listA(action.post)}
  }
  where
    listA(elements: Seq(Action)): AlloyModel ≙
      join(map(elements, λ elem: Actionelement.name, , )
```

Regarding atomic actions, an example is given in Listing 5.3. The signature name identifies the command to be executed (in this case, *Click on a view with a given description*), and the relevant input variable is the *DescriptionValue* and the device with *SystemUnderTestOne* as reference.

We also provide an example of a composite action (see Listing 5.4): *Send Email Message* has the goal of opening the e-mail application. It first ensures, via its precondition, that the current layout has a button with the description *Apps*. The sequence of actions that defines the action behavior includes a composite action that opens the app tray and an atomic action. Finally, the postcondition contains an atomic action that checks if the email was sent.

Listing 5.3 – Example of an atomic action in Alloy

```
sig ClickOnDescription extends AtomicAction {} {
  parameters =  Description -> Value
  device = SystemUnderTestOne
}
```

Listing 5.4 – Example of a composite action in Alloy

```
one sig OpenGmailApp, WriteAnEmail
    extends CompositeAction{}
sig PressButton extends AtomicAction {}
sig Description, Title, Text extends Variable {}
sig GMail, HelloWorld, Send extends Value {}
[..]

one sig PressSendButton extends ClickOnDescription{} {
  parameters = { Description ->  Send }
}

one sig SendEmailMessage extends CompositeAction {} {
  pre = IsInAppsTray
  post = EmailMessageSent
  sequence = { 0 -> OpenGmailApp +
               1 -> WriteAnEmail +
               2 -> PressSendButton }
}
```

In the previous example, the first action that composes *Send Email Message* is also a composite action *Open Gmail App*, whose details we omit here. It illustrates the composite pattern discussed in Section 1.3.2. The atomic action *Press Send Button* extends the one defined previously.

Composite actions can be parametrized by variables instantiated within the set *Variable*. However, instead of a direct specification, they are inferred from the atomic actions that compose them.

Rule 15 defines the abstract signature for frames and calls the recursive function to list each one.

Rule 15. Frame Sequence frameSequence(frames: Seq(Frame)): AlloyModel =

```
abstract sig Frame {
    operation: one Operation,
    patient: one Patient,
    extra: set Slot,
    action: one Action,
  }
  defineFrames(frames)
  where

  defineFrames(frames: Seq(Frame)): AlloyModel ≙
    if #frames > 0
```

```
    then
      [[head frames]]
      defineFrames(tail frames)
```

Rule 16 defines each individual frame and its properties in Alloy.

Rule 16. Frame [[frame: Frame]]: AlloyModel =

```
one sig Frameframe.name extends Frame {
    operation: Operation frame.operation.name
    patient: Patient frame.patient.name
    slots: {join(map(slots, λ slot: Slotslot.name, , )}
}
```

Rule 17 defines an abstract signature for all associations and two sub-signatures to represent the type: dependency or cancellation. Then, it uses a recursive function to define each association individually.

Rule 17. Association Sequence associationSequence(associations: Seq(Association)): AlloyModel =

```
abstract sig Association {
    source: Frame,
    target: Frame
} abstract sig Dependency, Cancellation extends Association {}
defineAssociations(associations: Seq(Association))
where
    defineAssociations(associations: Seq(Association)): AlloyModel =
      if #associations > 0] then
        [[head associations]]
        defineAssociations(tail associations)
```

Rule 18 verifies whether an association is a dependency or cancellation to extend the corresponding signature.

Rule 18. Association [[association: Association]]: AlloyModel =

```
    one sig relName(association.type) sourceId + targetId extends relName(association.type) {}{
      source in FramesourceId
      target in FrametargetId
      if #association.matching > 0 then
```

matching = {join(map(association.matching, λ match: match.source → match.target), +)}
}
where
sourceId = association.source.id
targetId = association.target.id
relName(type): String =
if type == cancellation: Cancellation
else if type == dependency: Dependency

We developed a strategy to automatically check the well-formedness of the actions as well as their dependencies, in order to provide a coherent (and possibly optimal) execution order. These dependencies are represented in terms of signatures that extend the base *Association*. For instance, Listing 5.5 shows some examples of associations among frames that are used to check the consistency of an action sequence. By inspecting the *DeletionNeedsCreation* signature, we notice that in order to erase a message, first a message must have been sent. Additionally, the matching rules must apply: the author, title, and recipient of the message to be erased must correspond to the sender, title and receiver specified when the message was sent.

Listing 5.5 – Associations in Alloy

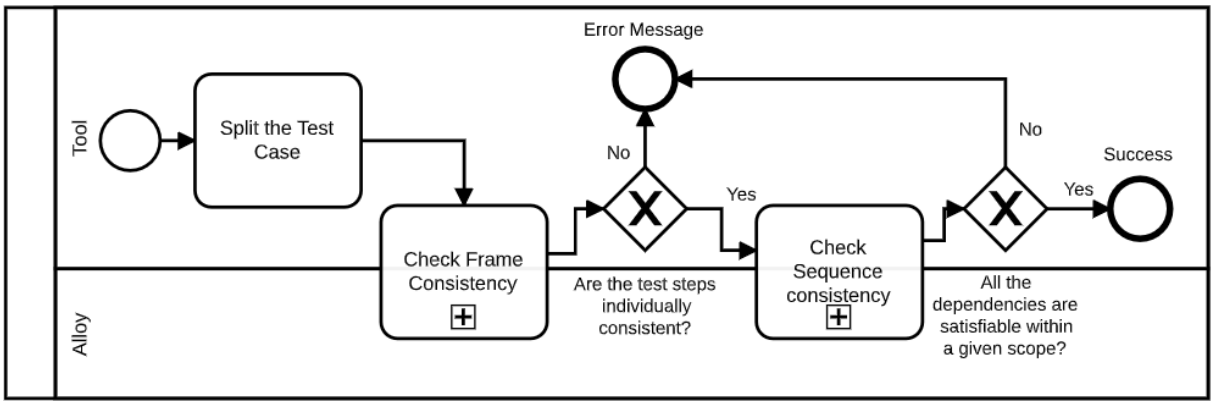
```
one sig MessageNeedsConnection extends Dependency {} {
    source in SendMessageFrame
    target in ActivateConnectionFrame
}
one sig DeletionNeedsCreation extends Dependency {} {
    source in EraseMessageFrame
    target in SendMessageFrame
    matching = { (Author -> Sender)
               + (Title -> Title)
               + (Recipient -> Receiver) }
}
one sig AirplaneCancelsConnection
    extends Cancellation {} {
    source in ActivateAirplaneModeFrame
    target in ActivateConnectionFrame
}
```

The automatic generation of the predicates to carry out model finding using the Alloy Analyzer is captured by three additional rules; this is addressed next.

5.4 THE OVERALL CONSISTENCY ANALYSIS PROCESS FOR LEGACY TEST CASES

The consistency analysis strategy involves defining: (1) which actions are individually valid by evaluating the associated frames; (2) what are their dependencies and cancellations; and (3) how to correctly order actions or which actions can be inserted to allow the execution of a set of test cases. The valid actions and their associations are represented as a domain model. Then, for every execution request, a predicate is evaluated to find a valid sequence of test actions. The systematic process is shown in Figure 17 and is detailed next.

Figure 17 – Overall Consistency Analysis for Legacy TCs modeled using BPMN



Source: The author (2020)

The next two sections detail the verification of frame consistency and frame sequence consistency.

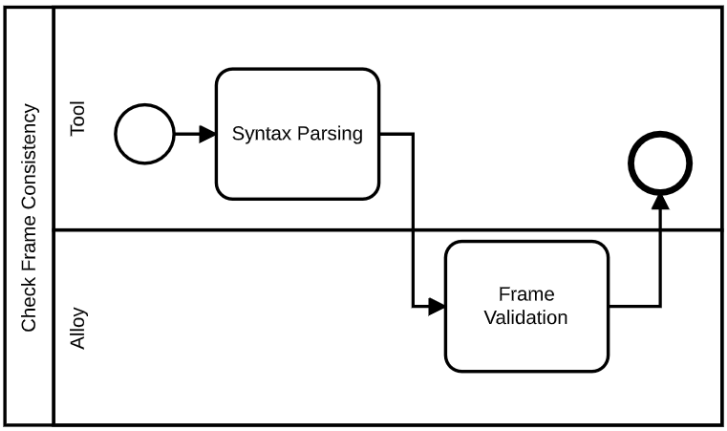
5.4.1 Split the test case

A test case is itself a composite action. The consistency analysis works by checking consistency for the associated frame (via the domain model mapping) of each action and then proceeds recursively, considering its direct children.

5.4.2 Frame consistency

The frame consistency analysis detailed in Figure 18, covers two stages: (1) Syntax parsing; and (2) Frame validation. They are detailed next using the definitions from the previous section.

Figure 18 – Frame consistency analysis modeled using BPMN



Source: The author (2020)

Syntax parsing

To obtain the frames, each test step description—that complies with our CNL—is parsed and represented as a frame instance. It is also worth mentioning that, because the parser is dynamic, they automatically change when there are new grammar entries, avoiding developer intervention or unavailability due to parser generation.

Frame Validation

We use frames, as defined in Section 2.2, to represent knowledge extracted from sentences that comply with our controlled natural language. These frames can be dynamically defined by a user, filling the operation, patient, and the extra slots that together represent a valid and consistent frame. To check consistency, the tool verifies whether the extracted frame is present in the domain model which is dynamically generated as an Alloy module (see Section 5.3.2.2).

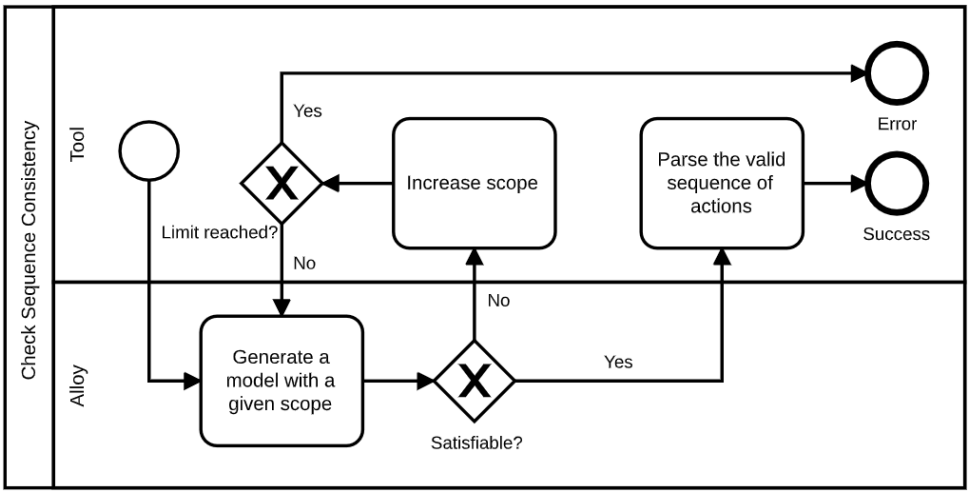
5.4.3 Sequence Consistency

The sequence consistency analysis, which evaluates whether the associated sequence of actions can actually be executed, consists of three activities (assuming an initial scope): (1) Generate a model with a given scope; (2) Increase scope; and (3) Parse the valid sequence of frames. These activities, also illustrated in Figure 19, are detailed next.

Defining an initial scope

Scope definition in Alloy means the number of instances of each signature used in the bounded verification. When generating an instance from a model, it is necessary to define a scope for limiting the analysis. The scope should start small because each interaction might significantly increase the time to find a satisfiable solution. Therefore, the first

Figure 19 – Sequence consistency analysis modeled using BPMN



Source: The author (2020)

scope defaults to the number of test steps analyzed, to check if they can be executed without introducing any extra action.

Generate a model with a given scope

Assuming that each step is individually consistent, the tool has to analyze the dependencies and reorganize or even introduce other frames to ensure global consistency. The model generation follows a similar principle to that of the previous step: dependencies and cancellations are defined by a user, which serves as input for the Alloy model generation. The following rules show how we generate not only the semantics of the domain model (Section 5.3.2.2) but also the predicates and necessary elements to carry out the analysis.

Rule 19. Program ⟦ program: Program ⟧: AlloyModel =

```
open util/ordering[State]
    sig State {
      conditions: set Action,
      currentAction: one Action
    }
    ⟦program.domain ⟧
    generatePredicates(program.steps, program.strategy)
    modelFinding(program.strategy)
```

Rule 20. Generate Predicates generatePredicates(steps: Seq(Action)): AlloyModel =

```
pred valid(s: State) {
      let s' = s.next | {
```

```
      some s' implies s'.conditions = s.conditions + s.currentAction - cancellations[s.currentAction]
      all d:Dependency | {
        d.source = s.currentAction implies d.target in s.conditions
      }
    }
}
fun cancellations(a: Action): set Action {
    { cancels: Action | {
      all c:Cancellation {
        c.target in cancels iff a in c.source
      }
    } }
}
pred find {
    first.conditions = none
    all s:State | valid[s]
    {for action in steps {
      some s:State | {
        s.currentAction = Actionaction.id
      }
    }
}
pred inject {
    first.conditions = none
      all s:State | valid[s]
    some join(map(steps, λ action: saction.id , ): State | {
      for action in steps {
      saction.id.currentAction = Actionaction.id
      let previous = before(action, steps) in
        if previous then
          saction.id in sprevious.n̂ext
        }
      }
    }
}
```

Rule 21. Model Finding modelFinding(strategy: Strategy): AlloyModel =

---

run strategy.name for exactly strategy.scope State

For instance, to *Send an Email Message*, first one has to *Login* in that email account, so the sender and the logged account should match—or when the user activates

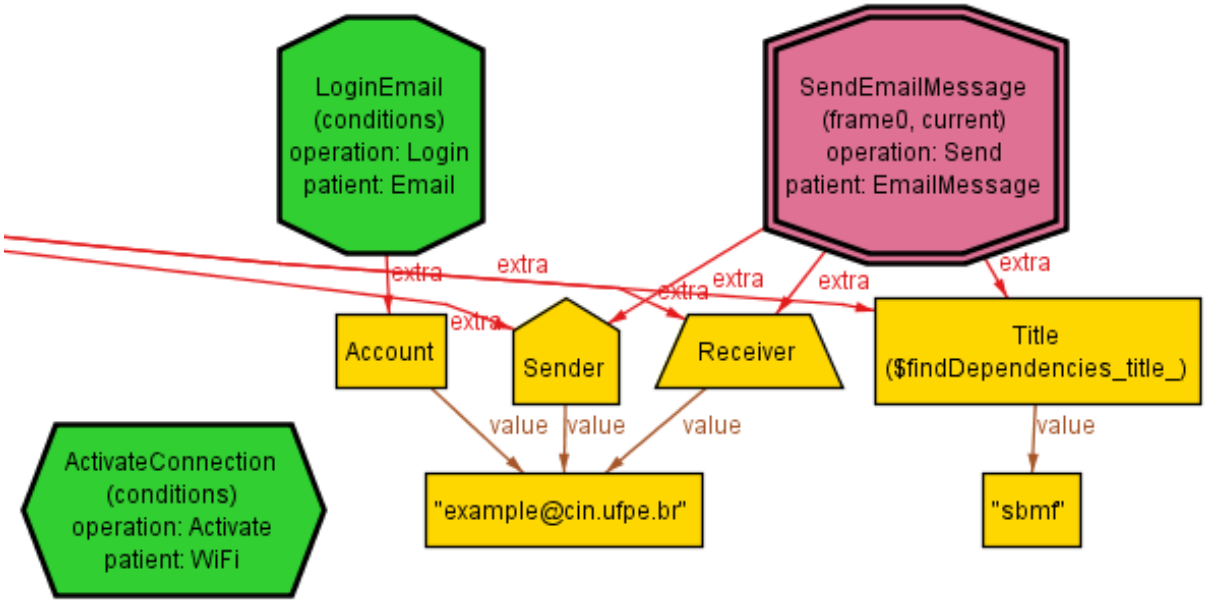
the *Airplane Mode*, all conditions from any action that activates a connection must be discarded. Rule 20 generates three predicates and a function that are used to analyze the dependencies and cancellations of a given frame from the defined associations.

Nevertheless, the model still lacks the notion of a sequence of steps. We then modeled the execution flow as a sequence of `State`, each one associated with the activated conditions, besides the frame whose associated action is currently being performed. In Rule 20, there is a predicate *valid* in which each state transfers its conditions and the current frame to the next state, but removing frames that are canceled by the current one.

To find a valid sequence of `State`s, a corresponding predicate is generated. For instance, considering the predicate *find* shown in Rule 20 and a frame "*Sending an Email message*" with a title slot "sbmf", then values for the sender and the message body could be arbitrarily chosen since the predicate does not fix this.

In this case, an example of the Alloy output after running the predicate (see Rule 21) is illustrated in Figure 20, in which the last `State` (of a scope of three `State`s) is projected. The current frame is shown in the pink color (*Send an Email Message*); *ActivateConnection* and *LoginEmail* are represented as conditions inherited from previous `State`s that are necessary to execute the current action.

Figure 20 – Alloy analyzer illustrating the model found for a valid sequence of States



Source: The author (2020)

Another interesting application of frame sequence consistency is to optimize a test suite, possibly merging test cases. For example, Table 4 summarizes how our tool takes two test cases as input and gives another consistent sequence, but is more efficient in terms of the total number of steps. Not only does it recognizes that some actions inherit

behavior from others, but it also takes advantage of the matching behaviors to optimize the suite by merging some steps.

Table 4 – Dynamically rearranging test cases executions

Step	**Normal**	**Merged TC**
1	TC1 - Activate Connection	Activate [WiFi]
2	TC1 - Login into Email Account	Login into Email Account
3	TC1 - Send email	Send email [with attachment] [to yourself]
4	TC1 - Check sent messages	Check sent messages
5	TC2 - Activate WiFi	Download attachment
6	TC2 - Login into Email Account	Check downloaded attachment
7	TC2 - Send email with attachment to yourself	
8	TC2 - Download attachment	
9	TC2 - Check if it has been downloaded	

Source: The author (2022)

Increase Scope

If it is not possible to find a solution, then the scope is gradually increased until a given limit (because it becomes costly, timewise, to carry on).

Parse the valid sequence of frames

To interpret the results generated by the Alloy Analyzer, the tool retrieves the XML file and parses it into an object that is expected by the consolidation function. The process involves getting the information via XPath queries and transforming them into a JSON object. Then, the consolidated results can be shown to the user as natural language descriptions or in a structured form.

In summary, a dependency analysis can both detect inconsistent sequences, as well as automatically insert actions to turn an inconsistent sequence into a consistent one. The main challenge here is scalability, which is addressed in the next chapter.

6 TOOLS, AND EVALUATIONS

In this chapter, we discuss the practical aspects of our work by employing custom tools in an industrial context. These tools were developed and evaluated within an industrial context of a partnership with Motorola Mobility. We also present the results of evaluating the strategy via these tools. Section 6.1 shows an overview of each tool and some implementation details. Then, Section 6.2 presents the conducted experiments, including their corresponding research questions, design, and results.

6.1 TOOLS

In this section, we present the tools that implement our strategy. As we address the complete process from requirements specification in a CNL to test script generation, and also consider legacy test cases, we created several tools for different tasks, which complement each other. First, we present *SmarTest* in Section 6.1.1 which is responsible for the test generation from features that comply with our CNL. Then, in Section 6.1.2 we discuss the different tools implemented to deal with legacy test cases with freestyle natural language descriptions.

6.1.1 *SmarTest*: Generation and Automation from Features

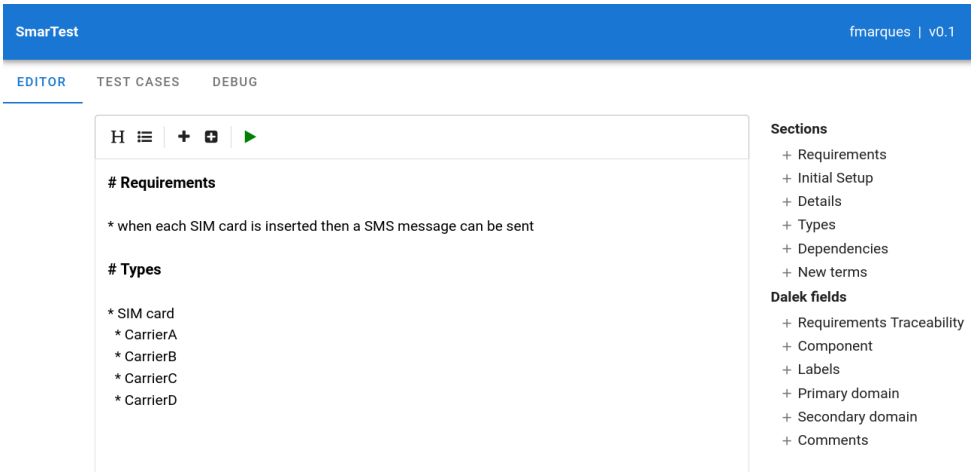
SmarTest is a web-based tool in which test architects and analysts can input the relevant requirements, in compliance with our CNL, and get automatically generated test cases from them. Because we wanted to reduce friction in order to increase user adoption, the tool was developed to run in the browser and with a familiar interface. The interface was designed to look like a text editor (frequently used in our context to write requirements and comments) and to keep the same naming convention the users use in their daily activities and artifacts. Because most software systems in our context are web-based, the tool also runs in a browser to ease the user adoption. Regarding the techonology stack, the tool backend is implemented in Java, because both GF and FDR have official Java libraries for their APIs, using the Spring Framework[1]. The frontend, in turn, is implemented in Javascript and Vue[2].

Figure 21 shows the latest user interface of *SmarTest*. In the left panel there is a text area in which the requirements and domain model are written. For better organization, we use one document for each feature by convention. The feature document in the tool is structured following Markdown[3] rules. The first section, with the header “Requirements”,

[1] https://spring.io/
[2] https://vuejs.org/
[3] https://daringfireball.net/projects/markdown/

should contain one or more sentences representing each requirement. All sentences must be items of an unordered list, which in Markdown are denoted preceding the sentence with an asterisk, plus sign, or hyphen.

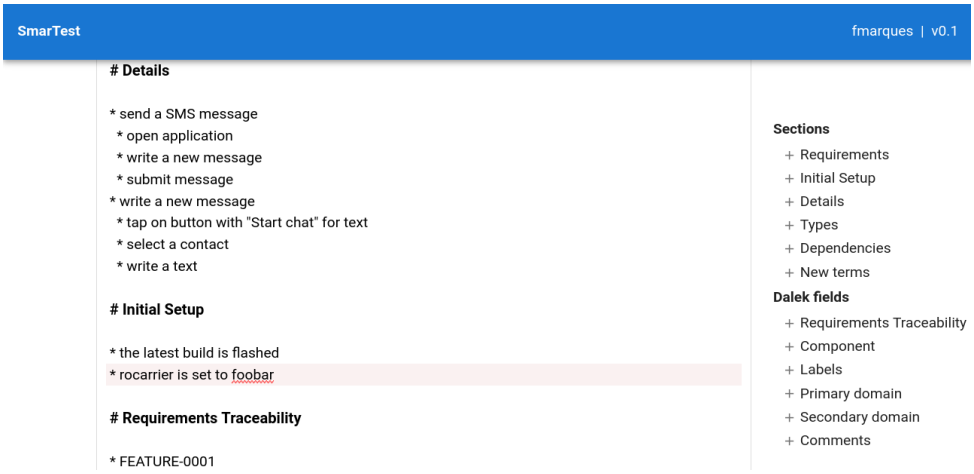
Figure 21 – *SmarTest* interface



Source: The author (2022)

The right panel contains suggested sections that can be inserted into the document. The first category of sections is used to fill out the specification, including the domain model. The second category represents sections from which their values will be directly mapped to fields of external test management tools. Figure 22 shows another excerpt of the feature document that comprises the domain model.

Figure 22 – *SmarTest* - Text excerpt containing domain details

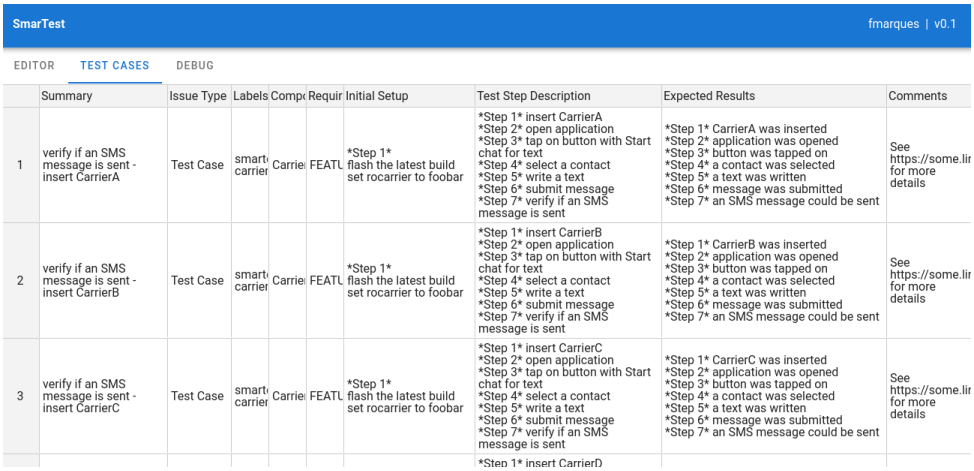


Source: The author (2022)

The domain model can be represented by declaring a section for each association type. For instance, Figure 22 shows how to model the “Details”, i.e., the sequence of concrete actions that represent a given abstract action. Each leading sentence is the main element or source (see Section 3.3), and the sub-elements are the targets. This figure also shows

how errors are reported. For instance, because the “foobar” is not a valid word in this domain, it is highlighted in red. When editing, the tool can also autocomplete sentences based on the grammar defined in Chapter 2. After inserting the requirements, domain model, and extra fields (including a vocabulary for new terms), the user can then ask the tool to generate the test cases by pressing the “play” button on the toolbar above the text area. The test cases are shown on the “Test Cases” tab, as seen in Figure 23.

Figure 23 – *SmarTest* - Test generation output



	Summary	Issue Type	Labels	Compo	Requir	Initial Setup	Test Step Description	Expected Results	Comments
1	verify if an SMS message is sent - insert CarrierA	Test Case	smart carrier	Carrie	FEATU	*Step 1* flash the latest build set rocarrier to foobar	*Step 1* insert CarrierA *Step 2* open application *Step 3* tap on button with Start chat for text *Step 4* select a contact *Step 5* write a text *Step 6* submit message *Step 7* verify if an SMS message is sent	*Step 1* CarrierA was inserted *Step 2* application was opened *Step 3* button was tapped on *Step 4* a contact was selected *Step 5* a text was written *Step 6* message was submitted *Step 7* an SMS message could be sent	See https://some.lir for more details
2	verify if an SMS message is sent - insert CarrierB	Test Case	smart carrier	Carrie	FEATU	*Step 1* flash the latest build set rocarrier to foobar	*Step 1* insert CarrierB *Step 2* open application *Step 3* tap on button with Start chat for text *Step 4* select a contact *Step 5* write a text *Step 6* submit message *Step 7* verify if an SMS message is sent	*Step 1* CarrierB was inserted *Step 2* application was opened *Step 3* button was tapped on *Step 4* a contact was selected *Step 5* a text was written *Step 6* message was submitted *Step 7* an SMS message could be sent	See https://some.lir for more details
3	verify if an SMS message is sent - insert CarrierC	Test Case	smart carrier	Carrie	FEATU	*Step 1* flash the latest build set rocarrier to foobar	*Step 1* insert CarrierC *Step 2* open application *Step 3* tap on button with Start chat for text *Step 4* select a contact *Step 5* write a text *Step 6* submit message *Step 7* verify if an SMS message is sent	*Step 1* CarrierC was inserted *Step 2* application was opened *Step 3* button was tapped on *Step 4* a contact was selected *Step 5* a text was written *Step 6* message was submitted *Step 7* an SMS message could be sent	See https://some.lir for more details
							Step 1 insert CarrierD		

Source: The author (2022)

The output test cases are presented within an editable spreadsheet using a template that can be parsed by external tools. Users might decide to keep only some test cases. They are also allowed to edit any field before exporting to *csv*. It is worth mentioning that our tool also provides an autocomplete mechanism (backed by Grammatical Framework (GF)) to help writing the specification and even show inline errors.

The design of the tool is modular, which allows interchangeable generation mechanisms. Even though we have implemented the mechanism presented in Chapter 4, FDR (used for automated refinement verification) cannot be used in production without further licensing. For this reason, another module that uses a version of the TaRGeT tool was implemented (which does not use FDR) together with Kaki, described in the next section, to generate tests and make them consistent, respectively.

6.1.2 Automation of Legacy TCs

In this section we briefly describe some tools that compose our strategy to automate legacy TCs written in freestyle natural language, as well as some assessment of their use. These TCs do not necessarily have associated requirements to rely upon, so our fully automatic strategy (from requirements to scripts) is not feasible. The first tool, *Force IDE*, is an evolution of a previously developed tool (ARRUDA; SAMPAIO; BARROS, 2016) with different requirements from our industry partner to enable testers and developers to

share test scripts. The original tool allowed testers to capture interactions with a mobile device and associate them with natural language descriptions that are used later on to match with similar sentences from other test cases, improving reuse. With the current evolution, scripts made by developers are also associated with natural language sentences and indexed for reuse.

The other tool, Kaki, was developed (ARRUDA, 2017) to waive the final user from manually performing the consistency analysis discussed in Chapter 3. In this work, we contribute in three additional aspects. First, a rigorous semantic definition(Section 5.3.2). Second, a performance evaluation of Kaki concerning the time spent to check frame consistency of test cases from, this time, a real repository. Finally, motivated by the feedback regarding dependency analysis response time in Section 1.1, even though the performance was acceptable, we still pursued near real-time response. Thus, we present another version focused on performance by using a different modeling approach (Section 6.2.3).

As discussed in Chapter 5, there are many TCs already created using ad-hoc strategies without any standardization. To support their automation, the general idea is to match test step descriptions with existing test actions *via* text similarity and to allow testers to create new test actions by using C&R. Next, we present *Force IDE*, which implements this strategy.

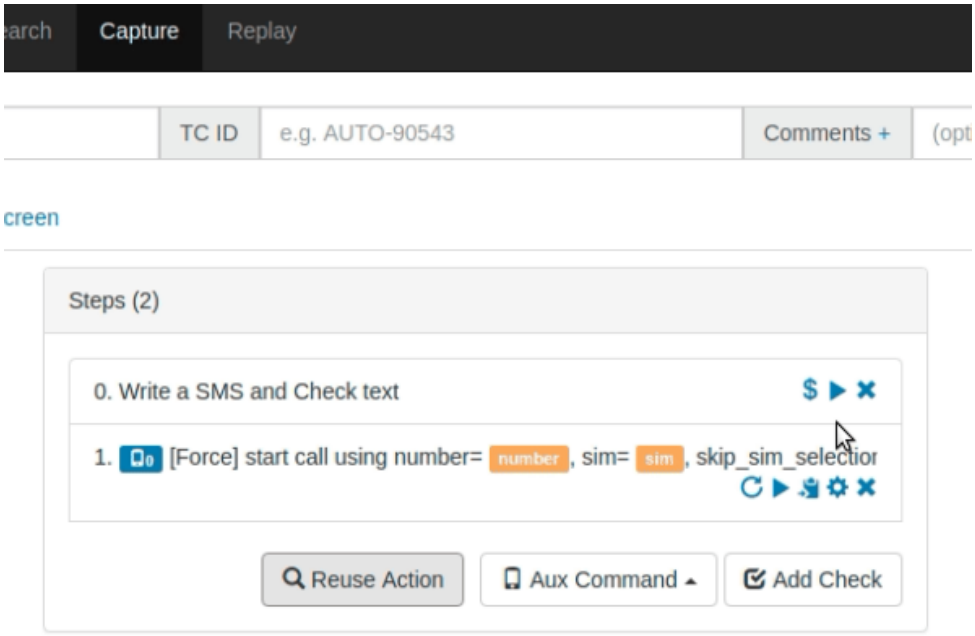
6.1.2.1 *Force IDE*: Automation with reusable indexed scripts

Our industry partner requested that testers should reuse scripts automated by developers and also contribute to the code repository. This is a different strategy from *AutoMano* (ARRUDA; SAMPAIO; BARROS, 2016), since the end users were only testers. Also, the commands were sent to an interpreter and not translated into plain code. Then, *Force IDE* began as an internal hard fork of *AutoMano*, but with different execution methods and codebase. Still, it remains a web-based tool. The backend was implemented in Java, and uses external dependencies such as Wordnet (MILLER, 1995) for searching synonyms and Solr [4] for indexing the actions descriptions. The frontend was implemented as a single page application using Javascript and Angular [5].

The overall idea is that the existing test script codebase, made by developers, is indexed, and all testing methods (we call controller actions) converted into test actions. Then, the tester can reuse scripts made by developers transparently. All test cases automated by the testers can also be merged back into the codebase since there is a code generator plugged in. Figure 24 illustrates a test script that was imported from the codebase. The second action with the "Force" tag is originally a test method. Its text description, "start call using...", is derived from the method signature $start\_call(number, sim, skip_s im_s elector)$, while the parameters are automatically extracted as variables.

[4] https://solr.apache.org/
[5] https://angular.io/

Figure 24 – Reusing test methods by text similarity



Source: The author (2022)

A user can follow the simplest path to automate a test case: capture the actions and then save the test case locally. It is also possible to reuse the controller actions, since they were indexed and converted into test actions. However, instead of only saving locally, the user can generate new code that can be refactored by experienced developers and merged back into the codebase. Then, the codebase is indexed again, and the action is available for reuse. This tool provides a way for developers and testers to share test actions transparently. The data extracted from real-world use confirmed that, with just one-day training, novice testers automated more than what was expected. Also, the tool is being used to train new contractors since it has a shallow learning curve.

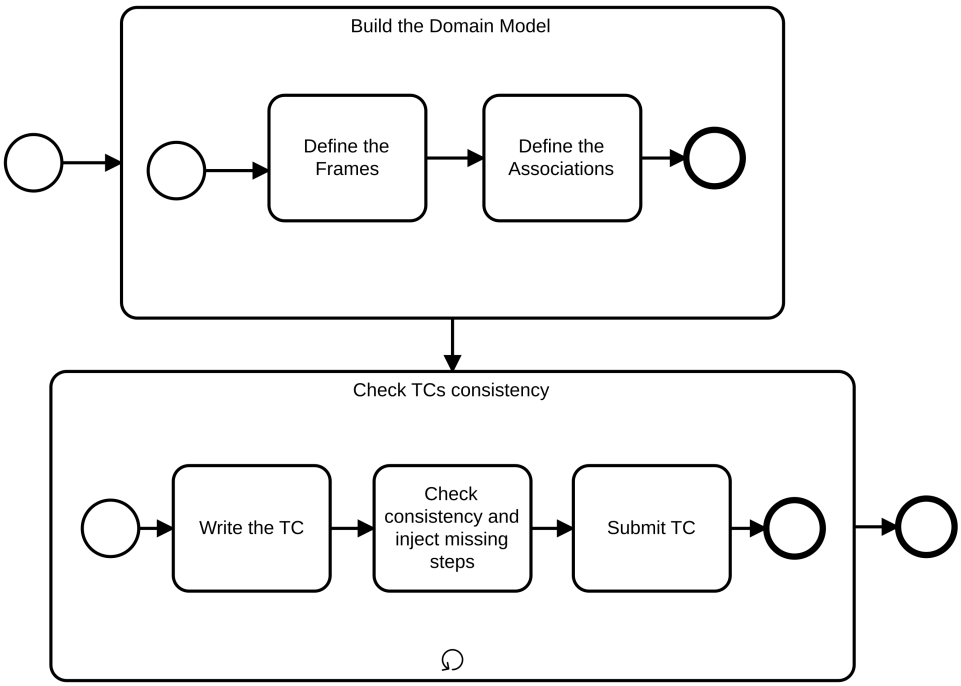
6.1.2.2 Kaki: CNL and Consistency Analysis and Solving

There is a more disciplined scenario where we can use TC written according to a CNL as input. It is a plausible scenario because we observed in practice that hundreds of TCs were manually rewritten to match a specific language format (to improve test automation). Based on this possibility and assuming a standardization of the test step descriptions, we implemented a tool that executes a consistency analysis, similar to what is detailed in Chapter 3, using Alloy or Clingo instead of Machine readable syntax for CSP (CSPm).

The tool we have developed to check the consistency of test cases (Kaki) has two main facets: (1) build the domain model; and (2) suggest a valid execution sequence that makes a test case consistent in an optimal order. Figure 25 presents an overview of the interaction with the tool from the user perspective. Each of the steps is detailed next.

To check consistency and suggest a valid sequence, a test engineer should first define

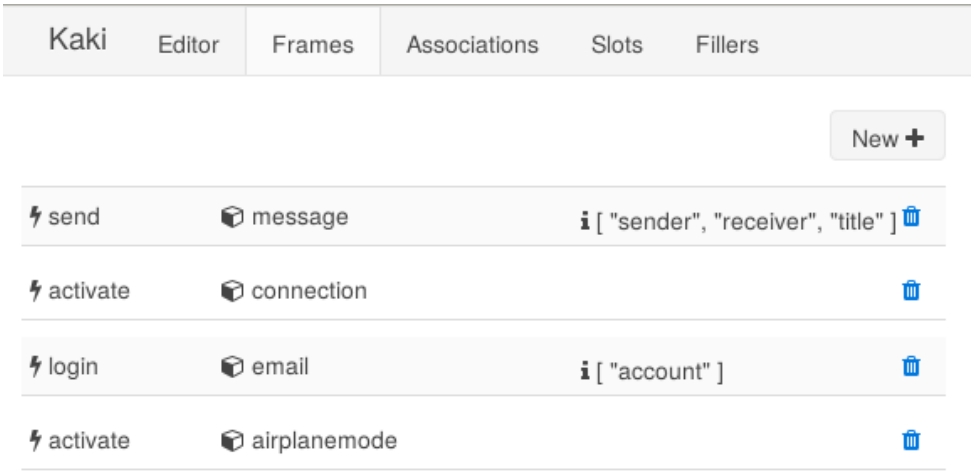
Figure 25 – Tool activities in the perspective of the user



Source: The author (2020)

the domain model. The valid frames, for the domain model, are dynamically defined via the tool interface. Figure 26 illustrates the definition of a frame by filling the operation, patient and the extras slots that together represent a valid and consistent frame.

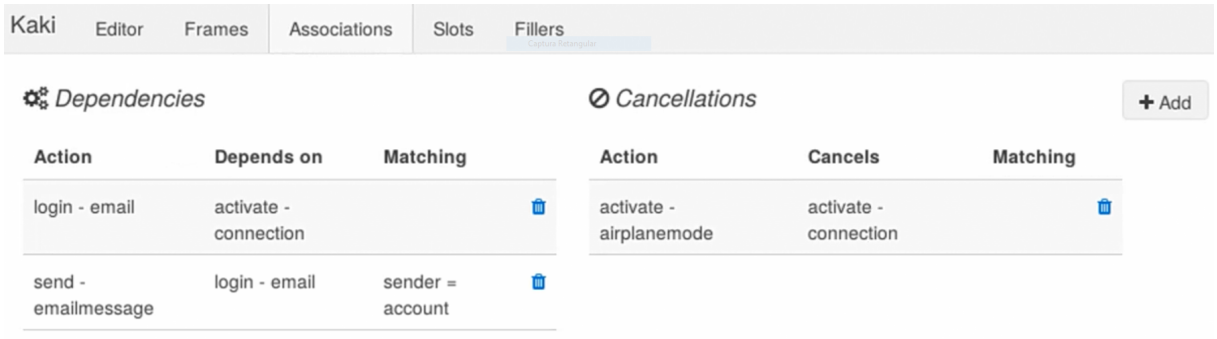
Figure 26 – Example of user-defined frames



Source: The author (2020)

Dependencies and cancellations are also defined via the user interface, as shown in Figure 27. The matching rules can also be set. For instance, to send an email message, the “sender” value must match the “account” logged in. Then, the Alloy code for the domain model can be dynamically generated on-the-fly from the user input.

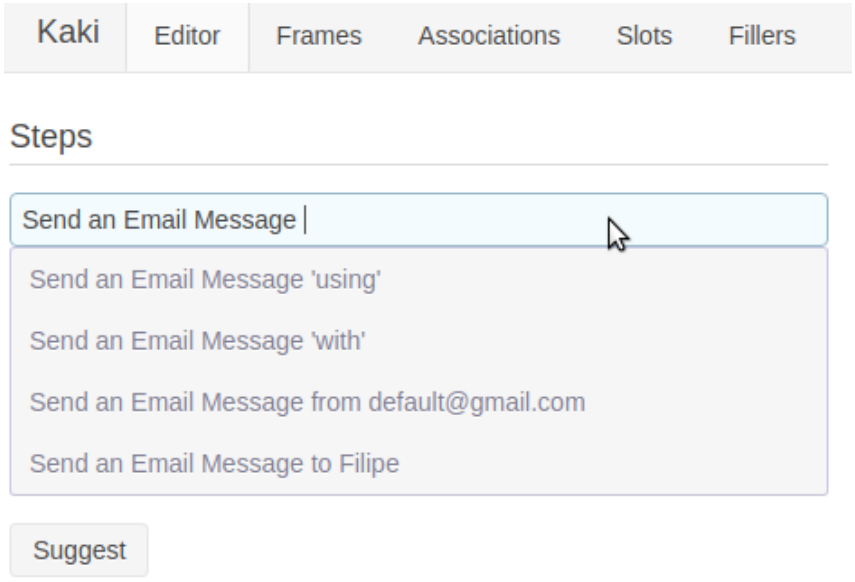
Figure 27 – Defining associations between frames via tool interface



Source: The author (2020)

Figure 28 shows how to enter the steps of a test case. The tool helps the user by auto-completing the sentence and, finally, marking the sentence green if it complies with the CNL.

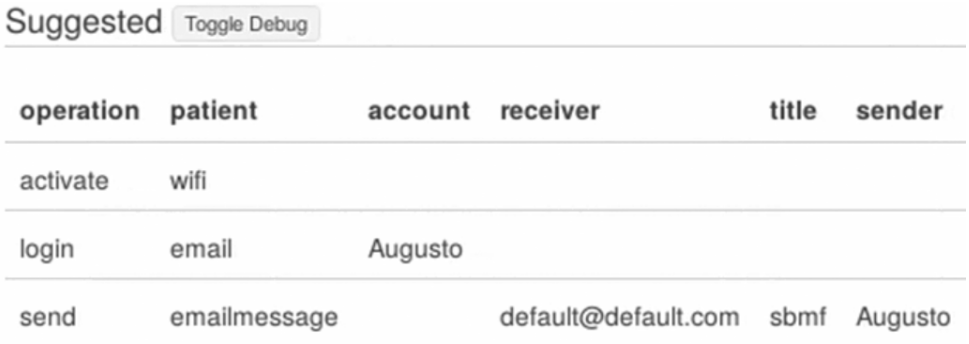
Figure 28 – Syntax suggestions



Source: The author (2020)

With test steps that comply with the CNL, the user can ask for a valid execution sequence. Figure 29 shows a valid sequence of frames that are required to send an email message from the “Augusto” account. It is worth mentioning, one can modify the slot values (in the sentences) and the tool adjusts the dependencies accordingly.

Figure 29 – Suggestions made after the dependency analysis



Source: The author (2020)

6.2 EVALUATIONS

6.2.1 *SmarTest*: From Requirements to Scripts

To quantitatively assess the feasibility of automating the generation and automation process by using our tool, we compared the scripts generated automatically from requirements *versus* scripts based on test cases created by hand from the same requirements.

Goal, Research Questions and Metrics

We structured this evaluation by using the GQM (Goal, Question, Metric) approach (CALDIERA; ROMBACH, 1994) as detailed next.

Goal Analyze our formal approach for the purpose of evaluating the test generation and automation with respect to improving the process from the viewpoint of the test engineers in the context of a direct-engineering testing process.

[**RQ1**] **Research Question** Can our approach generate test cases that encompass the ones created manually by specialists?

Metric Automation scripts matches, i.e., the percentage of method calls and arguments inferred from the same TCs that were identical (automatic *versus* manual generation).

[**RQ2**] **Research Question** How much effort can be saved when generating test cases from a feature using our strategy?

Metric 1 Size of the required text, i.e., the number of words/sentences needed to generate the same artifacts.

Metric 2 Number of changes needed to update the test cases.

RQ1 - Design and Execution

To achieve the evaluation objective and analyze the metric described before, three stages are required: (1) obtain the legacy data; (2) generate new artifacts; (3) apply the same automation strategy in both artifacts. First, we investigated which TCs would be appropriate candidates for this evaluation. Because there is already an initiative (within *Motorola, a Lenovo Company*) to make test steps clearer and ready for automation, we only considered these refactored TCs as the main criteria in our search, which left us with more than 200 TCs. From these, we only considered the ones that reference the feature they were based upon and, additionally, features that have more than 10 and less than 20 associated TCs to discard either too simple or too complex features. Using these filters, we queried 2 features (and the corresponding 24 TCs) from the global database and exported the results to a file in a compatible format for the automation strategy.

Then, we reverse engineered the relevant requirements and domain model based on each TC's summary and test steps. Each scenario became a single sentence in compliance with our CNL that defines a requirement. Each requirement was further described by detailed substeps in the domain model. It is worth mentioning that additional information, such as labels and identifiers, was also defined in the input document for our tool. Then, the text was processed, and the test cases were reported in a compatible format for the next evaluation step. It is worth mentioning that we generated the same number of TCs because the tool was configured to derive only trivial scenarios and ignore additional scenarios (see Section 4.1.5).

Finally, we applied the mechanized automation strategy (already being used internally) to create scripts from the input sentences in English. This strategy relies on matching an input sentence with a method already present in the company codebase for test scripts. The goal is not to compare the English texts directly but use the associated code, as shown in Listing 6.1, to "semantically" compare them.

Listing 6.1 – Inferred code from an English sentence

```
    # Step: go to home screen
2   main_device.launcher3.launcher.navigate()
```

RQ1 - Results and Discussion

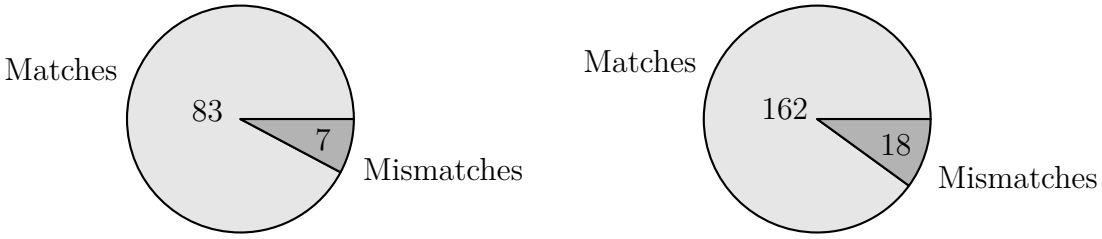
To compute the similarity of the code generated from both strategies, we used a diff mechanism similar to the one implemented by the git tool[6], where we compare two files line-by-line. In that case, we consider a match only if the lines from the two corresponding source codes are identical. That means that not only the method signatures must be the same, but also all arguments must match. Since the tool used to infer these method calls produces a source code with additional information such as comments and *import*

[6] https://git-scm.com/docs/git-difftool

statements, we had to post-process the output to ignore these lines. Only lines that contained methods were preserved. This was a fairly simple filter, since the tool always generate the source code using a single template.

The results shown in Figure 30 illustrate the distribution of method call matches — and mismatches — when comparing the output of the inference mechanism from both legacy test cases and the ones generated automatically by our tool. For instance, in Figure 30a, from 90 (ninety) inferred method calls, we only observed 7 (seven) mismatches. The next feature produced, as illustrated in Figure 30b, from 180 (one hundred and eighty) method calls, only 18 (eighteen) mismatches. These data exhibit a consolidated result of more than 90% (ninety percent) of matching precision.

Figure 30 – Mismatched method calls or arguments x Matching inferred method calls

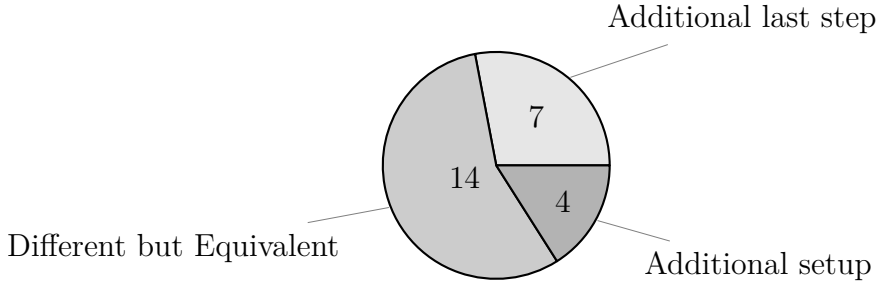


(a) Feature 6813 (b) Feature 7163

Source: The author (2022)

The mismatches causes are illustrated in Figure 31, in which we notice three causes:

Figure 31 – Mismatching causes



Source: The author (2022)

Additional last step This happens because the original TCs have different writing styles. In some cases, their last test step does not mandate verifying the postcondition — instead, the test case goal is inserted in the expected result of the previous step. Therefore, because our strategy always generates a "verify condition" as the last step by default, then we can have a mismatch (or more precisely a virtual additional step) in some cases.

Additional setup Similar to the previous cause, our strategy can sometimes generate an additional setup as the first test step. This arises from the fact that our tool generates,

by default, a shared setup for all TCs in the same feature. We assume that if a setup must be executed only in a particular scenario, then it must be part of the test steps themselves. That being said, this mismatch (or misplaced) setup can occur.

Different but equivalent Because the sentences of the legacy test cases may not always comply with our CNL, then it is reasonable that some sentences must be written differently. Because all machine-learning algorithms are prone to inaccurate classification, then it is natural that the inferred method calls from similar texts may differ. However, after further manual investigation, we noticed that despite the code mismatches, the sentences have the same semantics in English.

RQ1 - Threats to Validity

We discuss here the threats to the validity of the evaluation of RQ1.

Conclusion Validity The discussion claims a high precision match. Because the domain model and requirements, rewritten in compliance with CNL, were based on the legacy test cases, the generation was biased by this reverse-engineering. However, we do not claim to always generate identical tests. The goal is to assess whether the tool can generate the same TCs, i.e., has the same expressiveness. Then, it is reasonable to say that this high precision match translates into equivalent expressiveness. Furthermore, it may be relevant to compare two third-parties generating test cases (each with a different strategy) simultaneously, with none having previous knowledge about the feature.

The absence of a statistical test is due to the 100% semantics matching after reviewing the syntactic mismatch causes (outliers) in the data points. There is no fundamental change (as expected) in the results of the proposed automatic approach when compared to the traditional approach.

Internal Validity Within *Motorola, a Lenovo Company* there are many teams with different inputs, strategies, writing styles, and contexts. We have chosen only a subset of features in which their corresponding test cases were refactored. However, since all teams use the same management tool and, despite having different inputs, one can always rewrite the artifacts in compliance with our CNL, then it may be representative enough. Furthermore, we plan to execute more quantitative experiments with other teams, besides the already positive feedback we have received.

External Validity We assumed an industrial scenario in which functional requirements are described in reference documents, and test cases are generated from them based on a well-established template. It does not represent all use cases in the industry, especially the ones that do not have specifications written in natural language. However, since *Motorola, a Lenovo Company* is a global reference in its market, then this result should apply to similar scenarios.

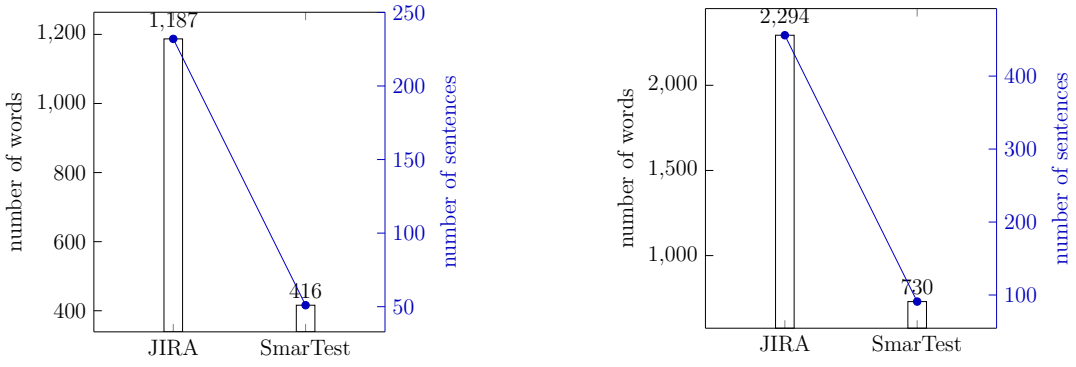
RQ2 - Design and Execution

To assess the potential effort minimization of our approach, we used as evidence the size of the text (Metric 1) handled as input to generate the same test cases (from the same features seen in RQ1) in comparison to the traditional approach of writing every test case by hand. As 'text', for the traditional approach, we consider every word present in the "Step", "Summary", and "Expected Results" columns of the respective TCs. To make a fair comparison, in our approach, we considered the words present both in the requirements and the corresponding domain model. We do not consider the original feature document in the word count of the traditional approach since it is only a reference document. Regarding the number of sentences: in the traditional approach, each "Step", "Summary", and "Expected Results" is counted as one sentence; while in our approach we consider each requirement and entry in the domain model as a sentence. Regarding the number of changes needed to update the test cases (Metric 2), we tracked a real update made by a test engineer directly on the affected TCs.

RQ2 - Results and Discussion

As seen in Figure 32, there is a drastic text size reduction after adopting our strategy when we compare the number of words and the number of sentences to create the same test cases. The reduction proportion was similar in both features analyzed, demonstrating that it may not be an isolated case. It is worth noting that the reduction of the number of sentences accompanied the reduction of words, which demonstrates that we create sentences with the same size (or word count) as the testers are used to create by hand.

Figure 32 – Text size reduction



(a) Feature 6813

(b) Feature 7163

Source: The author (2022)

Concerning Metric 2, the test engineer made the change in a specific step that was shared by all test cases from Feature 7163. Since we modeled this test step in our domain model as a substep (i.e., detail of an abstract action), any change to this substep would update all references to it accordingly. That being said, while this specific change needed

16 (sixteen) separate updates in the traditional approach, it would require only 1 (one) update by using our solution.

RQ2 - Threats to Validity

We discuss here the threats to the validity of the evaluation of RQ2.

Conclusion Validity The discussion claims that the proposed approach minimizes the effort for generating and especially maintaining test cases. For this claim, we considered text size as a proxy measure. Of course, when one reads an artifact to understand or modify its text, it is always better to have a smaller and abstract piece of text (with further details provided when needed). However, the effort to write the requirements in compliance with our CNL (and not simply writing in freestyle natural language) was not quantified. The intuition is that, despite the theoretical drawback of a constrained specification, the long-term advantages outweigh this initial inconvenience.

Internal Validity In fact, all evaluations (quantified or not) demonstrated a considerable reduction of text size and changes needed to update the test cases. It may not present such a drastic reduction in particular cases, in which only a single and simple scenario should be tested.

External Validity As discussed in RQ1, it may not represent all use cases in the industry, especially the ones that do not have specifications written in natural language. However, since *Motorola, a Lenovo Company* is a global reference in its market, then this result should apply to similar scenarios.

6.2.2 *SmarTest*: Additional Contexts

The quantitative evaluation tackled features from a single team and similar product functionalities. This threat to validity was also discussed in detail in the previous section. To mitigate this situation, we promoted the tool to teams across different company sites and projects to analyze whether the tool is applicable in other contexts.

[**RQ3**] **Research Question** Is our approach applicable to other teams and projects?

Metric Number of features/inputs that could not be covered

RQ3 - Design and Execution

We had to survey managers that have experience in test generation to suggest focal points that could use the tool and provide feedback. The tool was used to write features and generate test cases in English for 3 (three) Motorola sites in Brazil. To diversify the input types, we also considered the ones that have never been analyzed before, including:

manuals, slides, and even bug reports. Each focal point suggested a representative feature to be rewritten and analyzed using the tool.

RQ3 - Results and Discussion

After initial meetings with the focal points, they presented the chosen requirements to be rewritten while they provided feedback. We analyzed three features (A, B, and C) and generated their respective test cases.

Feature A was described in presentation slides, from which we extracted and rewritten some sentences that described the relevant scenarios (guided by the focal point). Then, step details were incrementally added to reduce ambiguity. According to the focal point, the generated test cases by the tool were sufficiently similar to the legacy test cases.

Feature B had an associated feature document from which we could extract the requirements. Without even writing the domain model, the focal point mentioned that the tool covered the same test scenarios and even ones that he did not think of before. The final feedback was that this was already useful enough for him to use the tool.

Feature C had as input a comprehensive manual specifying every functionality to be tested. This feature, in particular, had no prior associated TCs, so the ones generated by the tool would be used in production, in case they were approved. We built the domain model from the details that the focal point knew by experience on how to execute some abstract steps. Then, after adding information such as labels and custom fields, the generated tests were analyzed and finally approved to be used in production without further human intervention.

In this light, we noticed that all features could be analyzed by the tool without significant changes in the tool. We assume, after all feedbacks, that the tool could be used in any team or project within the scope of *Motorola, a Lenovo Company*.

RQ3 - Threats to Validity

Conclusion Validity The assumption that the tool could be used by any team is based on the the fact that any requirement could be rewritten in compliance with our CNL. We failed to determine if technical specs that require, for instance, time or arithmetics could be properly specified and tested.

Internal Validity As in RQ2, there is a wide range of teams across different sites. Even though we analyzed more teams, it is still a small proportion.

External Validity We have only considered black-box testing. That being said, writing requirements in natural language may not be enough for every use case.

6.2.3 Kaki: Scalability Evaluation

As discussed in Chapter 5 and Section 6.1.2, we also deal with a scenario in which test cases are already generated with no standard whatsoever. That being said, a mechanism to allow a consistency analysis without the requirements was developed. The consistency of a single test step can be checked by only analyzing whether it fits in any frame (syntax parsing). Therefore, the main concern regarding performance is the response time of the semantic mechanism to check the consistency of a sequence of frames considering their associations. The response time is particularly relevant because the test engineer should check the TC consistency on-demand while she is submitting it to the test management tool.

Goal, Research Questions and Metrics

Goal Analyze the Kaki tool for the purpose of evaluating the consistency analysis mechanism with respect to its performance from the viewpoint of the test engineer in the context of dependency analysis of real TCs.

[RQ4] Research Question Does our approach provide an acceptable feedback delay when checking the consistency and dependencies of a test case?

Metric Response time in seconds, i.e., the delay between the test steps input and the tool response for the consistency and dependency checking.

RQ4 - Design and Execution

To accomplish the evaluation objective, three stages are required: (1) Extract and prepare information to build the chosen domain model; (2) Obtain the test cases that will be checked against consistency issues; and (3) Perform the consistency check for each test case and gather the result and the time elapsed.

First, structured information describing paths among mobile applications was extracted from a Motorola code repository annotated by experts. These artifacts can be categorized into components, which can be used to identify applications or distinctive features. Then, after data selection (chosen components), cleaning, and integration (among components), a directed graph was inferred from which we derived a domain model.

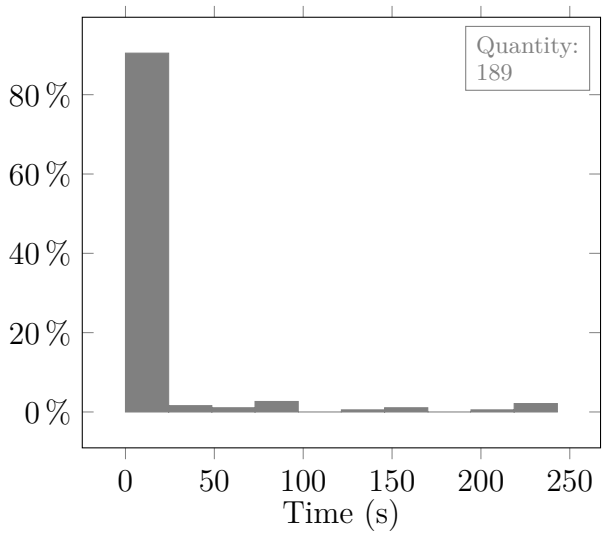
More than 200 (two hundred) real TCs from 4 (four) different components were available for the evaluation, and all of them were UI/functional tests. These TCs also have associated execution scripts, so we could extract exactly which actions were coded to perform the TC. This way, the (screen) paths among these UI actions were to be discovered after our dependency check/injection mechanism. In this evaluation, these paths describe how to reach the screen whose actions must be executed upon.

To automate the execution of the evaluation, we developed a script that, for each test case: (a) selects a subgraph from the domain model baseline, considering only the features pertinent for the particular TC; (b) executes our consistency mechanism over the test steps to inject the missing dependencies when applied; and (c) collects and stores the results into different data formats (text logs, csv, etc.).

RQ4 - Results and Discussion

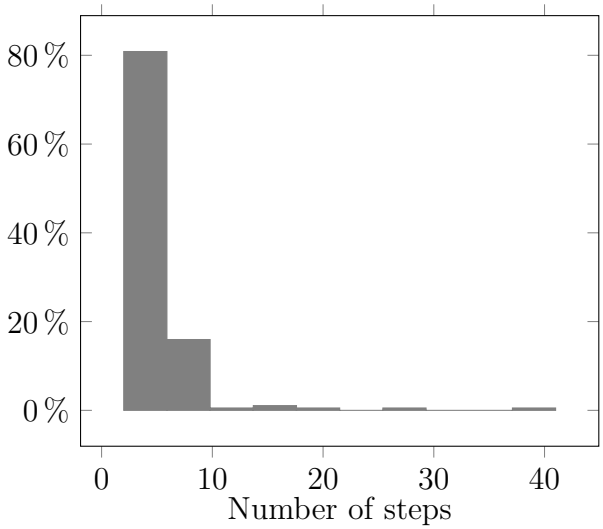
The results automatically gathered by the evaluation script are summarized in Figure 33 and Figure 34. In these figures, two fundamental vectors are shown: the time spent to check the consistency and inject the missing dependencies, and the corresponding number of steps.

Figure 33 – Histogram: Time spent



Source: The author (2022)

Figure 34 – Histogram: Steps



Source: The author (2022)

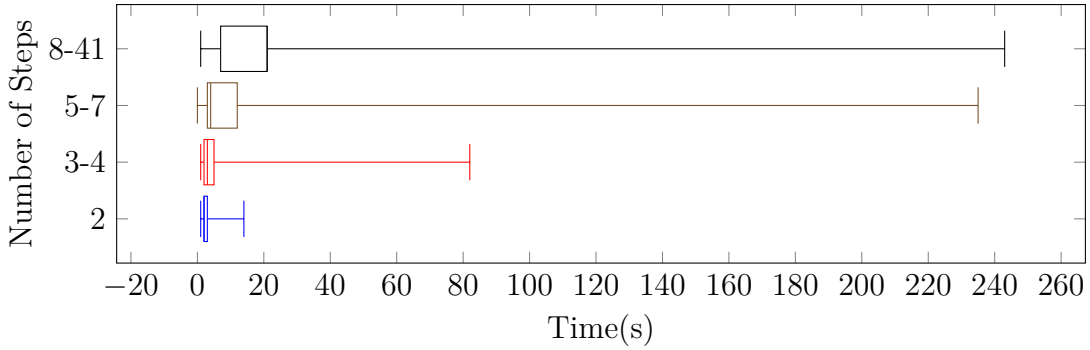
As can be seen in Figure 33, for the vast majority of TCs evaluated, the consistency analysis took less than 10 (ten) seconds to give the best global solution. There are also outliers, taking more than 200 (two hundred) seconds to respond. Also, observing Figure 34, there are some test cases with more than 40 (forty) steps. These unlikely scenarios contributed to raising the time spent with the analysis of the outliers. It is also worth mentioning that 18% of the test case analyses timed out (300 seconds limit). This drawback was already expected from a tool that relies upon model finding. However, since the analysis finds optimal solutions to problems that include generalization/data dependency resolution (not quite well handled by other algorithms), we consider this timeout ratio acceptable.

Figure 35 shows an overview of the relationship between the number of steps and the time spent as a box plot. It displays the spread and skewness of the analysis time for each group/class through their quartiles. Although classes have, usually and preferably, fixed width, we chose this range of values to avoid an unbalanced frequency distribution, as illustrated in Figure 34. From this box plot, we can deduce that, for most cases, the analysis time increases proportionally to the number of steps (steadily). However, for each group, there are outliers with significantly higher time duration. By sampling analysis, we correlate these outliers to the size of the corresponding domain model (too many dependencies or large hierarchy trees).

Concerning the environment configuration to carry out the evaluation, Table 5 details hardware specifications and software configurations.

Considering the results shown, it becomes easy to perceive that the tool can provide fast feedback regarding dependencies when considering each TC at a time. We argue that it is unlikely for a given frame or test case to have so many dependencies that it becomes infeasible to make a bounded verification in Alloy. For larger scopes (bounds), the process

Figure 35 – Box plot: Time x Number of Steps



Source: The author (2022)

Table 5 – Environment specifications

Environment	**Spec**	**Value**
Hardware	Memory	8003 MiB
	Processor	Intel(R) Core(TM) i5-6600K
Software	Alloy Version	4.2
	SAT Solver	MiniSAT

Source: The author (2022)

becomes too expensive (timewise), as seen in the results for the timeouts. However, since we found out that test cases have — on average — 4 (four) test steps in our scenario, it is acceptable to analyze and make consistent not only one by one but even multiple test cases handles at once.

Threats to Validity

We discuss here the threats to the validity of our evaluation.

Conclusion Validity The discussion in the evaluation claims that the response time is enough since it is less than 300 seconds in extreme cases, and even less in most cases, considering an intermediary processor. However, a qualitative study to find the test engineer perception of this delay, in a wider context, was not performed.

Internal Validity There are two variants that may influence different results in subsequent evaluations: (1) the test cases used in this round might not be representative, since we had limited access to automated scripts in a given set of components/features; (2) the hardware used have reasonable processing specifications, but they might not be enough for multiple users at real time, which could demand a dedicated high-end server or a different architecture.

External Validity The number of test steps could also vary for different applications, platforms, and companies — which could negatively affect the response time. However,

because an action supports any abstraction level (test steps, test cases or even entire suits), it is possible to apply a strategy to consolidate a set of actions in a more abstract one. In this way, even if the test case has a high number of test steps, we can reduce it by building one or more abstraction layers. In other words, instead of having a *flat* sequence, we can transform it into a *nested* sequence by using the test action structure and analyze each associated frame at a time.

Scalable Implementation

We presented our semantic prototype implementation in Alloy, in which the semantics of a domain model, and all its elements, are formally defined; Alloy was used to rigorously perform the consistency analysis. However, even with the performance improvements, it is yet not ideal for industrial use because of its runtime overhead.

Due to this performance bottleneck that may prevent the adoption of our strategy in practice, we decided to make an alternative implementation based on the same requirements and definitions. Still, we chose declarative programming because it is more adequate for the purpose of our work. Answer Set Programming (ASP) is a particular form of declarative programming that uses non monotonic reasoning over knowledge representations. The syntax is somewhat similar to Prolog, but with different techniques that allow fast SAT solvers Lifschitz (2019). One mature ASP system is called *clingo* that comprises a grounder *gringo* and a solver *clasp* Gebser et al. (2014).

Although Alloy allows more abstract modelling and is easier to understand, *clingo* is faster because of its stable model semantics but is harder to encode complex problems. That is why we chose to make this alternative implementation just for the sake of performance; a formal translation from Alloy to clingo is out of the scope of our work.

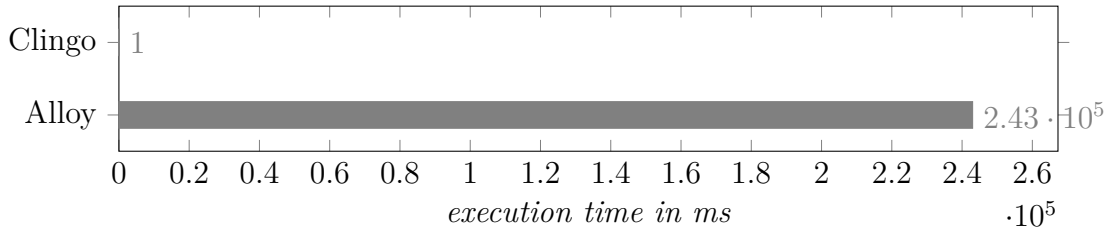
The Listing 6.2 illustrates a simplified way to encode our dependency analysis. First, in Line 1, we declare all possible actions via the predicate *action*/1 (arity 1). We use the character ';' only as a syntactic sugar to *action*(*wifi*)., *action*(*send*)., and so on. Then, we define which actions should be performed *before* others (dependency) and also dictate action hierarchy. These facts are denoted by using the predicates *before*/2 and *sub*/2, respectively, in lines 2–5. We also define, in Line 6, the *root* action which should be performed first. In Lines 7–10, we define the inference rules to discover a consistent sequence of actions. These rules are encoded using the predicate *next*/2 and variables (uppercase letters) to define which step transitions are allowed. Finally, in Line 11, we restrict the output to show only the relevant predicate.

Listing 6.2 – Dependency analysis using clingo

```
action(wifi;g;send;login;press;swipe;conn;root).
before(send, conn).
before(send, login).
before(login, conn).
sub(conn, g); sub(conn, wifi).
next(root, send).
next(root, X) : action(X) :- next(X, _).
next(X, Z) :- next(X, Y), next(Y, Z).
next(Y, X) :- before(X, Y), not sub(Y, _).
next(C, A) :- before(A, B), sub(B, C).
#show next/2.
```

Considering the same TCs, our strategy using *clingo* achieved 1ms of max execution time, even in the same cases in which the Alloy implementation times out, as illustrated in Figure 36.

Figure 36 – Max execution time: Alloy *vs* Clingo implementation



Source: The author (2022)

7 RELATED WORK

7.1 TEST GENERATION FROM NATURAL LANGUAGE DESCRIPTIONS

As previously discussed, imposing on users the adoption of unfamiliar notations is often time-consuming and inefficient. This statement holds for testers that do not have any programming background and need to create or maintain code scripts, as well as for developers that commonly have little or no experience with formal methods to build and maintain an abstract model model of the SUT expected behaviour. This scenario has motivated the use, by several companies, of test artifacts written in natural language. However, because of NL ambiguous nature, it is hard to systematically verify consistency and even harder to define a mapping between the descriptions and the corresponding automation framework scripts without human intervention.

The search for an optimal mapping between NL descriptions and executable tests has been an active research area. Cucumber (WYNNE; HELLESOY, 2012), for instance, assists users on writing acceptance tests in a behavior-driven development environment: parameterized scenarios are written in natural language and semi-automatically mapped to a source code or stub, in order to provide a better tracking. Each scenario is a list of steps that must follow some basic syntactic rules defined by a grammar called Gherkin.

DASE (WONG et al., 2015), Document-Assisted Symbolic Execution, is an approach that uses NL processing and heuristics to inspect the code's textual documentation and automatically extracts input constraints to improve symbolic execution for test generation. The textual documents used as input are manual pages and code comments through which the tool identifies grammar relationships that define input constraints, enabling DASE to perform a symbolic execution.

TaRGeT (FERREIRA et al., 2010) relies on the concept of *user view templates*, which capture user interactions with the system through sentences used to describe user actions, system states and system responses. This template is the input to the tool (MACHADO; SAMPAIO, 2010; FERREIRA et al., 2010), whose purpose is to mechanize a test case generation strategy. The tool generates test cases (for manual execution) which include the test procedure, description, and related use cases. Moreover, the tool can generate a traceability matrix relating test cases, use cases and requirements. There is also an extension of the tool that adopts an CNL (ucsCNL) (BARROS et al., 2011) to reduce ambiguity, but it only provides syntactic assistance, which does not influence the inferred semantics model.

UMTG (WANG et al., 2015) also deals with the generation of TCs from use cases. The overall approach is similar to TaRGeT (NOGUEIRA; SAMPAIO; MOTA, 2014), however UMTG adopts a different natural language parsing strategy, extracts the data from a particular use case format (RUCM), and employs a solver for OCL (Object Constraint

Language) constraints manually written in order to enable TC and input generation.

Following a similar approach, NAT2TEST (CARVALHO et al., 2014) generates TCs from textual descriptions, but the inputs are system requirements rather than use cases. To minimize NL ambiguity, requirements are written according to a specific controlled natural language named *SysReq-CNL*. The syntactic requirements are mapped into semantic representations, so that specifications in several target formalisms can be derived, from which test cases are generated. NAT2TEST focuses on reactive systems, having been successfully used in the Avionics and Automotive application domains. A test case in this context is a vector with values for input and output variables, and associated time spans, rather than a sequence of test steps, as in the previous approaches.

Still regarding TC generation from requirements, the RTCM (YUE; ALI; ZHANG, 2015) framework uses the RUCM use case format and OCL constraints, as in UMTG (WANG et al., 2015). However, RTCM goes a step further and generates executable scripts (via aToucan4Test (YUE; ALI; ZHANG, 2015)) by embedding function calls into the specifications. A more detailed comparison is presented in Section 7.3.

7.2 TEST AUTOMATION

7.2.1 Ad-hoc Test Automation

The simplest, *ad hoc*, form to execute Graphical User Interface (GUI)-based test cases is to interpret the description and manually interact with the system to perform the corresponding actions. However, to reduce costs and meet deadlines, companies seek to automate the execution of these test cases by converting them into test scripts. Different approaches are described and analyzed next. Due to the vast literature on test automation, however, we focus here on approaches that are closely related to our work and specifically to the Android platform.

7.2.1.1 Coding

Since executable test scripts are simply code snippets, a straightforward approach is to count on developers to interpret the tests descriptions and code them using an automation framework.

As an example, we highlight UIAutomator[1], which is a framework developed by Google to allow test automation through coding (see Listing 7.1) based on atomic actions such as: click on a button with a given text; swipe operations; rotate screen; and check if a given element exists. UIAutomator is a Java library with APIs to create customized functional UI tests for Android, and it has an engine to automate and run the tests (ANDROID, 2015). Besides being officially supported, one of its main advantages is the available functionality

[1] http://developer.android.com/intl/en-us/tools/testing-support-library

Listing 7.1 – UIAutomator code example – Opening the Email application by pressing its launcher button

```
[...]
UiObject2 emailBtn =
    device.findObject(By.description("Email"))
emailBtn.click()
```

to test multiple user/system apps and their interactions without even having their source codes, since it is UI-focused.

As another example, Selendroid[2] is a test automation framework for Android native and hybrid applications. It uses the Android instrumentation framework and tests one app at a time. Because tests are written using the Selenium Webdriver client API, it may be fully integrated to existing Selenium frameworks.

Despite the facilities provided, the use of these frameworks demands specialized knowledge from the developer. Moreover, the automated TCs are framework dependent and may become outdated when they use features that have been discontinued/deprecated after a framework or system update, requiring significant code refactoring.

7.2.1.2 Capture & Replay

Automation based on the C&R approach, on the other hand, does not require prior knowledge of an automation framework, and tends to be comparatively faster (LEOTTA et al., 2013a). However, the created test scripts are typically linear, in the sense that they do not embody alternative paths, serving only the purpose of pure playback.

RERAN (GOMEZ et al., 2013) is a C&R tool that focuses on the accurate reproduction of gestures and actions performed in Android smartphones. Challenges such as complex gestures, precise timing, and input from multiple sensors are dealt with by listening and parsing low-level events from the stream. They are accurately reproduced by sending the same event inputs with the original delay from each other, as they were captured.

Robotium Recorder (ROBOTIUM, 2013) is also a C&R tool, which transforms user interactions into code scripts compatible with the Robotium automation framework. The scripts can be later executed, refactored or changed by developers.

There are also hybrid approaches that mix C&R with a strategy to capture keywords from the GUI to compose test actions. The test actions are stored as a script to be reproduced later. MonkeyTalk (CLOUDMONKEY, 2013) and Robotium Recorder™ (ROBOTIUM, 2013) are examples of frameworks that support C&R of keywords from applications built on Android or IOS platforms. Although they are well-consolidated commercial tools, they do not link test actions with TC descriptions (which is an important link we explore to ease the process of designing and reusing TCs).

[2] http://selendroid.io/ - Accessed 06/29/2017

Our work was also inspired by an empirical analysis presented in (LEOTTA et al., 2013a), which shows that a test suite development requires more time when testing approaches based on coding are adopted (between 32% and 112%) compared to C&R approaches. On the other hand, the benefits of coding (concerning time saved) increase over successive releases, because code is easier to maintain by specialists. Although these results were obtained in a different context (Web), we observed that they are similar or even more prominent in the mobile context. The absence of a reuse strategy for C&R artifacts is expected to be directly related to high maintenance effort, while coded tests benefit from design patterns such as Page Objects, as noted in (LEOTTA et al., 2013a). In our approach, we combine the advantages of both approaches, by conceiving an elaborate reuse strategy for C&R based on test actions expressed in natural language.

7.2.2 Script Generation

In the literature, we can identify several strategies which generate scripts from specifications. These specifications, however, are often detailed at a similar abstraction level to the IUT, and use formal notations. Next, we discuss some of these related strategies.

Copstein and Oliveira (2005), for instance, proposes a strategy to automate script design for statistical testing. The authors use a stochastic automata network (SAN) for the specification and an Interface Event-State Model as an intermediate model (which maps abstract models into implementation elements). Although an intermediate model is used, the strategy uses a notation that requires specialized training and does not allow multiple or composite model details. Besides, the generation is ad-hoc and, thus, there is no established conformance relation to allow performing soundness or consistency analysis.

Balcer, Hasling and Ostrand (1989) creates a language (TSL) for writing formal test specifications. In TSL, environment conditions, parameters and result verifications can be encoded. In our work, however, because dependencies and instantiations are already encoded within the domain model, generated test cases are directly mapped into scripts without additional intermediate representations, while still being framework-agnostic.

Silveira et al. (2011) proposes a strategy for generating scripts based on UML (Unified Modeling Language) models. The general idea is to annotate the model with custom stereotypes and tags. Then, to automate the test execution, these tagged stereotypes have a direct mapping to test code. This work, besides using a specialized notation to derive tests from, requires adding execution details to the abstract specification.

A recent work (LI, 2022) leverages pre-trained Natural Language Processing (NLP) models to generate test scripts by matching natural language descriptions with test methods that locate GUI elements. Because a pre-trained NLP model is applied, there is a considerable error rate while matching, even though only simple and atomic descriptions are used. In our context, using a CNL for requirements and domain model allows us

to create accurate matches, define abstract actions on-the-fly, and ensure soundness and consistency because of the formal semantics.

7.3 FINAL REMARKS

Although the related work presented here share some features with ours, particularly considering that the inputs are textual documents in natural language, they do have some fundamental differences that are summarized Table 6.

Table 6 – Related work: comparison

	Smartest + ForceIDE + Kaki	NAT2TEST	TaRGeT	UTMG	RTCM	Cucumber
CNL	Y	Y	(y)			
Free-Style NL	Y		Y	(y)	(y)	(y)
Formal Semantics	Y	Y	Y	Y	Y	
Flexible Abstraction Levels	Y					
Reuse	Y					
Test Automation (Coding or C&R)	Y			(y)	(y)	Y
Consistency Analysis	Y					
Test Input Data		Y	Y	Y	(y)	
Time		Y				
Parametrized by Generation Mechanism		Y				

Y - Supported / (y) - Partially Supported

Source: The author (2022)

In general, the related work suffer from one or more of the following limitations:

- Cucumber, NAT2TEST, UMTG, and RTCM restrict the input to a given subset of natural language that must obey a particular standard and do not present alternatives to deal with legacy/freestyle NL inputs. In our approach, we consider both the case of freestyle and a CNL textual document as input.
- DASE, NAT2TEST, TaRGeT, UMTG, and RTCM provide no reuse of specification/test artifacts. Cucumber allows a simple form of reuse when the step is shared among test scenarios. Our work, instead, addresses reuse to a much larger extent. For instance, in our approach, a test case can be a step of a more elaborate test case, whereas a scenario is not qualified as a possible step for reuse in Cucumber.
- The aforementioned strategies that generate test scripts adopt specifications that either match the implementation abstraction level or annotate the abstract model with low-level details. Because our strategy proposes a compositional domain model, details can be further added according to the organization's roles and processes. Besides, because the semantics are formally defined to the atomic level, generated test scripts are verified to be sound and consistent.
- Most of the above-mentioned approaches focus on generation instead of automation of existing test cases. These approaches rely on formal and well-documented

requirements from which they can generate TCs. Unfortunately, up-to-date requirements are seldom available in the non-critical software industry. In addition, to allow a fully mechanized generation of automated test cases, the requirements need to be specified at a lower level of abstraction. None of these approaches provide a (bottom-up) alternative to building a domain model from manually generated TCs and allow consistency checks as we propose here.

- None of the cited approaches address test step sequence consistency and dependency notions, with an associated verification mechanism; this is a distinguishing contribution of our approach.

Despite the large scope of our work, we do not support some elements featured in other tools/strategies. For instance, even though we support passing data though parameters, we do not generate test input data. We also do not model timed-based behaviors. Finally, the generation mechanism is tied to the cspio relation and cannot be instantiated in other formalisms as easily as in NAT2TEST, which has an intermediate and hidden formalism.

8 CONCLUSIONS

In this work, we present a strategy to generate and automate test cases from requirements written in natural language. Through some immersion activities in the context of our industrial partner, Motorola Mobility—a Lenovo company, we were able to identify and deal with several issues that come with testing software in large scale with no formal requirements. However, based on previous experience, imposing the adoption of a formal notation or a rigid process was infeasible. In this light, our main contribution is to allow testing teams, that are also involved in a similar industrial perspective, to generate sound and consistent test cases automatically while still writing specifications in natural language (whose meaning is automatically obtained using an underlying, and hidden, formal semantics in CSP). Additionally, with the help of a dynamically evolving domain model, the test teams are not compelled to specify the entire functionality in a single take, allowing them to postpone more concrete descriptions for other roles down the line or when the feature is mature enough to be tested. Evidence from the experiments showed that our strategy has more than 90% precision (same test scripts were generated) and 80% text size reduction when compared with the artifacts generated manually.

Regarding legacy test cases, we also implemented a strategy to deal with steps written in freestyle natural language, which have no requirement to rely upon. For this scenario, our strategy translates a derived domain model into Alloy micro-models to allow the verification of consistency. Even with the Alloy bounded verification, however, we observed that testers wanted a faster response. Hence, we have implemented another consistency analysis mechanism using Clingo, which turned out to be much faster. While most analyses using Alloy took approximately 2-10 seconds, the Clingo alternative spent 1ms for the same analyses. The proposed approach to the direct engineering of test script generation from requirements written in a CNL, combined with the strategy to automatically generate scripts for legacy test cases, has proved promising in the particular industrial partnership with Motorola Mobility, but seems also useful to support mechanized test case generation in other contexts, as the issues and improvement opportunities are similar.

In summary, the scope of this work encompasses requirement and domain model specifications, and the generation of test scripts, including the necessary artifacts in between. Specification, verification, and generation are supported by custom tools tailored for this strategy. Because the semantics are formally defined and mechanically verified by custom tools, we ensure two important properties automatically: soundness and consistency. While the former property is well-known and regarded in the literature for test generation, the latter is a major contribution of our work, which ensures that the generated test cases can be executed on a concrete implementation. This consistency notion is derived from the information present in the domain model (dependencies, cancellations, details, etc.)

while the generation mechanism ensures that these relations do not change the behavior specified by the requirements. Another major contribution is the definition of a CNL – flexible enough for describing both requirements and the corresponding test cases – and an implementation that allows extending the CNL itself while writing the specification. These contributions were tested against an industrial context through experiments and evaluations, which demonstrated the practical application of our approach. In addition to generating and automating test cases that are equivalent to the ones described by hand, we also discovered new scenarios and created, from new features, test cases approved to be used in production. Although the strategy is evaluated within an industrial context, we can argue that, in principle, it is suitable for any functional, GUI-based, black-box testing automation process in which the CNL can be applied or adapted.

It is worth mentioning that the main issues on traditional software testing automation mentioned in Chapter 1 were all addressed by using our approach. The controlled natural language is expressive enough to allow references to other entities (using frame equivalence) for further description (Abstraction gap and traceability); to express all kinds of artifacts (Heterogeneous notations); to use mature mechanisms for automatic translation (Internationalization); and to be easily interpreted by humans (Legibility). Additionally, the definition of a domain model allows continuous addition of concrete details to abstract requirements while maintaining traceability (Duplicate artifacts), verifying consistency (Inconsistency), and avoiding duplicate mappings to concrete actions (Abstraction gap and traceability), which can also reduce the maintenance effort. The adoption of test actions as an intermediate structure allows for the representation of any test artifact with a single notation (Abstraction gap and traceability) as well as being framework agnostic (Technology lock-in). Finally, with the help of a domain model, the generation mechanism can create tests that leverage the system state from previous steps, avoiding repeating unnecessary actions (Execution time).

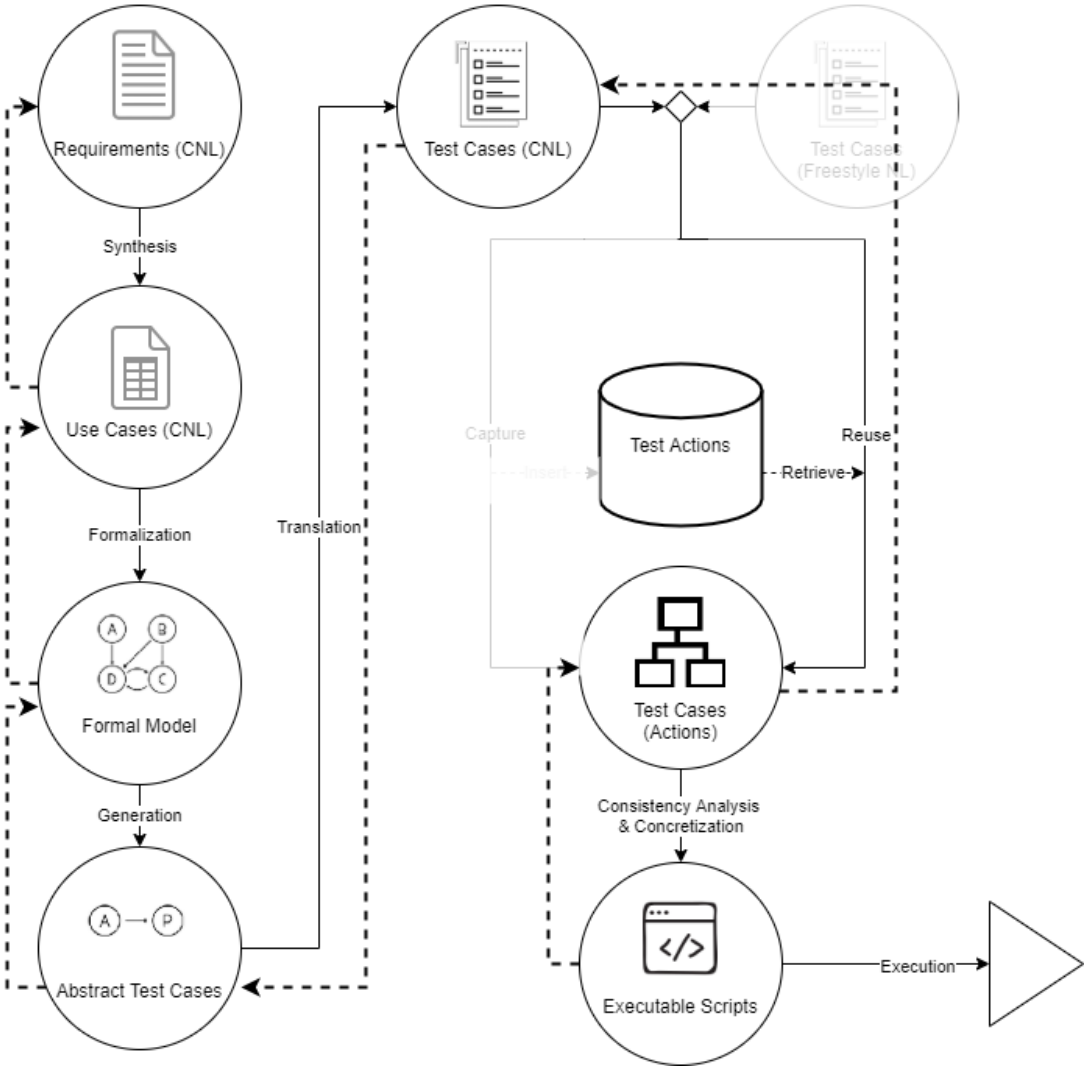
8.1 ONGOING AND FUTURE WORK

It is not uncommon to find teams that do not follow the traditional specification phase and skip it until they need to write test cases. Because of that, there are no requirements to rely on, blocking test design teams from using established strategies that generate test cases while ensuring soundness. In this situation, the only relevant artifacts are the existing test cases or scripts. So a possible alternative is to apply a reverse engineering process: the existing scripts can be used to abstract test case descriptions in a CNL, from which use cases can also be derived. Figure 37 illustrates a possible reverse engineering process from executable test scripts. The overall idea is to convert them into test actions, and by extracting associated metadata (method name and documentation), generate high-level descriptions that match our CNL. Then, the next step is to abstract each set of TCs into use cases and, finally, into high-level requirements. Considering the newly extracted

requirements, we could use direct engineering strategy (also captured in Figure 37) and presented in this work, to generate new test cases and, finally, automate them into scripts.

In general terms, the teams describe the tests directly in an executable manner (scripts) or step-by-step in natural language (test cases). For both situations, we can associate a sentence in compliance with our CNL to the artifact. When all artifacts are annotated we can build abstractions over it (reverse engineering) and take advantage of the already established techniques to generate and automate more test cases.

Figure 37 – Round-trip engineering: Integrated Framework



Source: The author (2020)

Unified Semantics

Because our strategy encompasses two scenarios (direct engineering and legacy test cases) with different inputs and challenges, we chose distinct languages and paradigms to formally define the semantics for soundness and consistency analysis. Although these multiple

semantic definitions are contributions of our work, we should benefit from a single and unified semantics for future contributions to consistency analysis.

Natural Language Processing

The use of a CNL allows us to precisely define parsing rules and its formal semantics. Nevertheless, writing the requirements in compliance with these rules is not as simple as test engineers are accustomed to. Despite the fact that our editor is able to point out errors and help the user by autocompleting the sentences, some users are still reluctant to adopt the approach. In this light, we investigate the possibility of annotating a corpus of requirements and use machine-learning algorithms to map freestyle natural descriptions into frames. While there is no guarantee to achieve an unambiguous matching, the automatic mapping would improve the productivity significantly, besides lowering the adoption barrier. Furthermore, with this automatic mapping, all legacy descriptions could be parsed, allowing both reverse and direct (round-trip) engineering as previously discussed.

Parallelization of test scripts

Execution time can be a critical factor, especially in regression and smoke test suites that need to be run multiple times for each software update. In the case that test scripts are configured to be executed in sequence, there might be a considerable overhead. Then, we propose a strategy to optimally break the test suite into multiple smaller ones to be executed in parallel, so it should execute faster. To be able to achieve this, there must be a consistency analysis (dependency resolution), specifically to break these tests in an optimal way that minimizes the need for setup steps in each independent SUT. This has also been explored by other researchers (BERTOLINO et al., 2018) and is regarded as an important research opportunity.

Test Selection and Prioritization

Generating test cases automatically from features can sometimes uncover scenarios that can be either too trivial or unreasonable, as previously discussed in Section 4.1.5. Thus, analysts usually discard those scenarios. While they are able to delete each scenario manually, ideally the tool should offer additional filters in a mechanized fashion, such as similarity and coverage (as in Ferreira et al. (2010)). Additionally, regarding regression testing, retesting all cases is often expensive. Thus, selection and prioritization can be used to reduce the suite. The consistency analysis can also help optimize the suite by reducing redundant or duplicate steps. We can leverage the FDR analysis mechanism to give the shortest trace, thus not including the re-execution of dependencies or auxiliary steps when it is not necessary.

REFERENCES

ALEGROTH, E.; NASS, M.; OLSSON, H. H. Jautomate: A tool for system-and acceptance-test automation. In: IEEE. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. [S.l.], 2013. p. 439–446.

ANDROID. *Android UIAutomator description*. 2015. <http://developer.android.com/tools/help/uiautomator/index.html>. Visited on 2015-03-25.

ANGELOV, K. *The mechanics of the Grammatical Framework*. [S.l.]: Chalmers Tekniska Hogskola (Sweden), 2011.

APACHE. *FreeMarker Java Template Engine*. 2022. Available at: <https://freemarker.apache.org/>.

ARRUDA, F.; SAMPAIO, A.; BARROS, F. Capture and replay with text-based reuse and framework agnosticism. In: *Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc., 2016. Available at: <http://dx.doi.org/10.18293/SEKE2016-228>.

ARRUDA, F. M. C. d. *Test automation from natural language with reusable capture & replay and consistency analysis*. Master's Thesis (Master's Thesis) — Universidade Federal de Pernambuco, 2017.

BALCER, M.; HASLING, W.; OSTRAND, T. Automatic generation of test scripts from formal test specifications. In: . New York, NY, USA: ACM, 1989. v. 14, n. 8, p. 210–218. ISSN 0163-5948. Available at: <http://doi.acm.org/10.1145/75309.75332>.

BARROS, F. A.; NEVES, L.; HORI, É.; TORRES, D. The ucsCNL: A controlled natural language for use case specifications. In: *SEKE*. [S.l.: s.n.], 2011. p. 250–253.

BERNSTEIN, A.; KAUFMANN, E. GINO – a guided input natural language ontology editor. In: *Proceedings of the 5th International Conference on The Semantic Web*. Berlin, Heidelberg: Springer-Verlag, 2006. (ISWC'06), p. 144–157. ISBN 3-540-49029-9, 978-3-540-49029-6.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: *2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 85–103. ISBN 0-7695-2829-5. Available at: <http://dx.doi.org/10.1109/FOSE.2007.25>.

BERTOLINO, A.; CALABRÓ, A.; ANGELIS, G. D.; GALLEGO, M.; GARCÍA, B.; GORTÁZAR, F. When the testing gets tough, the tough get ElasTest. In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. [S.l.], 2018. p. 17–20.

BRINKSMA, E. A theory for the derivation of tests. In: NORTH-HOLLAND. *Proc. 8th Int. Conf. Protocol Specification, Testing and Verification*. [S.l.], 1988. p. 63–74.

CALDIERA, V.; ROMBACH, H. D. The goal question metric approach. *Encyclopedia of software engineering*, v. 2, n. 1994, p. 528–532, 1994.

CARVALHO, G.; FALCÃO, D.; BARROS, F.; SAMPAIO, A.; MOTA, A.; MOTTA, L.; BLACKBURN, M. NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, Elsevier, v. 95, p. 275–297, 2014.

CAVALCANTI, A.; GAUDEL, M.-C. Testing for refinement in CSP. In: SPRINGER. *International Conference on Formal Engineering Methods*. [S.l.], 2007. p. 151–170.

CAVALCANTI, A.; HIERONS, R. M.; NOGUEIRA, S.; SAMPAIO, A. A suspension-trace semantics for CSP. In: *10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*. [s.n.], 2016. p. 3–13. Available at: <http://dx.doi.org/10.1109/TASE.2016.9>.

CHANDRA, R.; KARLSSON, B. F.; LANE, N.; LIANG, C.-J. M.; NATH, S.; PADHYE, J.; RAVINDRANATH, L.; ZHAO, F. *Towards Scalable Automated Mobile App Testing*. [S.l.], 2014.

CHOMSKY, N. Deep structure, surface structure, and semantic interpretation. *Semantics*, Cambridge University Press Cambridge, p. 183–216, 1971.

CLOUDMONKEY. *MonkeyTalk*. 2013. <https://www.cloudmonkeymobile.com/monkeytalk>. Visited on 2015-05-25.

COPSTEIN, B.; OLIVEIRA, F. Automated test script generation for model-based testing. In: SBC. *Anais do IV Simpósio Brasileiro de Qualidade de Software*. [S.l.], 2005. p. 248–260.

DIMITROVA, V.; DENAUX, R.; HART, G.; DOLBEAR, C.; HOLT, I.; COHN, A. G. Involving domain experts in authoring OWL ontologies. In: *The Semantic Web - ISWC 2008: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 1–16. ISBN 978-3-540-88564-1.

FERREIRA, F.; NEVES, L.; SILVA, M.; BORBA, P. Target: a model based product line testing tool. *Tools Session of CBSoft*, 2010.

FILLMORE, C. J. The case for case'in bach & harms (eds.) universals in linguistic theory. *Holt, Rinehart, and Winston*, 1968.

GAMMA, E. *Design patterns: elements of reusable object-oriented software*. [S.l.]: Pearson Education India, 1995.

GAUDEL, M.-C. Testing can be formal, too. In: SPRINGER. *Colloquium on Trees in Algebra and Programming*. [S.l.], 1995. p. 82–96.

GEBSER, M.; KAMINSKI, R.; KAUFMANN, B.; SCHAUB, T. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

GF. *Using the Java binding to the C runtime*. 2022. Available at: <https://www.grammaticalframework.org/doc/runtime-api.html#java>.

GOMEZ, L.; NEAMTIU, I.; AZIM, T.; MILLSTEIN, T. RERAN: Timing- and touch-sensitive record and replay for android. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 72–81. ISBN 978-1-4673-3076-3. Available at: <http://dl.acm.org/citation.cfm?id=2486788.2486799>.

GRANO, G.; SCALABRINO, S.; GALL, H. C.; OLIVETO, R. An empirical investigation on the readability of manual and generated test cases. In: IEEE. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. [S.l.], 2018. p. 348–3483.

GRECHANIK, M.; XIE, Q.; FU, C. Maintaining and evolving GUI-directed test scripts. In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 408–418. ISBN 978-1-4244-3453-4. Available at: <http://dx.doi.org/10.1109/ICSE.2009.5070540>.

GRIESKAMP, W. Multi-paradigmatic model-based testing. In: *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification*. Berlin, Heidelberg: Springer-Verlag, 2006, (FATES'06/RV'06). p. 1–19. ISBN 3-540-49699-8, 978-3-540-49699-1.

GRUZITIS, N.; PAIKENS, P.; BARZDINS, G. Framenet resource grammar library for GF. In: SPRINGER. *International Workshop on Controlled Natural Language*. [S.l.], 2012. p. 121–137.

HALLETT, C.; SCOTT, D.; POWER, R. Composing questions through conceptual authoring. *Computational Linguistics*, MIT Press, v. 33, n. 1, p. 105–133, 2007.

JACKSON, D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 11, n. 2, p. 256–290, 2002.

JACKSON, D. *Software Abstractions: logic, language, and analysis*. [S.l.]: MIT press, 2012.

JONES, C. B. *Systematic software development using VDM*. [S.l.]: Prentice Hall Englewood Cliffs, 1990.

KUHN, T. A principled approach to grammars for controlled natural languages and predictive editors. *Journal of Logic, Language and Information*, Springer, v. 22, n. 1, p. 33–70, 2013.

LEOTTA, M.; CLERISSI, D.; RICCA, F.; TONELLA, P. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: *Reverse Engineering (WCRE), 2013 20th Working Conference on*. [S.l.: s.n.], 2013. p. 272–281.

LEOTTA, M.; CLERISSI, D.; RICCA, F.; SPADARO, C. Improving test suites maintainability with the page object pattern: An industrial case study. In: IEEE. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. [S.l.], 2013. p. 108–113.

LI, C. Mobile GUI test script generation from natural language descriptions using pre-trained model. In: IEEE. *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. [S.l.], 2022. p. 112–113.

LIFSCHITZ, V. *Answer set programming*. [S.l.]: Springer International Publishing, 2019.

LJUNGLÖF, P. Expressivity and complexity of the grammatical framework. 2004.

MACHADO, P.; SAMPAIO, A. Automatic test-case generation. In: *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering, PSSE, Recife, Brazil*. [S.l.: s.n.], 2010. p. 59–103.

MAHMOOD, S.; AJILA, S. A. Software requirements elicitation–a controlled experiment to measure the impact of a native natural language. In: IEEE. *2013 IEEE 37th Annual Computer Software and Applications Conference*. [S.l.], 2013. p. 437–442.

MANNING, C. D.; SURDEANU, M.; BAUER, J.; FINKEL, J. R.; BETHARD, S.; MCCLOSKY, D. The Stanford CoreNLP natural language processing toolkit. In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. [S.l.: s.n.], 2014. p. 55–60.

MILLER, G. A. WordNet: a lexical database for English. *Communications of the ACM*, ACM, v. 38, n. 11, p. 39–41, 1995.

MILLER, G. A. *WordNet: An electronic lexical database*. [S.l.]: MIT press, 1998.

MINSKY, M. A framework for representing knowledge. *The Psychology of Computer Vision*, MIT Press, 1975.

NOGUEIRA, S.; SAMPAIO, A.; MOTA, A. Test generation from state based use case models. *Formal Asp. Comput.*, v. 26, n. 3, p. 441–490, 2014. Available at: <http://dx.doi.org/10.1007/s00165-012-0258-z>.

OLIVEIRA, R. R. D.; MARTINS, R. M.; SIMAO, A. D. S. Impact of the vendor lock-in problem on testing as a service (taas). In: IEEE. *2017 IEEE International Conference on Cloud Engineering (IC2E)*. [S.l.], 2017. p. 190–196.

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, ACM, v. 31, n. 6, p. 676–686, 1988.

PILONE, D.; PITMAN, N. *UML 2.0 in a Nutshell*. [S.l.]: O'Reilly Media, Inc., 2005.

PRATCHETT, T. *Equal rites*. [S.l.]: HarperCollins, 2009.

RAFI, D. M.; MOSES, K. R. K.; PETERSEN, K.; MÄNTYLÄ, M. V. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: IEEE. *2012 7th International Workshop on Automation of Software Test (AST)*. [S.l.], 2012. p. 36–42.

RANTA, A. The GF resource grammar library. *Linguistic Issues in Language Technology*, v. 2, 2009.

RANTA, A. *Grammatical framework: Programming with multilingual grammars*. [S.l.]: CSLI Publications, Center for the Study of Language and Information Stanford, 2011.

RICHTERS, M.; GOGOLLA, M. On formalizing the UML object constraint language OCL. *ER*, Springer, v. 98, p. 449–464, 1998.

ROBOTIUM. *Robotium Recorder™*. 2013. Visited on 2015-05-25. Available at: <http://robotium.com/>.

SAMPAIO, A.; NOGUEIRA, S.; MOTA, A. Compositional verification of input-output conformance via csp refinement checking. In: SPRINGER. *International Conference on Formal Engineering Methods*. [S.l.], 2009. p. 20–48.

SAMPAIO, A.; NOGUEIRA, S.; MOTA, A.; ISOBE, Y. Sound and mechanised compositional verification of input-output conformance. *Softw. Test., Verif. Reliab.*, v. 24, n. 4, p. 289–319, 2014. Available at: <http://dx.doi.org/10.1002/stvr.1498>.

SCHNEIDER, S. *The B-method: An introduction*. [S.l.]: Palgrave, 2001.

SILVEIRA, M. B. da; RODRIGUES, E. M.; ZORZO, A. F.; COSTA, L. T.; VIEIRA, H. V.; OLIVEIRA, F. M. de. Generation of scripts for performance testing based on UML models. In: CITESEER. *SEKE*. [S.l.], 2011. p. 258–263.

SPIVEY, J. M.; ABRIAL, J. *The Z notation*. [S.l.]: Prentice Hall Hemel Hempstead, 1992.

TILLEY, S.; PARVEEN, T. *Software testing in the cloud: migration and execution*. [S.l.]: Springer Science & Business Media, 2012.

TRETMANS, J. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, v. 17, n. 3, p. 103–120, 1996.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, Wiley Online Library, v. 22, n. 5, p. 297–312, 2012.

WANG, C.; PASTORE, F.; GOKNIL, A.; BRIAND, L. C.; IQBAL, Z. UMTG: A toolset to automatically generate system test cases from use case specifications. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2015. (ESEC/FSE 2015), p. 942–945. ISBN 978-1-4503-3675-8. Available at: <http://doi.acm.org/10.1145/2786805.2803187>.

WONG, E.; ZHANG, L.; WANG, S.; LIU, T.; TAN, L. Dase: Document-assisted symbolic execution for improving automated software testing. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 620–631.

WOODCOCK, J.; DAVIES, J. *Using Z: Specification, Refinement, and Proof*. [S.l.]: Prentice Hall International, 1996.

WYNNE, M.; HELLESOY, A. *The cucumber book: behaviour-driven development for testers and developers*. [S.l.]: Pragmatic Bookshelf, 2012.

YUE, T.; ALI, S.; ZHANG, M. Rtcm: A natural language based, automated, and practical test case generation framework. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2015. (ISSTA 2015), p. 397–408. ISBN 978-1-4503-3620-8. Available at: <http://doi.acm.org/10.1145/2771783.2771799>.