



**Pós-Graduação em Ciência da Computação**

**THAÍS MELISE LOPES PINA**

**AUTO TEST GENERATOR:** a framework to generate test cases from requirements in natural language



Universidade Federal de Pernambuco  
posgraduacao@cin.ufpe.br  
[www.cin.ufpe.br/~posgraduacao](http://www.cin.ufpe.br/~posgraduacao)

Recife  
2019

Thaís Melise Lopes Pina

**AUTO TEST GENERATOR:** a framework to generate test cases from requirements in natural language

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

**Área de concentração:** Engenharia de Software

**Orientador:** *Prof. Dr. Augusto Cezar Alves Sampaio*

**Coorientador:** *Prof. Dra. Flávia de Almeida Barros*

Recife  
2019

Catálogo na fonte  
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

P645a Pina, Thaís Melise Lopes  
*Auto test generator: a framework to generate test cases from requirements in natural language* / Thaís Melise Lopes Pina. – 2019.  
81 f.: il., fig., tab.

Orientador: Augusto Cezar Alves Sampaio.  
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.  
Inclui referências e apêndices

1. Engenharia de software. 2. Especificação de requisitos. 3. Linguagem natural controlada. I. Sampaio, Augusto Cezar Alves (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-037

**Thaís Melise Lopes Pina**

**Auto Test Generator: a Framework to Generate Test Cases from Requirements in  
Natural Language**

Dissertação de Mestrado apresentada ao Programa de Pós Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 19 de fevereiro de 2019

**BANCA EXAMINADORA**

---

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

---

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

---

Prof. Dr. Augusto Cezar Alves Sampaio  
Centro de Informática / UFPE  
(Orientador)

*Dedico este trabalho a minha família, aos amigos e aos professores  
que me deram o suporte necessário para chegar até aqui.*

## AGRADECIMENTOS

Primeiramente quero agradecer a Deus, pois foi Ele que me guiou até aqui, finalizando meu curso de pós-graduação.

Depois quero agradecer a minha mãe e meu pai, Socorro e Antônio, duas pessoas que continuamente lutaram pela minha educação e me ensinaram os valores necessários para a vida. Sempre também apoiaram minhas decisões e realizações, vibram com elas e nunca me deixaram desanimar (mesmo quando eu não acreditava mais ser possível). Além do imensurável amor e carinho.

Agradeço também a minha irmã, Evelyn, por sempre me perguntar sobre o andamento do projeto e fazer uma revisão deste trabalho. Sou grata também por ela colocar em minha vida Renato (Cunhado!), seu marido, com o qual me divirto, mesmo sem ele querer.

Há mais de doze anos, tenho a meu lado um amigo e companheiro para todas as horas, meu marido, André. Além de gostar de mim, obrigada pela força, apoio, compreensão nos meus dias de estresse (que não foram poucos!) e acompanhamento em todos os projetos da faculdade, desde a graduação até agora.

Aos meus demais familiares agradeço pela presença na minha vida, carinho, amor, e compreensão por minha ausência, em certos momentos familiares. Existem aqueles que não estão mais presentes de modo físico em minha vida, mas sempre me ensinaram a ser uma pessoa ética e responsável, minhas avós, Carminha e Lourdes, e meus tios, Tia Fafá e Tio Dinho. Saudades eternas.

Aos amigos! Amigos da Motorola – em especial, Ana, Chaina, Claudio, Daniel, Filipe, Jacinto, João, Lucas, Marlom e Rodolfo – agradeço a vocês também por serem minhas fontes de distração durante a semana. Mesmo sem perceber, vocês foram de grande importância para mim, obrigada pela companhia, amizade e por escutar minhas reclamações e solucionaram, na maioria das vezes, as minhas dúvidas no projeto. Agradeço também às “amigas da academia” Alline, Kalyne e Thuany, que apesar do meu “sumiço” e outros percalços da vida estão sempre procurando se encontrar comigo e estão sempre presentes virtualmente. Obrigada aos amigos Jeaninne, Jéssica, Halles, Valentina e Vitória que estavam sempre ao meu lado nos dias de luta e nos dias de glória.

Ao projeto CIn-Motorola pelo apoio financeiro e pela infraestrutura física e pessoal. Em especial agradeço a Alexandre, Virgínia, Alice, Danila, Momenté, Fernanda, Bazante e Denis, pelo feedback constante e apoio na realização dos experimentos.

Obrigada ao professor Gustavo Carvalho pelo apoio constante em diversos aspectos do meu projeto, principalmente na construção e validação da minha CNL. Gostaria de agradecer também a Eudis Teixeira e aos professores Sérgio Soares e Liliane Fonseca pelo auxílio na descrição da minha avaliação experimental.

Agradeço aos meus orientadores, Augusto e Flávia, pelos conselhos, pelos ensinamentos, pela paciência e pela confiança para comigo na realização deste trabalho.

E, por fim, um muito obrigado a todos que diretamente ou indiretamente contribuíram para que eu chegasse até aqui.

Muito Obrigada!

*“A tarefa de viver é dura, mas fascinante.” [1]*  
(SUASSUNA, Ariano. 2013)

## ABSTRACT

Testing is essential in the software engineering development process. However, it is also one of the most costly tasks. Thus, test automation has become the goal of many researches. Since design, implementation, and execution phases depend substantially on the system requirements, it is of the utmost importance that requirements text is standardized and clear. However, most companies use free natural language to write these documents, which entails the phenomenon of (lexical and structural) ambiguity, giving rise to different interpretations. An option to mitigate this problem is via the use of a Controlled Natural Language (CNL), aiming at standardization and accuracy of texts. A CNL is a subset of a natural language that uses a restrict lexicon to a particular domain, and follow grammatical rules which guide the elaboration of sentences, thus reducing ambiguity and allowing mechanized processing, like the automatic generation of test cases from CNL requirements. This work, in the software testing area, presents the Auto Test Generator (ATG), a tool to assist the writing of requirements and the automatic generation of test cases written in English, which are then automatically translated in test scripts using an automation framework. From a requirement written in CNL, the ATG creates a Use Case (UC). Due to the standardization of the language, it is possible to perform a consistency and dependency analysis, for each UC step, through a graph of associations (dependencies and cancellations) between test actions. Test cases are generated automatically in a transparent way from UCs to the user. ATG was developed and evaluated in partnership with Motorola Mobility. Experimental evaluations were performed. From the seven requirements analyzed, it was possible to create 34 test cases in total. The generated test cases resulted in 151 steps, which were passed to the Zygon (a proprietary automated tool for testing) in order to be automated. As a result, 131 test steps were correctly automated (86% of the total given as input).

**Keywords:** Requirements Specification. Controlled Natural Language. Automatic Generation of Test Cases. Automation of Tests.



## RESUMO

Testes são essenciais nos processos de desenvolvimento de software. Contudo, esta é também uma das tarefas mais custosas. Assim sendo, a automação de testes tornou-se objetivo de diversas pesquisas. Visto que as fases de projeto, implementação e execução de testes dependem essencialmente dos requisitos do sistema, é de suma importância que eles sejam textos padronizados e de qualidade. Todavia, a maioria das empresas utiliza linguagem natural livre para escrever essa documentação, podendo assim produzir textos com ambiguidade (léxica ou estrutural), dando margem a diferentes interpretações. Uma opção para mitigar esse problema é o uso de uma Linguagem Natural Controlada – CNL, do inglês *Controlled Natural Language* – visando padronização e precisão dos textos. Uma CNL é um subconjunto de uma dada língua natural, que usa um léxico restrito a um domínio particular e regras gramaticais que orientam a elaboração de sentenças, com redução de ambiguidade e permite mecanizar o processo, como a geração automática de casos de testes a partir de requisitos escritos na CNL. Este trabalho, na área de testes de software, apresenta o Auto Test Generator (ATG), uma ferramenta para auxiliar a escrita de requisitos usados na geração automática de casos de testes escritos em inglês, que são automaticamente traduzidos em *scripts* de testes usando um framework de automação. A partir de um requisito escrito na CNL, o ATG cria um caso de uso – UC, do inglês *Use Case*. Devido à padronização da linguagem, em cada passo do UC, foi possível fazer uma análise de consistência e dependência, através de um grafo de associações (dependências e cancelamentos) entre ações de teste. Os casos de teste são gerados automaticamente de modo transparente para o usuário a partir dos UCs. O ATG foi desenvolvido e avaliado em parceria com a Motorola Mobility. Foram feitas avaliações experimentais e, a partir de sete requisitos analisados, foi possível criar 34 casos de testes no total. Os casos de teste gerados resultaram em 151 passos, que foram passados para a ferramenta Zygon (uma ferramenta proprietária de automação de testes), a fim de serem automatizados. Como resultado, 131 passos de teste foram corretamente automatizados (86% do total dado como entrada).

**Palavras-chave:** Especificação de Requisitos. Linguagem Natural Controlada. Geração Automática de Casos de Testes. Automação de Testes.

## LIST OF FIGURES

Figure 1 – ATG overview .....	18
Figure 2 – SysReq-CNL .....	22
Figure 3 – Cucumber feature .....	23
Figure 4 – Cucumber match .....	23
Figure 5 – Zygon capturing process by using BPMN.....	25
Figure 6 – Android UIAutomator accesibility events .....	26
Figure 7 – Add pre-registered action .....	26
Figure 8 – Execute test on search screen .....	27
Figure 9 – Execute test on capture screen .....	28
Figure 10 – Association between frames .....	29
Figure 11 – Test process .....	30
Figure 12 – Grammar .....	34
Figure 13 – Database structure .....	37
Figure 14 – Database example.....	39
Figure 15 – Consistency and dependency analysis.....	40
Figure 16 – TaRGeT's Feature Model .....	40
Figure 17 – Reusing action by matching similar descriptions .....	41
Figure 18 – Capture screen .....	41
Figure 19 – Choose value to variable .....	42
Figure 20 – XLS file .....	42
Figure 21 – ATG Architecture .....	43
Figure 22 – ATG tool .....	44
Figure 23 – ATG process.....	44
Figure 24 – TestCase structure .....	46
Figure 25 – Sentences well-formed .....	47
Figure 26 – Runtime Parser analysis.....	47
Figure 27 – How create aliases .....	48
Figure 28 - Use Cases screen .....	49
Figure 29 – Created slots .....	50
Figure 30 – Created frames.....	50
Figure 31 – Associations .....	50
Figure 32 – Add a dependency.....	51
Figure 33 – Input TaRGeT.....	52
Figure 34 – Test cases screen .....	53

## LIST OF TABLES

Table 1 – Comparison of related work .....	24
Table 2 – Zygon GUI Functions .....	26
Table 3 – Frame example .....	28
Table 4 – Mapping between ATG and Kaki .....	39
Table 5 – Components details .....	45
Table 6 – Experimental evaluation I - Computer configuration .....	56
Table 7 – Experimental evaluation II - Computer configuration .....	57
Table 8 – Original input TaRGeT - Word .....	66
Table 9 – Original input TaRGeT - XML .....	69
Table 10 – Original output TaRGeT - HTML .....	75

## LIST OF ALGORITHMS

Algorithm 1 – Extraction algorithm.....	36
---	----

## LIST OF CHARTS

Chart 1 – Specialist consultations.....	57
Chart 2 – ATG tool has a friendly user interface.....	58
Chart 3 – ATG interface helps the user to write requirements.....	58
Chart 4 – ATG interface helps the user to reduce the automation effort.....	59

## LIST OF ACRONYMS

<b>ATC</b>	Auto Test Coverage
<b>ATG</b>	Auto Test Generator
<b>ATP</b>	Auto Test Plan
<b>BPMN</b>	Business Process Model and Notation
<b>CFG</b>	Context Free Grammar
<b>CNF</b>	Conjunctive Normal Form
<b>CNL</b>	Controlled Natural Language
<b>CPU</b>	Central Processing Unit
<b>C&amp;R</b>	Capture and Replay
<b>DFRS</b>	Data-Flow Reactive Systems
<b>DOM</b>	Document Object Model
<b>EBNF</b>	Extended Backus-Naur Form
<b>GLR</b>	Generalized LR
<b>GQM</b>	Goal, Question and Metric
<b>GUI</b>	Guide User Interface
<b>HTML</b>	HyperText Markup Language
<b>HTTP</b>	HyperText Transfer Protocol
<b>JSON</b>	JavaScript Object Notation
<b>MBT</b>	Model-Based Testing
<b>NLP</b>	Natural Language Processing
<b>POS</b>	Parts of Speech
<b>SUT</b>	System Under Test
<b>TaRGeT</b>	Test and Requirements Generation Tool
<b>TC</b>	Test Case
<b>UC</b>	Use Case
<b>XML</b>	Extensible Markup Language

## CONTENTS

<b>1</b>	<b>INTRODUCTION .....</b>	<b>16</b>
1.1	PROBLEM STATEMENT .....	17
1.2	CONTRIBUTIONS .....	17
1.3	DOCUMENT ORGANIZATION .....	18
<b>2</b>	<b>BACKGROUND &amp; RELATED WORK.....</b>	<b>20</b>
2.1	NATURAL-LANGUAGE PROCESSING .....	20
2.2	TOOLS .....	24
2.2.1	TaRGeT .....	24
2.2.2	Zygon.....	25
2.2.3	Kaki.....	28
<b>3</b>	<b>STRATEGY .....</b>	<b>30</b>
3.1	CONTROLLED NATURAL LANGUAGE .....	31
3.1.1	Lexicon .....	31
3.1.2	Grammar .....	33
3.2	EXTRACTION ALGORITHM .....	36
3.3	CONSISTENCY AND DEPENDENCY ANALYSIS .....	39
3.4	AUTOMATIC TEST CASE GENERATION .....	40
3.5	TEST AUTOMATION.....	41
3.6	JIRA INTEGRATION .....	42
<b>4</b>	<b>TOOL SUPPORT .....</b>	<b>43</b>
4.1	PARSER .....	46
4.2	USE CASES .....	48
4.3	KAKI INTEGRATION .....	49
4.4	TARGET INTEGRATION.....	51
4.5	TEST CASES .....	52
<b>5</b>	<b>EXPERIMENTAL EVALUATION .....</b>	<b>54</b>
5.1	PLANNING .....	54
5.1.1	Definition.....	54
5.1.2	Ethical Concerns.....	54
5.1.3	Research questions .....	55
5.1.4	Participants .....	55
5.1.5	Procedures and data collection.....	55
5.2	EXECUTION AND RESULTS.....	56
5.2.1	Experimental evaluation – stage I.....	56
5.2.2	Experimental evaluation – stage II.....	57
5.2.3	Results.....	57
5.3	THREATS TO VALIDITY .....	59

6	<b>CONCLUSIONS .....</b>	<b>61</b>
6.1	<b>FUTURE WORK.....</b>	<b>61</b>
	<b>REFERENCES .....</b>	<b>63</b>
	<b>APPENDIX A – TARGET EXAMPLE.....</b>	<b>66</b>
	<b>APPENDIX B – EBFN NOTATION .....</b>	<b>78</b>
	<b>APPENDIX C – PARTICIPANT CONSENT FORM.....</b>	<b>79</b>
	<b>APPENDIX D – ATG SURVEY.....</b>	<b>81</b>



# 1

## INTRODUCTION

The design and implementation of a software system are based on a survey of the capacities or conditions to be met [2]. This list is usually called *requirements specification* and consists of statements in natural language and/or diagrams, elicited by clients and users, on the functions the system must provide and the constraints under which it must operate [3].

The Use Cases (UCs) [4] are created from the requirements analysis and serve as a guide for developers, testers and analysts to conduct their respective activities and give customers an idea of what to expect from the system [5]. In order to guarantee the software quality, the system under development must go through different testing phases (e.g., exploratory tests, regression testing, among others), aiming to anticipate errors that could take place when the software is released to the users. Thus, testing is a crucial activity to verify the correct implementation of the system requirements [6].

Faced with the high competitiveness in the information technology area nowadays, one of the main concerns in the software market is the product quality and, thus, an increase in interest and investment in defect prevention techniques is noticed. Since this task requires much effort, reaching half the total cost of software development [7] and, if performed casually, time is wasted and, even worse, errors pass through undetected [3].

Testing activities involve the creation of test cases (TCs), which simulate the user interaction with the system. Usually, test cases consist of a set of steps, described in natural language, to be executed against the system. TCs may be manually or automatically executed.

Despite the consolidation of the testing area as an essential verification activity, it is not always feasible to complete a testing campaign due to deadlines and/or financial limitations [8]. This explains the growing demand for automation and optimization of the quality assurance process, from the generation to the execution of TCs. Automation seeks to improve the testing process, making it more agile, less susceptible to errors, less dependent on human interaction, and with higher coverage.

Note that the cost of the testing process is proportional to the amount of TCs to run and to maintain and test automation requires a specialized professional to create test scripts, to define input values for the variables, analyze eventual associations between the tests, and also run them [9].

Automation tools can also aid this process, but the language used in the descriptions of the requirements (natural and unstructured language) makes it challenging to derive TCs directly from requirements, through such tools. Because the ease of writing, derived from the use of natural language, this can cause software implementation defects such as ambiguity, incompleteness and requirements inconsistency [3].

However, the work reported in [10] indicates, with hands-on testing experiences, that forcing programmers to learn new notations and specific tools is not the best option, reinforcing the use of well-known notations and environments. In other words, requiring testers to develop a representation from a model or formal specification, which describes the expected behavior for

the software being tested in order to allow the efficient automatic transformation, it is not indicated.

## 1.1 PROBLEM STATEMENT

Testing tasks tend to be repetitive. For example, in regression test campaigns to detect errors that may have been introduced to the modified code, each new version of the software must be tested in its entirety, to verify the correctness of the system [11]. However, because of cost constraints, alternative approaches select and execute only a subset of the set of tests [12].

Such problems are magnified when we consider mobile development because scripting for these applications has a higher level of complexity when compared to traditional systems. In mobile applications, more test cases are needed to increase coverage, since there are different execution environments in which the conditions are even more uncontrollable, such as dynamic localization, different hardware platforms, networks, and sensors [13].

Therefore, automation of code-based scripting tests can mitigate efforts to run a whole test suite manually, but also gives rise to some problems: (I) it requires the hiring of specialized people to automate TCs; (II) tests code (scripts) maintenance is not an easy task and is usually necessary, mainly when it is based on user interface; (III) each new test case must be created from scratch by a specialized professional, even when it is similar to another previously created script.

Another way to automate test execution is to use Capture and Replay (C&R) tools, which does not require programming skills and can be used during the test campaign while testing is performed manually. However, the problem of item III remains, since current C & R tools allow little reuse of TCs [14] and require testers to run all TCs at least once.

However, the first step to perform the tests is to create the TCs, and usually, it is a hard job because the automatic test generation tools have different focuses or behaviors [15]. When we analyze these tools, the following criteria should be observed: (I) input quality (is it complete? – does it provide sufficient, clear, unambiguous information?); (II) the quality of the internal process of generation? (is it systematic and deterministic?); and (III) the coverage and representation of the suites generated (does the generated test suite cover all the functionalities? Is the formalism of the output representation clear to the testers who will manually perform the tests?)

The present work is focused on the first criterion presented above, but it does not exclude the others. Generally, the test-generation tools receive as input or requirements or more detailed use-case specifications from which the tests are derived. As stated earlier, this entry must be complete and unambiguous to preserve the quality of the entire testing process.

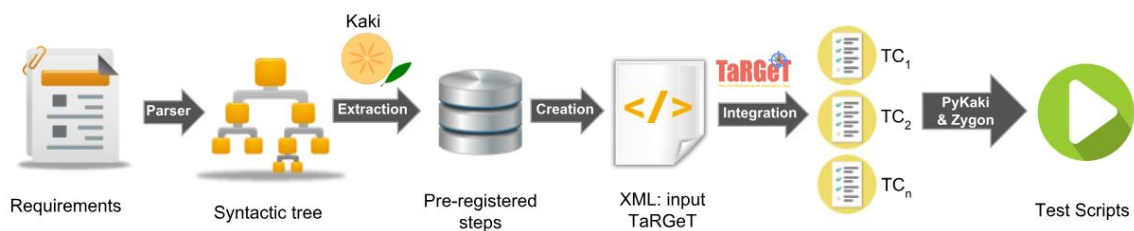
## 1.2 CONTRIBUTIONS

The main problem we tackle in this work is an integrated approach to generate automatic test cases from requirements written in a controlled natural language. Considering this scenario, the primary goal of the Auto Test Generator (ATG) is to create a strategy to generate automated test cases, from requirements written in natural language, combining the development of new components with some existing tools: TaRGeT (Test and Requirements Generation Tool) [16],

Kaki and Zygon [17] – all developed in partnership with Motorola Mobility. The most specific contributions of the present work are as follows:

- Assist in writing the requirements (through a graphical interface which indicates the next grammatical classes or reserved words expected in the sentence);
- Mechanize the adherence to the adopted Controlled Natural Language (CNL);
- Insert associations (cancellations and dependencies) between the tests, through integration with the Kaki tool;
- Automatically generate textual test cases (according to the proposed CNL) with the aid of TaRGeT;
- Encoding of the textual test cases test scripts using an automation framework (Zygon).

Figure 1 presents the project overview.



**Figure 1 – ATG overview**

It is possible to subdivide the ATG into four processing modules: parsing, extraction, creation, and integration. The parsing is the module responsible for the suggestion of the subsequent grammatical classes or reserved words expected, as well as for the interpretation of the requirement for the generation of the syntactic tree. The extraction module is responsible for extracting information from the syntax tree, transforming the requirement sentences into test actions in the Kaki language and grouping them into use cases, which in turn will serve as input to TaRGeT (Test and Requirements Generation Tool). The creation of the TaRGeT's input – XML (Extensible Markup Language) file – and the entire graphical interface are treated in the creation module. The communications between the external tools (TaRGeT and Dalek) are considered in the integration module. Due to the use of the Kaki language, the tests generated by ATG are consistent and can be directly automated in a framework like Zygon.

### 1.3 DOCUMENT ORGANIZATION

The remainder of this work is structured within the following chapters.

- Chapter 2 discusses related work and basic concepts. Notably, we describe alternatives to test automation, as well as to the natural language processing. Besides, we present the tools used with our approach to generate/automate test cases.
- Chapter 3 explains our proposed strategy: requirement writing by using the controlled natural language; the syntax tree generation through parsing; details of the extraction algorithm; the approach to integrate TaRGeT, Kaki, Zygon and Dalek; and the the test automation process.
- Chapter 4 describes the implemented tool (ATG)

- Chapter 5 explains the conducted evaluations with the achieved results.
- Chapter 6 presents our conclusions and discusses future work.

# 2

## BACKGROUND & RELATED WORK

In this chapter, we contextualize our work by introducing fundamental concepts, and we discuss the related approaches concerning the main scientific contributions of this research. In Section 2.1, we highlight the problem of directly generating TCs from natural language requirements. We also expose some alternative approaches that use a controlled natural language to verify errors beforehand and try to ease the burden of mapping ambiguous specifications to tests. Then, in Section 2.2, we introduce the accessory tools that we used integrated with ATG.

### 2.1 NATURAL-LANGUAGE PROCESSING

As noted earlier, forcing users to adopt unknown notations is time-consuming and inefficient. This, together with the lack of readability, makes the use of natural language processing (NLP) an excellent alternative to the task since it is a language of general knowledge, except for the specificities of controlled grammar. However, because of its ambiguous nature, it is difficult to verify the consistency and accuracy of a mapping between requirements and a test automation framework without human intervention. The search for an optimal mapping between natural language, descriptions and executable tests has been an active research area.

Due to the vast literature on natural-language processing, we focus here on approaches that are closely related to ATG. PENG<sup>1</sup> (Processable English) is a computer-processable CNL for writing unambiguous [18]. PENG requirements are a subset of English and are defined by a restricted vocabulary and grammar. Such a vocabulary consists of words of specific domain content that can be described by the author instantly and predefined function words that form the skeleton of the language [18]. PENG can be deterministically analyzed and translated into structures of representation of the discourse and also interpreted into first-order predicate logic. However, there are no correlations of the work with the automatic generation of tests.

The work reported in [19] presents RETNA, a requirements analysis tool. It receives natural language requirements as input and, with human interaction, it transforms them into a logical notation. From this, RETNA generates test cases. This dependence on human interaction to create the requirements in the particular logical notation for RETNA creates an overhead of both time and specialized people.

The work present in [20] addresses the generation of tests case from natural language specifications, but it is necessary massive intervention from the user to generate TCs. To use the methodology presented in [20], one needs three types of effort: (I) to deal with a complex dictionary: identifying and partitioning inputs and outputs, as well as defining mapping functions; (II) to translate abstract test cases manually to executable ones; (III) to define scenario by using combinatorial designs. Approaches like ours facilitate the automation of the tests because it uses a CNL that limits the possibilities of writing. On the other hand, there is a restriction on the structure of CNL.

---

<sup>1</sup> <http://web.science.mq.edu.au/~rolfs/peng/>

The requirements written in [21] are represented in a strict if-then sentence template, and is based on three elements: initial condition (if), consequence (then) and final condition (until). It is a very restricted grammar for our purposes because expressiveness is an essential feature. Furthermore, it does not specify which kind of execution from TCs generated.

Processing NL requirements seems more common than UCs. However, the approaches presented in [16], [22] and [23] receive UCs described in natural language as input, unlike our work, which it receives requirements. These works have another similarity between them because all these works generate TCs for manual execution. Furthermore, in contrast to those works, our method considers dependency injection between statements from generated TCs (i.e., there is a verification of consistency in the sequence of steps, thus the addition of new steps may be also suggested).

DASE (Document-Assisted Symbolic Execution) is an approach to improve the creation of automatic tests and the detection of bugs [24]. DASE performs a natural language processing along with heuristics to parse the text of the program documentation and then extracts input constraints automatically to guide a symbolic execution. Fortunately, information about input constraints usually exists in software documents such as program man pages (for example, an output from `man rm`) and comments from header files (e.g., `elf.h`) [24].

The work reported in [25] uses natural language as input to its automated test generation strategy: NAT2TEST. It was developed to generate test cases for timed reactive systems, considering examples provided by Embraer<sup>2</sup> and Daimler<sup>3</sup>. First, a syntactic analysis of requirements is performed, based on the CNL called SysReq-CNL, among others. Defined as a CFG (Context-Free Grammar), as shown in Figure 2, SysReq-CNL is used to provide structure to the text, aiming at automation of test case generation, besides mitigating ambiguity. This grammar follows some lexical divisions, such as: determiners (DETER); nouns (NSING for singular and NPLUR for plural); adjectives (ADJ); adverbs (ADV); verbs (VBASE / VPRES3RD / VTOBE\_PRE3 / VTOBE\_PRE / VTOBE\_PAST3 / VTOBE\_PAST); conjunctions (CONJ); prepositions (PREP); numbers (NUMBER); and comparisons symbols (COMP).

The results of syntactic analysis are syntactic trees that serve as input to a semantic analysis. Through the mapping of words into semantic representations (requirement frames, based on the case grammar theory [26]), an intermediate formalism (models of Data-Flow Reactive Systems – DFRSs) is generated. Then, TCs can be derived with the aid of more concrete formalisms such as Software Cost Reduction [27], Internal Model Representation [28], and Communicating Sequential Processes [29], among others. This work is particularly relevant to us, as we base our grammar in that defined in, there is no mention of dependency injection between the test actions in the TCs generated.

---

<sup>2</sup> Empresa Brasileira de Aeronáutica – <http://www.embraer.com/en-us/pages/home.aspx>

<sup>3</sup> <https://www.daimler.com/en/>

```

Requirement → ConditionalClause COMMA ActionClause;
ConditionalClause → CONJ AndCondition;
AndCondition → AndCondition COMMA AND OrCondition | OrCondition;
OrCondition → OrCondition OR Condition | Condition;
Condition → NounPhrase VerbPhraseCondition;
ActionClause → NounPhrase VerbPhraseAction;
NounPhrase → DET? ADJ* Noun+;
Noun → NSING | NPLUR;
VerbPhraseCondition → VerbCondition NOT? ComparativeTerm? VerbComplement;
VerbCondition → (VPRE3RD | VTOBE_PRE3 | VTOBE_PRE | VTOBE_PAST3 | VTOBE_PAST);
ComparativeTerm → (COMP (OR NOT? COMP)?);
VerbPhraseAction → SHALL (VerbAction VerbComplement |
    COLON VerbAction VerbComplement (COMMA VerbActionVerbComplement)+);
VerbAction → VBASE;
VerbComplement → VariableState? PrepositionalPhrase*;
VariableState → (NounPhrase | ADV | ADJ | NUMBER);
PrepositionalPhrase → PREP VariableState;

```

**Figure 2 – SysReq-CNL**  
Source: [27]

Below, there are some valid requirements in compliance with the SysReq grammar (Source: [27]):

- *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.*  
Vending machine
- *When the left priority button is not pressed, and the right priority button is not pressed, and the left command is on neutral position, and the right command is on neutral position, the Priority Logic Function shall assign 0 to the Command-In-Control.*  
Priority Control
- *When the water pressure becomes higher than or equal to 900, and the pressure mode is low, the Safety Injection System shall assign permitted to the pressure mode.*  
Nuclear Power Plant

In [30], the authors introduce a command-line tool: Cucumber. In other words, one must have minimum computer knowledge to use it. It is possible to apply this tool to automate new tests or tests that developers have already done. Cucumber is integrated with some approaches like QTP<sup>4</sup> and Selenium IDE<sup>5</sup>, so, one can get automatable tests, but needs to learn external tools.

Firstly, the tool reads some given features (some text files written in a structure natural language). Then, the tool examines them for scenarios (list of steps) to test, and runs this against

<sup>4</sup> <https://www.tutorialspoint.com/qtp/>

<sup>5</sup> <https://www.seleniumhq.org/projects/ide/>

the system under test (SUT), like our work. For all this to happen, these feature files are standardized according to some basic syntax rules (Gherkin is the name for this set of rules). A concrete example of feature is described in Figure 3.

```
Download step_definitions/Intro/features/cash_withdrawal.feature
Feature: Cash withdrawal
  Scenario: Successful withdrawal from an account in credit
    Given I have $100 in my account
    When I request $20
    Then $20 should be dispensed
```

**Figure 3 – Cucumber feature**  
Source: [30]

Gherkin avoids vague requirements due giving real examples of how the system should be run. Similarly to our strategy, also there exists a concern with the language being readable by stakeholders as well as interpreted by computers. Each step of a scenario should be mapped into code, as illustrated by Figure 4.

```
Given /I have deposited \$(.*) in my Account/ do |amount|
  # TODO: code goes here
end
```

**Figure 4 – Cucumber match**  
Source: [30]

However, in addition to being developer-centric automation, detailed reuse and consistency/dependency checking are outside the scope of the tool, as our work.

Table 1 summarizes our analyses of related work considering these perspectives. Here, we analyze work from six different perspectives: (I) domain: whether the modeling approach is tailored to a specific area; (II) input: how the system requirements are documented; (III) specific notation: whether the notation is trivial for non-experts; (IV) tests: if the tool generates TCs and whether it is for manual or automatic execution; (V) human: analyzes whether user intervention is required for the generation of test cases; (VI) dependency injection: we consider the consistency and associations between the TC statements.



Table 1 – Comparison of related work

	Domain	Input	Specific notation	Tests	Human	Dependency Injection
<b>ATG</b>	<b>General</b>	<b>NL requirements</b>	<b>No</b>	<b>Automatic</b>	<b>No</b>	<b>Yes</b>
[16]	General	UCs	Yes	Manual	No	No
[18]	General	NL requirements	No	No	-	-
[19]	General	NL requirements	Yes	Automatic	Yes	No
[20]	General	NL requirements	Yes	Yes, but it does not specify which type	Yes	No
[21]	General	NL requirements	Yes	Yes, but it does not specify which type	No	No
[22]	General	UCs	No	Manual	No	No
[23]	Mobile	UCs	Yes	Manual	No	No
[24]	General	NL requirements	Yes	Manual	No	No
[27]	Embedded	NL requirements	No	Automatic	No	No
[30]	General	NL requirements	Yes	Automatic	No	No

## 2.2 TOOLS

In this section, we detail each one of three tools that are integrated into our strategy: TaRGeT [16], Kaki and Zygon [17]. All of them were developed in partnership with Motorola Mobility. ATG combined the automatic test generation from TaRGeT, test automation with reuse from Zygon, and the consistency and dependency analysis from Kaki. Examples to illustrate the use of all these tools are presented in Chapter 4.

### 2.2.1 TaRGeT<sup>6</sup>

The main purpose of TaRGeT is, in an integrated and systematic way, to deal with requirements and test artifacts. With the approach used, the test cases can be generated automatically from scenarios of use cases written in natural language [16]. The tool was developed as a line of software products due to the need for different profiles, identified through customer requests.

The input for TaRGeT (as we show in Appendix 0) are UC written following an XML schema, which is designed to contain the information necessary to generate the procedure, description, initial conditions and related requirements, among other information associated with a test case. Also, the tool can generate traceability matrices between test cases, use cases and requirements.

This tool was developed according to the Model-Based Testing (MBT) approach. Typically, MBT involves the creation of formal models. Using TaRGeT, however, this is completely hidden from the user. A formal model (either a labelled transition system or a process

<sup>6</sup> <https://twiki.cin.ufpe.br/twiki/bin/view/TestProductLines/TaRGeTProductLine>

algebraic model) is generated from the textual use cases; these models are internally used by the tool to generate (textual) test cases for manual execution.

### 2.2.2 Zygon

Zygon was created with the primary aim of being a tool with which users without programming knowledge could automate tests in the mobile context, as well as reuse the already automated tests. Besides, it is framework independent and, today, is part of Motorola's testing operation [17].

A test action is the base unit of Zygon. It is text-based, recursive, and framework-free. The implemented C&R strategy means that everything that is done on the screen of the cell phone is mapped to the test actions.

These test actions have been modeled so that they can refer to others using the composite pattern. Thus, in applying word-processing algorithms, it is possible to identify a correspondence between the new natural language TC descriptions to previously registered test actions, reusing them to automate the TCs.

Figure 5 illustrates the complete process of how the tool operates.

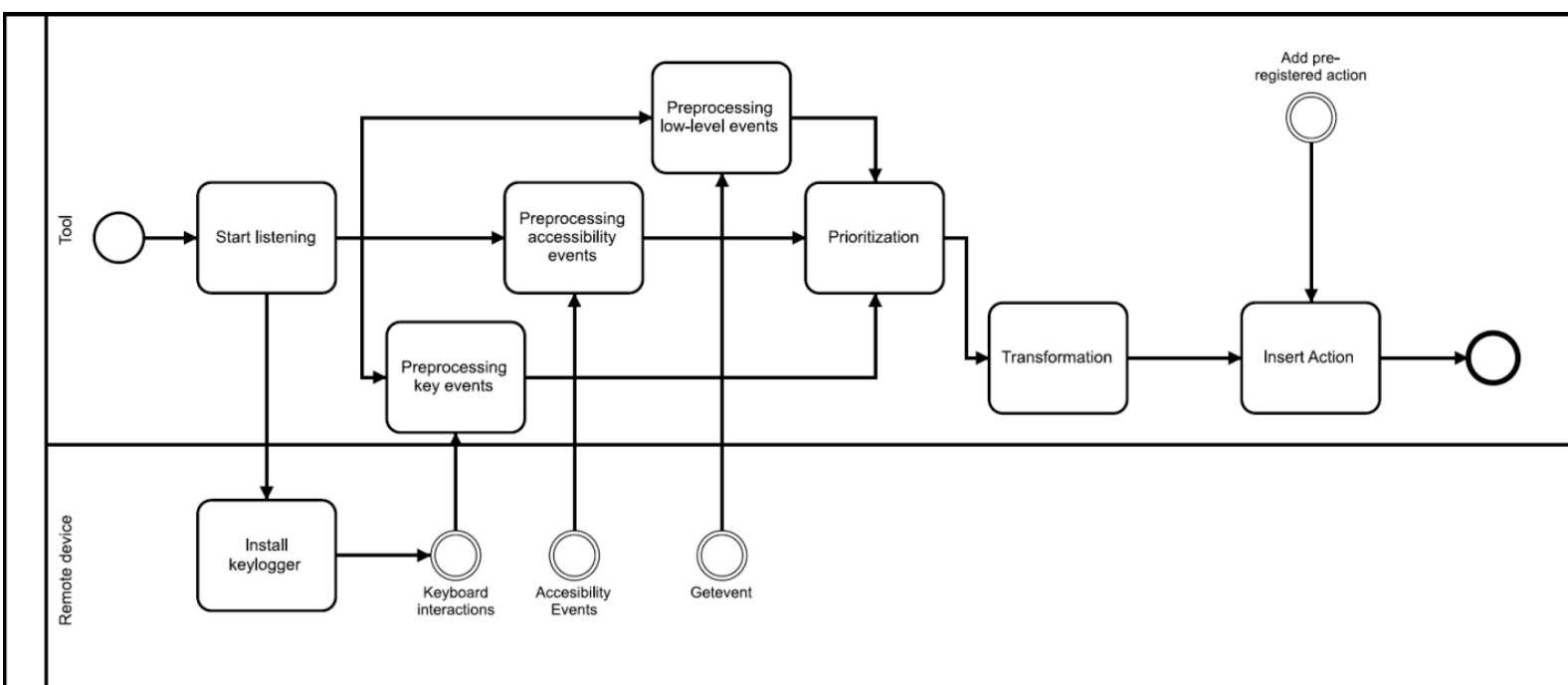


Figure 5 – Zygon capturing process by using BPMN

Source: [17]

A distinctive feature of this tool when contrasted to others similar approaches is reuse and the fact that captured events are stored at a high-level. For example, instead of capturing events such as *clicking on (x, y)*, the tool captures “click on button with description "Apps"” according to Android accessibility events. In this way the tool mitigates some compatibility issues on devices (due to different screen sizes, for example), moreover giving a more readable way to present information to the user. There is a listener that processes the Android accessibility log event, among others. An example is illustrated in Figure 6.

```

01-19 21:58:14.040 EventType: TYPE_VIEW_CLICKED; EventTime:
22931988; PackageName: com.lge.launcher2;
MovementGranularity: 0; Action: 0 [ ClassName:
android.widget.TextView; Text: []; ContentDescription
: Aplicativos; ItemCount: -1; CurrentItemIndex: -1;
IsEnabled: true; IsPassword: false; IsChecked: false;
IsFullScreen: false; Scrollable: false; BeforeText:
null; FromIndex: -1; ToIndex: -1; ScrollX: -1;
ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1;
AddedCount: -1; RemovedCount: -1; ParcelableData:
null ]; recordCount: 0

```

Figure 6 – Android UIAutomator accesibility events

However, listening only to high-level events of single-touch is not enough to cover more complex interactions, such as swipe or hard key events. Thus, the tool includes a preprocessing module to interpret low-level events.

With so many events happening at the same time, an overlap is possible to happen, due to the various streams of input to interpret. So the tool prioritizes actions, by considering defining a hierarchy of priorities between different event sources, and when two or more events overlap, given a short time, the one with the highest priority is chosen.

Zygon GUI allows the user to enter some predefined test steps, reuse test actions or even an entire test and add checks to the graphical interface, as shown in Figure 7.

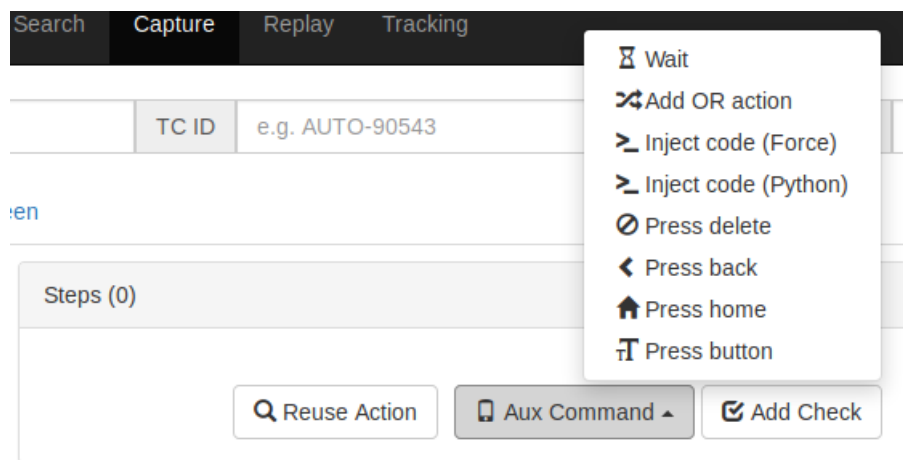


Figure 7 – Add pre-registered action

Table 2 summarizes each feature.

Table 2 – Zygon GUI Functions

Command	Function
Wait	Add some time to begins the next step
Add OR action	Represents a logic choice
Inject code	For specific/advanced API commands or checks.
Press delete	Delete some text
Press back	Click on back button
Press home	Click on home button
Press button	Click on some text
Reuse Action	Search a previously stored action
Add Check	Check on user interface if content is/has some text

Each step might contain the following types of actions associated with it: (I) create variable; (II) undo variable; (III) execute; (IV) duplicate; (V) settings; (VI) delete. These actions are described below.

- Create variable (\$): Create a variable instead of a fixed value. All variables are listed to the left of the sequence of steps and are very useful for speeding up the reuse of test actions;
- Undo variable (C): Removes the variable and maintains a fixed value;
- Execute (▶): Perform step individually;
- Duplicate (↻): Duplicate step;
- Settings (⚙): Opens the step settings. It serves, for example, to modify the waiting time;
- Delete (✖): Remove step.

After capturing a sequence of steps, the user must provide a representative description to store in the database. The database used by Zygon is Neo4J<sup>7</sup> because it is a graph-oriented database and allows one to find similar subgraphs and analyze transitive connections between test actions.

There are two possibilities to run the test: in the search screen, look for the test and click on the *Execute* button (Figure 8) or click on the *play* button (Figure 9).

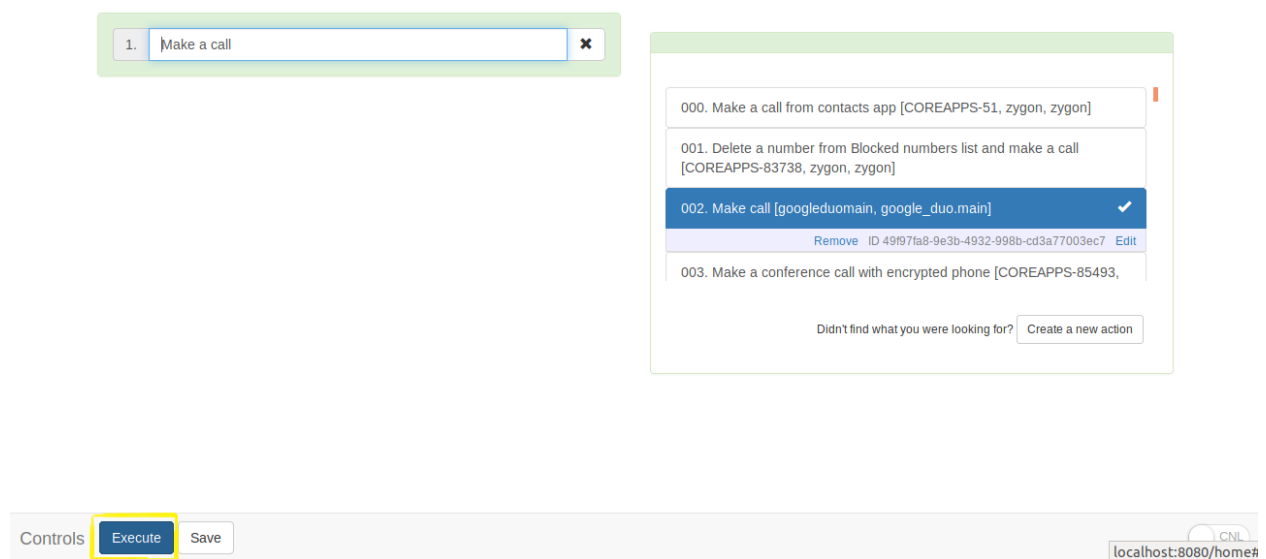


Figure 8 – Execute test on search screen

<sup>7</sup> <https://neo4j.com/>

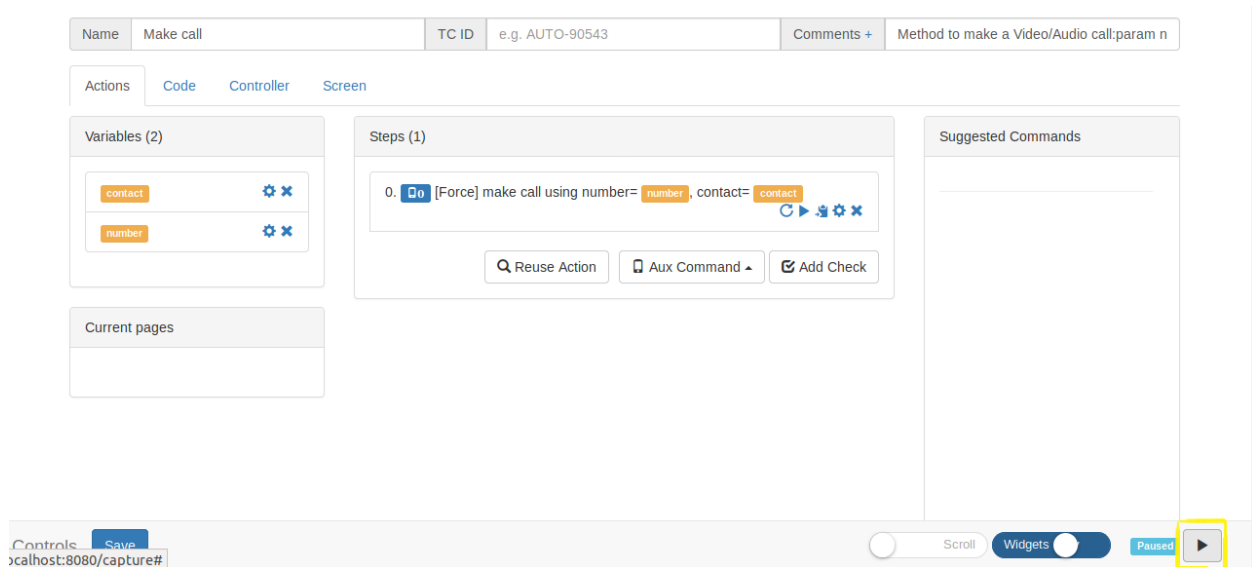


Figure 9 – Execute test on capture screen

### 2.2.3 Kaki

Observing a large number of test cases and their test steps, in the context of the cooperation with Motorola, it was possible to identify a typical pattern: it is always an *operation* on a *patient*, the passive object for the respective operation.

From this pattern, Kaki, a text editor, was built based on the concept of *frames* to represent knowledge [31]. Each *frame* consists of *slots*, each with a specific purpose. There are two fixed *slots* (operation and patient) and extra *slots* dynamically created. Thus, the tool can be check if a sentence is well-formed.

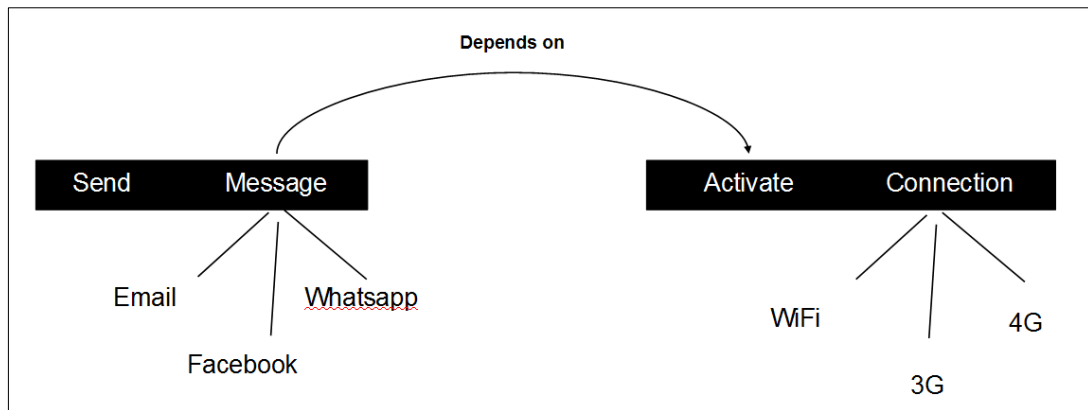
For instance, a *frame* which represents a scenario to send an e-mail, it needs action, receiver, message, title, among others as *slots* (see Table 3).

Table 3 – Frame example

Required (static)		Extra (dynamic)				
Operation	Patient	Sender	Receiver	Title	Body	...
Send	Email Message	tmlp@cin.ufpe.br	acas@cin.ufpe.br	SBMF	The article has been accepted	...

Besides, Kaki allows establishing associations (dependencies and cancellations) between test steps (actions). From these user-informed associations, the tool generates a model to verify consistency, as well as suggesting the insertion of missing steps to make test cases consistent [17]. Without Kaki the consistency of performing a sequence of actions would depend solely on the experience of the tester or test engineer and the individual knowledge about the domain provided.

Figure 10 exemplifies the consistency notion. In the example, sending a message requires that a connection is enabled. It is noticed that both test actions are valid individually, but the execution of the first fails if the second one is not previously. As another kind of association, one action can cancel the effect of another – "Activate airplane mode" cancels the "Activate Connection" action.



**Figure 10 – Association between frames**

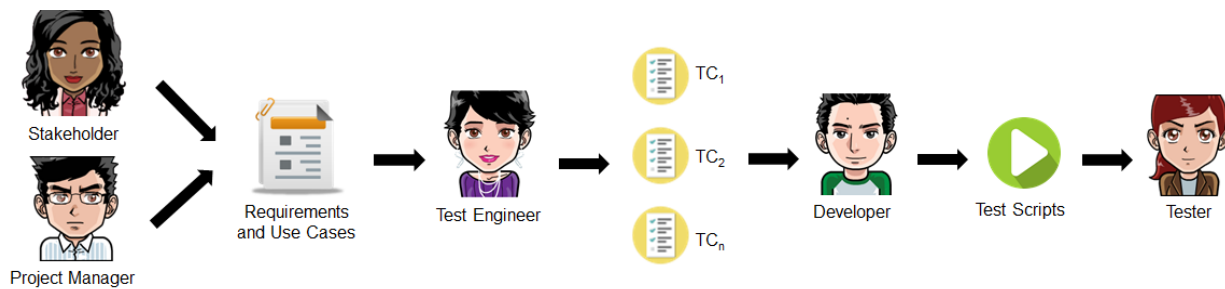
Source: [17]

In the Kaki graphical interface it is possible to define the dependencies and behaviors between test actions, and thus it is possible to provide a coherent (and possibly optimal) order of execution. The valid actions and their dependencies are represented as a domain model that is translated into Alloy [17]. In summary, the tool can detect inconsistent sequences as well as suggest the insertion of actions to automatically make it consistent.

# 3

## STRATEGY

A typical process of creating automated tests begins with the requirements made by the stakeholders. Then the project manager receives these specifications and creates the requirements and their respective use cases. From the UCs the test engineer examines them and creates test cases. Finally, the developer creates test scripts to automate the TCs, which is executed by the tester. Figure 11 exemplifies this process.



**Figure 11 – Test process**

The requirements and use cases are mostly written in free natural language, being vulnerable to the problems of free language: ambiguity, and imprecision. Such problems can affect the development process because it is known that the requirements are the basis for the entire development process since they must represent precisely what the client wants the system to do. From the requirements, the use cases are elaborated, and these, in turn, also written in natural language, are the basis for analysis and design and test cases.

The impact of problems introduced by the use of free natural languages can go beyond the development phase, also reflected in the testing phase, considering that the test cases are projected from the use case specifications. As illustrated in Figure 11, this process is a chain that, if at any time is misinterpreted, can compromise the system as a whole. In order to ensure the quality of the input specifications in the software development process, some companies use CNLs specially designed to meet their particular needs, as well as impose standardization on the requirements without losing the naturalness.

One of the main contributions of our work is the definition of a complete grammar for writing the requirements, as well as a lexicon for the domain of tests of mobile devices. Once the knowledge bases are ready, they are parsed to check for adherence to the structure of the CNL.

However, only the CNL does not supply the demand to create test cases to make the consistency analysis between them and to automate the TCs. So to support these tasks, we integrate TaRGeT, Kaki, and Zygon, respectively, into our solution. It is worth mentioning that we refer to ATG as a single tool, considering its integration with all these tools.

To support all these tasks in an automated way, we have to deal with the following challenges:

- I. Create a CNL easy to use, and to provide structure to the text, besides mitigating ambiguity;

- II. Supply consistency and dependency analysis for the CNL semantics to check the individual instances and their relationships;
- III. Generate a comprehensive suite of tests for the various possible scenarios;
- IV. Automate the generated test cases;
- V. Import the test suite to the usual Motorola Mobility platform, Jira<sup>8</sup>;

A summary of the overall approach was already presented in Figure 11. In the next section we detail our strategy; we start with the syntactic analysis from requirements (Section 3.1). In Section 3.2 we describe how we implement the extraction algorithm, and in Section 3.3 how we use the consistency and dependency analysis from Kaki. Section 3.4 presents our strategy to generate test cases automatically and Section 3.5 explains the test automation. Finally, the Jira integration is shown in Section 3.6.

### 3.1 CONTROLLED NATURAL LANGUAGE

The syntactic analysis of the ATG strategy verifies whether the system requirements are written according to a particular CNL, which is precisely defined in terms of grammar production rules.

The CNL is based on the English language. The vocabulary and syntactic structures commonly encountered in a mobile device environment have been taken into account in order to stimulate the CNL adoption.

However, we have designed a more formal CNL, with a lexicon of words and types with pre-defined terms and grammar, used to restrict the buildings sentences for specifying requirements. In the future, this grammar will be used as the basis for the mapping of sentences to Kaki, as we detail in Section 4.3.

However, users can find it challenging to write with the syntactic constraints of a CNL. Thus, to facilitate the work to write requirements with a restricted language, we implemented a predictive and guided approach, as suggested by [32]. The on-the-fly parser analyses each word and then suggests which are the next accepted grammar classes or keywords. With this, we accomplish our first challenge.

#### 3.1.1 Lexicon

The ATG Lexicon was built based on the context of testing engineering for mobile devices, thus encompassing special terms for the applications of these devices. As already mentioned, the terms of the lexicon are in English, which is the standard language in writing tests at Motorola Mobility.

These terms were classified according to their grammatical class, also known as Parts of Speech (POS) [33], such as determinant, name, verb, adjective, adverb, preposition and conjunction. In order to simplify the grammar, in addition to these classes, we created a category for numbers, one for comparisons and other for generic terms, which all words must necessarily be enclosed in double quotation marks.

---

<sup>8</sup> <https://br.atlassian.com/software/jira>



- determiners (DETER) are used to identify a noun or a set of them (e.g., a number, an article or a personal pronoun)
- nouns (NSING for singular and NPLUR for plural) represents the domain entity;
- verbs with inflections:
  - VBASE – base form;
  - VPAST – past indicative;
  - VPRE – present indicative
  - VPRE3RD – past indicative for the 3<sup>rd</sup> person;
  - VTOBE\_PRE – verb to be in present indicative;
  - VTOBE\_PRE3 – verb to be in present indicative for the 3<sup>rd</sup> person;
  - VTOBE\_PAST – verb to be in past indicative;
  - VTOBE\_PAST3 – verb to be in past indicative for the 3<sup>rd</sup> person;
  - VPHRASAL – phrasal verb.
- adjective (ADJ);
- adverbs (ADV);
- prepositions (PREP);
- conjunctions (CONJ);
- numbers (NUMBER);
- comparisons (e.g., greater than)
- generic (GENERIC) represents domain specific terms (e.g., “AUDIO\_MP3”).

Yet, we have special entries to identify keywords that are used in the grammar definition: “and” (AND), “or” (OR), “not” (NOT), “do not” (DO NOT), “may” (MAY), “must” (MUST), “case” (CASE), “if” (IF), “then” (THEN), “end” (END) “:” (COLON), and “,” (COMMA), “;” (SEMICOLON) and QUOTATION\_MARKS for quotation marks.

This CNL does not allow personal and deictic pronouns (e.g., this, those), thus eliminating the occurrence of anaphora, that is, expressions that refer to names of the same sentence. Not allowing the occurrence of pronouns in the language is another way to limit the complexity of sentences and reduce ambiguity.

Finally, we emphasize that since the lexicon is domain dependent, it must be created and maintained manually considering the current domain of the system. Despite the initial effort, vocabulary tends to become stable, which minimizes maintenance effort. This is a natural assumption for Natural Language Processing (NLP) systems that rely on a predefined set of lexical entries. However, it is possible to reuse part of an existing lexicon for a new application domain (for instance, prepositions and conjunctions).

Also there are some recommendations to be followed in the insertion of new terms, such as:

- Avoid abbreviations: abbreviations may not be consensual within a company. In this way, a user can add a new term in the lexicon whose abbreviation has already been added, allowing a single object to be referenced by more than one symbol. However, it is often unavoidable to deal with abbreviations, for the sake of simplicity, ease and even by the custom of the environment, e.g., SIM card, SMS.

- Treat each generic term as a name: abbreviations, symbols, or another sequence of characters that do not represent a domain entity must be enclosed in double quotation marks and treated as a name to avoid lexical pollution. For example, in the Click on label statement "Join the Moto community", the term quoted is treated as a name.
  - Do not separate compound terms that represent a specific entity in the domain. For example, the term "IP address".

### 3.1.2 Grammar

The syntax determines how the words will be combined in the formation of grammatical sentences, specifying their internal structure and functioning. This work is based on the Grammar of Immediate Constituents approach, according to which the sentence can be divided into other constituents, until reaching fundamental constituents, such as names and verbs [34]. A constituent can be a word or a group of words that occur as a unit in the rules of grammar rewriting. Note that these constituents will be nodes in the syntax tree.

The grammar used in this work was based on SysReq-CNL from [25] with a few modifications. It has been defined as a CFG, represented by the Extended Backus-Naur Form (EBNF) notation – capitalized words indicate terminal symbols and the other symbols of the EBNF notation used are explained in Appendix 0. In this work, the terminal symbols correspond to predefined lexical categories or special terms, as described above; see Figure 12.

The process of knowledge acquisition was done by analyzing the behavior of real data, in this case, the requirements and test cases available in the domain. First, we tried to follow the theory of the phrase structure grammar, initially introduced by Noam Chomsky [35], which assumes a binary division of the clause into a noun phrase and a verb phrase. However, after analyzing some cases, we saw that not always an action was associated with a condition. So we allow both possibilities.

The grammar start symbol is *Requirement*, which consists of a list of actions (*ActionList*). *ActionList* may have one or more *Action*; to separate an action from another, we use *COMMA* and the *AND* keyword. An *Action* can be a *ConditionalAction* or an *ImperativeAction*. A *ConditionalAction* comprises an *IFConditionalAction* (we use an if-then-else clause) or a *CaseConditionalAction* (we included a case structure to avoid nested if statements).

An *IFConditionalAction* term is composed of a conjunction *IF*, a list of conditions (*ConditionalActionList*), it followed by terminal symbol *THEN*, and a list of action (*ImperativeActionList*) which must be executed in case the conditions are true. Specifying which actions must be executed in case the conditions are false is optional, and must be done using the special *ELSE* symbol followed by another list of actions (*ImperativeActionList*). Finally, one needs to write *END* to end the statement.

The structure of *ConditionalActionList* and *ImperativeActionList* is similar to a Conjunctive Normal Form (CNF) – conjunction of disjunctions. The conjunctions are delimited by a *COMMA* and the *AND* keyword, whereas the disjunctions are delimited by the *OR* keyword.

The elementary condition (*ConditionalClause*) comprises a *NounPhrase* (one or more nouns, including generic words, eventually preceded by a determiner and adjectives) and a *VerbPhraseCondition* (*VerbComparative* term followed by *VerbComplement*).

Requirement	→ ActionList;
ActionList	→ Action COMMA AND ActionList   Action;
Action	→ (ConditionalAction   ImperativeAction);
ConditionalAction	→ (IFConditionalAction   CaseConditionalAction);
CaseConditionalAction	→ CASE NounPhrase VerbComparative COLON (CaseClause)+ END;
CaseClause	→ VerbComplement THEN ImperativeActionList SEMICOLON;
IFConditionalAction	→ IF ConditionalActionList THEN ImperativeActionList (ELSE ImperativeActionList)? END;
ConditionalActionList	→ ConditionalActionList COMMA AND ConditionalOrClause   ConditionalOrClause;
ConditionalOrClause	→ ConditionalClause OR ConditionalOrClause   ConditionalClause;
ConditionalClause	→ NounPhrase VerbPhraseCondition;
VerbPhraseCondition	→ VerbComparative VerbComplement;
VerbComparative	→ VerbCondition NOT? ComparativeTerm?
VerbCondition	→ VerbCondition? (VPRE   VPRE3RD   VTOBE_PRE3   VTOBE_PRE   VTOBE_PAST   VTOBE_PAST3   VPAST   VPHRASAL PREP);
ComparativeTerm	→ (COMP (OR NOT? COMP)?);
ImperativeActionList	→ ImperativeAction COMMA AND ImperativeActionList   ImperativeAction;
ImperativeAction	→ ImperativeOrClause;
ImperativeOrClause	→ ImperativeClause OR ImperativeOrClause   ImperativeClause;
ImperativeClause	→ ((NounPhrase ModalVerb NOT?)   (DO NOT))? ImperativeVerbPhrase;
ImperativeVerbPhrase	→ VerbImperative VerbComplement;
ModalVerb	→ MAY   MUST;
VerbImperative	→ VBASE;
VerbComplement	→ VariableState? PrepositionalPhrase*;
VariableState	→ (ADV   ADJ   NUMBER   NounPhrase);
PrepositionalPhrase	→ PREP VariableState;
NounPhrase	→ DETER? ADJ* (Noun   QUOTATION_MARKS GENERIC QUOTATION_MARKS)+;
Noun	→ NSING   NPLUR;

Figure 12 – Grammar

The elementary condition (*ConditionalClause*) comprises a *NounPhrase* (one or more nouns, including generic words, eventually preceded by a determiner and adjectives) and a *VerbPhraseCondition* (*VerbComparative* term followed by *VerbComplement*).

A *VerbComparative* term is a *VerbCondition* (at least one verb: to be or any other in the present or past tense, including phrasal verbs) followed by an optional *NOT*, which negates the meaning of the next term, an optional *ComparativeTerm*. A *VerbComplement* is an optional *VariableState* (an adjective, an adverb, a number or a *NounPhrase*) followed by zero or more *PrepositionalPhrase* (a preposition and a *VariableState*).

The elementary action (*ImperativeClause*) begins with an option: it starts with a *NounPhrase* followed by a *ModalVerb* (*MAY*, to indicate possibility, and *MUST*, mandatory), and an optional *NOT* or it begins with *DO NOT* keyword. Both are proceeding by an

*ImperativeVerbPhrase* term, which is a *VerbImperative* (base form) followed by one *VerbComplement*.

A *CaseConditionalAction* begins with the *CASE* keyword, to signal this type of instruction, and a *NounPhrase* followed by a *VerbComparative* and a *COLON*, to indicate the phrase to be checked at next sentences (*CaseClause*). *VerbComparative* begins with a *VerbCondition*, and is followed by an optional *NOT*, which negates the meaning of the next term, an optional *ComparativeTerm*.

A *CaseClause* can be one or more occurrences. Each *CaseClause* has a *VerbComplement*, *THEN* keyword, followed by an *ImperativeActionList*. Each *CaseClause* is terminated with a *SEMICOLON*, and a *CaseConditionalAction*, with the *END* keyword.

Thus, we can write requirements using several different sentence formations. Below, we present a typical requirement rewritten to adhere to our CNL and its respective original form as it was written by the Motorola requirements team.

- *Original*: This feature is to extend the functionality of Settings - Sound & notification in order to allow user to set different ringtones for each SIM in Dual SIM devices.
- *Rewritten*: Open the Settings, AND  
CASE the phone has:  
one SIM card THEN set a ringtone to “SIM 1”, and make a call to “SIM 1”;  
two SIM cards THEN set a ringtone to “SIM 1”, and set a ringtone to “SIM 2”, and make a call to “SIM 1”, and make a call to “SIM 2”;  
no SIM card THEN the system must not show two options, and set a ringtone, and make a call;  
END.

As can be seen, the rewritten requirement needs more information than the original version, consequently demanding more effort from the project manager. However, from this, ATG is able to generate UCs and a comprehensive test suite automatically.

As another interesting facility, the proposed CNL allows reuse of terms using aliases. The usage of aliases can be an alternative to avoid abbreviations, as we suggested previously.

To exemplify this, we create an alias (e.g., *CALL\_SIM1*) to refer an action, which appears repetitively in the above requirement. To use the alias one just needs to refer to the alias name within the requirement, as shown below.

Kaki user interface allows inserting an alias, as we explain in Section 4.1. Let *CALL\_SIM1* refer to *and make a call to “SIM 1”*, then the requirement can be rewritten as follows.

- Open the Settings, AND  
CASE the phone has:  
one SIM card THEN set a ringtone to “SIM 1”, *CALL\_SIM1*;  
two SIM cards THEN set a ringtone to “SIM 1”, and set a ringtone to “SIM 2”, *CALL\_SIM1*, and make a call to “SIM 2”;  
no SIM card THEN the system must not show two options, and set a ringtone, and make a call;

END.

Despite the reuse benefits, this feature is optional. Also, our parser is not aware of aliases; there is a pre-processing of the input and the parser receives the requirements without this.

### 3.2 EXTRACTION ALGORITHM

The result of parsing the requirements in the CNL is a syntactic tree, which is the input to the extraction algorithm. We are using the visitor design pattern [36] to analyze the syntax tree. Algorithm 1 is introduced as a pseudo-code to explain how we extract the information. Basically, we need to automatically create use cases with their respective main and alternative flows because the TaRGeT input are use cases. Each flow has *fromStep* and *toStep* fields, indicating in which step the flow begins and finishes, respectively. For the main flow, it is always *START* and *END*. For alternative flows, they are calculated at runtime.

The first command from the extraction algorithm is finding all actions from syntax tree. Next, there is a verification for each action to identify the particular type of action: *ImperativeAction* or *ConditionalAction*.

Besides, the algorithm extracts the predecessor operator from this action. This serve to decide whether the action belongs to the main flow or the alternative flow.

For any action we need to transform the verb to its base form, if the verb is not in infinitive mode. So, we create a step in active voice (to fill up Kaki's slots, operation, and patient) and passive voice.

The algorithm is slightly different for *ConditionalAction*. When a *ConditionalAction* is an *IFConditionalAction* the step has a condition. We consider the steps in *THENClause* as Main Flow actions, except those actions which have OR as a predecessor operator. As we explained in the previous section, the *ELSEClause* is optional. Thus, if *ELSEClause* exists, we consider the resulting steps as Alternative Flow. The resulting steps, we create a blank Alternative Flow with the negation of the condition, but with empty user action and system response.

---

#### Algorithm 1 – Extraction algorithm

---

1. find all actions from syntactic tree
  2. if the action is an *ImperativeAction*
    - a. verify verbal forms
    - b. create the step with their respective active and passive voice. Store the predecessor operator
      - i. if the operator is different of OR → Main Flow
      - ii. else → Secondary Flow
        - calculate *fromStep* and *toStep* fields
-

3. if the action is a ConditionalAction
  - a. verify which type of ConditionalAction it is
    - i. IFConditionalAction
      - select the condition
      - verify verbal forms from THEN clause
      - create the step with their respective active and passive voice. Store the predecessor operator
        - if the operator is different of OR → Main Flow
        - else → Alternative Flow
          - calculate *fromStep* and *toStep* fields
      - if ELSE clause exists
        - verify verbal forms from ELSE clause
        - create the step with their respective active and passive voice with the condition → Alternative Flow
      - else → create a blank Alternative Flow with condition denial
    - ii. CaseConditionalAction
      - for each CaseClause
        - mounted the full condition
        - create the step with their respective active and passive voice with the condition
          - if isFirst → Main Flow
          - else → Alternative Flow
        - calculate *fromStep* and *toStep* fields

When dealing with a *CaseConditionalAction*, we must complete the condition because it concatenates itself with the beginning of each *CaseClause*. Moreover, we consider the first *CaseClause* as Main Flow, and, the others, as Alternative Flows. Figure 13 shows the database structure.

```

UseCase = {
  id: Number,
  flows: [Flow]

  Flow = {
    id: Number,
    type: String,
    fromStep: String,
    toStep: String,
    steps: [Step]
  }

  Step = {
    id: Number,
    stepDescription: String,
    initialSetup: String,
    expectedResults: String
  }
}

```

Figure 13 – Database structure

A *UseCase* has an *id* (to identify each instance) and a list of *Flows*. Each *Flow* has an *id*, with the same purpose previously mentioned, a *type* to indicate which type of flow it is: main or alternative. Also, it has *fromStep* and *toStep* fields, and a list of *Steps*. In turn, a step has an *id*, *stepDescription* to indicate the action, an *initialSetup* for conditions, and *expectedResults* which is the system response.

Using the same example we can see how the information is stored in the database, as shown in Figure 14.

```
"UseCase":
{
  "id": 0,
  "flows": [
    {
      "id": 0,
      "type": "main",
      "fromStep": "START",
      "toStep": "END",
      "steps": [
        {
          "index": 0,
          "stepDescription": "Open the settings.",
          "initialSetup": "The phone has one SIM card",
          "expectedResults": "the settings was opened."
        },
        {
          "index": 1,
          "stepDescription": "set a ringtone to \"SIM 1\".",
          "initialSetup": "The phone has one SIM card",
          "expectedResults": "the ringtone was set."
        },
        {
          "index": 2,
          "stepDescription": "make a call to \"SIM 1\".",
          "initialSetup": "The phone has one SIM card",
          "expectedResults": "a call was made."
        }
      ]
    },
    {
      "id": 1,
      "type": "alternative",
      "fromStep": "1A",
      "toStep": "END",
      "steps": [
        {
          "index": 0,
          "stepDescription": "set a ringtone to \"SIM 1\".",
          "initialSetup": "The phone has two SIM cards",
          "expectedResults": "the ringtone was set."
        },
        {
          "index": 1,
          "stepDescription": "set a ringtone to \"SIM 2\".",
          "initialSetup": "The phone has two SIM cards",
          "expectedResults": "the ringtone was set."
        },
        {
          "index": 2,
          "stepDescription": "make a call to \"SIM 1\".",
          "initialSetup": "The phone has two SIM cards",
          "expectedResults": "a call was made."
        },
        {
          "index": 3,
```

```

        "stepDescription": "make a call to \"SIM 2\".",
        "initialSetup": "The phone has two SIM cards",
        "expectedResults": "a call was made."
    }
  ]
},
{
  "id": 2,
  "type": "alternative",
  "fromStep": "1A",
  "toStep": "END",
  "steps": [
    {
      "index": 0,
      "stepDescription": "Do not show two options.",
      "initialSetup": "The phone has no SIM card",
      "expectedResults": "two options was not shown."
    },
    {
      "index": 1,
      "stepDescription": "set a ringtone.",
      "initialSetup": "The phone has no SIM card",
      "expectedResults": "the ringtone was set."
    },
    {
      "index": 2,
      "stepDescription": "make a call to the tested phone.",
      "initialSetup": "The phone has no SIM card",
      "expectedResults": "a call was made."
    }
  ]
}
]
}
}

```

Figure 14 – Database example

### 3.3 CONSISTENCY AND DEPENDENCY ANALYSIS

In order to ensure that a sequence of test actions can be correctly executed, we integrate our work with Kaki's strategy. As we have previously mentioned in Section 2.2.3, an elementary *test action* has an operation and a patient. And, from the information extraction we can map Kaki's slots to use it, as shown in Table 4.

Table 4 – Mapping between ATG and Kaki

Syntax Tree from ATG	Kaki' slots
Verb in base form	Operation
NounPhrase, without DETER	Patient
VariableState into a PrepositionalPhrase or GENERIC associated with the noun	Extra

With this in mind, the algorithm verifies whether these slots still do not exist in the database, and enter as new Kaki slots. Following this, it defines that frame is valid. Also, if a noun has a generic associated term, a new slot is created, named generic. It is worth mentioning that the user can rename the extra slot.



- *ATG requirement*: Set a Ringtone to “SIM 1”
- *ATG Use Case*: Set a Ringtone with “SIM 1” as generic.
  - *Operation*: Set
  - *Patient*: Ringtone
  - *Extra (Generic)*: SIM 1

If the association between the actions (dependencies or cancellations) does not exist in the database, it is necessary to use the Kaki graphical interface to store them once we were not able to infer this from the requirements. In Section 4.1, we will show, step-by-step at how to do this.

Figure 15 illustrates how consistency and dependency analysis happen. In this example, the user forgets the action *Open the Settings* from requirement, and the system shows the correct sequence. The user decides whether to accept or ignore the suggestion. Then, we accomplish our second challenge: to supply consistency and dependency analysis for the CNL semantics in order to check individual instances and their relationships.

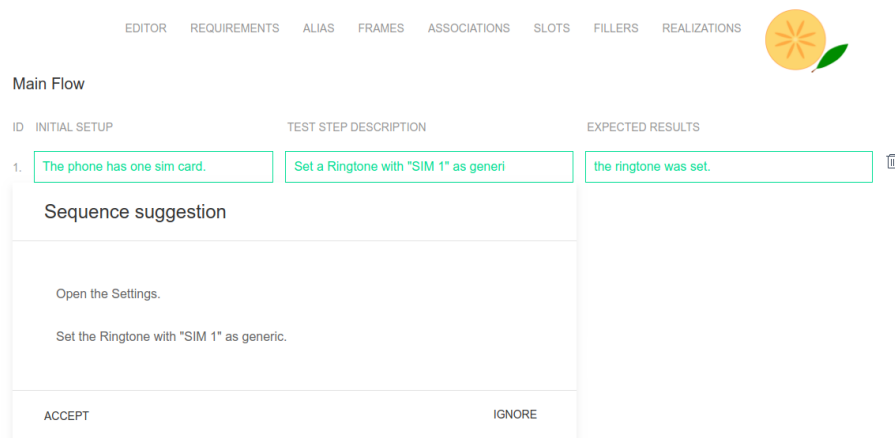


Figure 15 – Consistency and dependency analysis

### 3.4 AUTOMATIC TEST CASE GENERATION

TaRGeT is a powerful tool for test case generation from natural language, but it requires a constrained form of use cases described in a tabular form. In the current context of Motorola Mobility, instead of use cases, more abstract requirements are more commonly available. To mitigate the user's effort to follow the TaRGeT input model, the ATG automatically creates the input according to the information extracted by the extraction algorithm. TaRGeT has so many features, but we use only two main features, as shown in Figure 16.

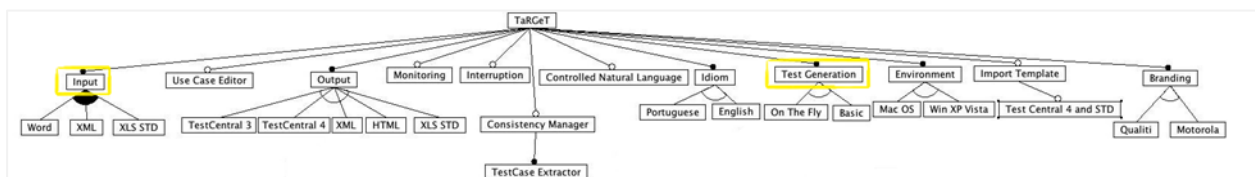


Figure 16 – TaRGeT's Feature Model

Source: [37]

Thus, TaRGeT can generate possible scenarios for that requirement, automatically, and we reach our third challenge. We show an example in Appendix 0 with input formats both in Word and in XML, besides the respective output.

### 3.5 TEST AUTOMATION

The integration with Zygon provides two benefits: test automation with potential reuse, corresponding to the challenge 4.

Due to the imposed standard to write requirements in the CNL, the TCs created automatically by TaRGeT are capable of automation using Zygon. If the test case step to be automated has similarity with existing ones in the database, then the implementation is reused using NLP techniques to combine natural language test steps with already automated test actions in the database.

According to [38] the tool can achieve a reuse rate of up to 70%. Figure 17 shows the corresponding test actions (on the right) for the test steps entered in the tool (left), for example.

1. Check the login procedure	TE - TC - POP: To Verify the Login Procedure of Gmail from MotoEmail for POP - Manual Login with regular procedure
2. Open Settings	Open Settings
3. Delete email account	Remove Email Account

**Figure 17 – Reusing action by matching similar descriptions**  
Source: [17]

If the test step does not exist in the database, it is possible to capture its execution, including using parameterization, as shown in Figure 18 and Figure 19. More details on how to use the tool can be found in [17].

**Figure 18 – Capture screen**

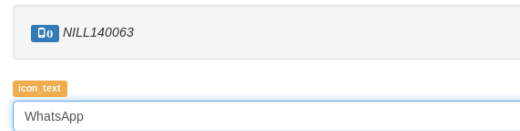


Figure 19 – Choose value to variable

### 3.6 JIRA INTEGRATION

Motorola Mobility projects are managed by the Jira platform, a modified instance called Dalek. In order not to introduce another tool to Motorola employees, we have created a strategy for integrating our project with this platform. In Dalek, it is common to develop a test suite from an XLS file. So we took the output of TaRGeT and generated an XLS file compatible with Jira. Figure 20 illustrates an example of this file. Finally, we have reached our last challenge.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Issue Id	Issue Type	Regression Level	Feature ID	summary	Component/s	Primary domain	Secondary domain	Labels	Initial Setup	Test Step Description	Expected Results
2		Test Case		Feature-5204	5204_MM_Func_001					1) The phone has one SIM card	*Step 1: select one ringtone. *Step 2: tap the button with description "ok". *Step 3: make a call to the tested phone.	*Step 1: the ringtone was selected. *Step 2: the button was tapped. *Step 3: a call was made.
3		Test Case		Feature-5204	5204_MM_Func_002					1) The phone has two SIM cards	*Step 1: select one ringtone. *Step 2: tap the label with description "SIM 2" *Step 3: select other ringtone. *Step 4: make a call to "SIM 1". *Step 5: make a call to "SIM 2".	*Step 1: the ringtone was selected. *Step 2: the label was tapped. *Step 3: other ringtone was selected. *Step 4: a call was made. *Step 5: a call was made.
4		Test Case		Feature-5204	5204_MM_Func_003					1) The phone has no SIM card	*Step 1: Do not show two options. *Step 2: select one ringtone. *Step 3: tap the button "ok" *Step 4: make a call to the tested phone.	*Step 1: two options was not shown. *Step 2: one ringtone was selected. *Step 3: the button was tapped. *Step 4: a call was made.

Figure 20 – XLS file

# 4

## TOOL SUPPORT

In this chapter we discuss how the strategy described in the previous chapter was designed and implemented. Our tool is developed as a Kaki's plugin; therefore, it uses the same architecture, as shown in Figure 21. We highlight the components which we need to implement new code. Other components we reuse from Kaki; more details on how they are implemented can be found in [39].

The ATG tool is written in JavaScript<sup>9</sup> (it is multi-platform), using Node.js<sup>10</sup>, an open source server environment. The GUI was built using a progressive framework, Vue.js<sup>11</sup>, which provides a declaratively render data to the DOM (acronym for Document Object Model, a standard for accessing valid HTML documents), i. e., the data and the DOM are now linked, and everything is now reactive.

To optimize the communication between client application and server application, we chose to use NGINX<sup>12</sup>, high-performance HTTP (HyperText Transfer Protocol) server and reverse proxy, and Express JS<sup>13</sup>, which provides a thin layer of fundamental web application features, without obscuring Node.js features. We use mongoDB<sup>14</sup> as database to store the data.

RabbitMQ<sup>15</sup> enables one to handle messaging traffic quickly and reliably, as well as being compatible with various programming languages, native administration interface and cross-platform.

The module responsible for doing the semantic analysis was implemented in Alloy [40], which receives instances according to the notation used to represent the processed syntax tree and maps it into an intermediate formalism to reason about its properties.

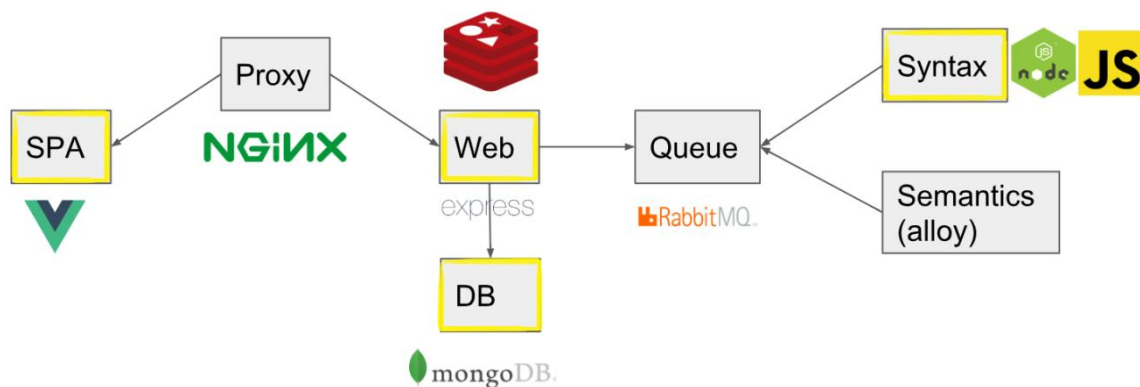


Figure 21 – ATG Architecture

Figure 22 shows the tool interface. In the upcoming subsections, we present how ATG works according to an example show previously, step by step.

<sup>9</sup> <https://www.javascript.com/>

<sup>10</sup> <https://nodejs.org>

<sup>11</sup> <https://vuejs.org/v2/guide/>

<sup>12</sup> <https://www.nginx.com/>

<sup>13</sup> <https://expressjs.com/>

<sup>14</sup> <https://www.mongodb.com/>

<sup>15</sup> <https://www.rabbitmq.com/>



NEW +

Feature ID

Feature Title

Type here...

TRANSFORM

Figure 22 – ATG tool

Each phase of the ATG strategy, presented in Section 3, is realized by a different component, which was modeled on a high-level process diagram using Business Process Model and Notation (BPMN) (see Figure 23).

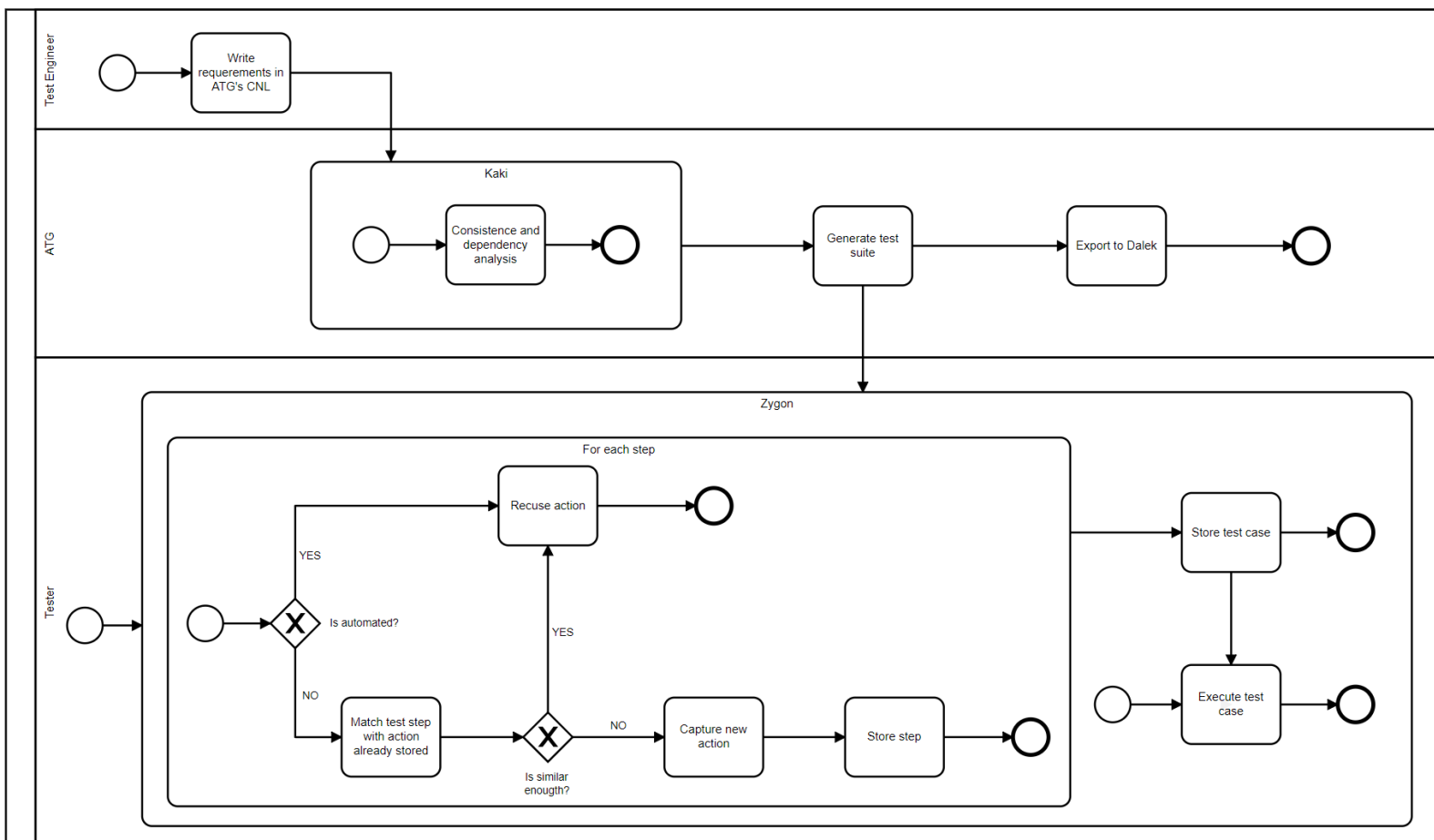


Figure 23 – ATG process

To guide the explanation of the tool operation, we detail each component at the user interface level, as well as its implementation/architecture options, as shown in Table 5.

Table 5 – Components details

ATG	<p><b>Write requirements in ATG's CNL</b> In this step the test engineer needs to write requirements in the CNL, as discussed in Section 3.1.</p> <p><b>Generate test suit</b> It is the core of our strategy, explained in Section 3.4. Only with a written standardized requirement, the tool can generate test suits automatically.</p> <p><b>Export to Dalek</b> The tool allows exporting an XLS file that can be imported into Dalek, see further details in Section 3.6.</p>
KAKI	<p><b>Consistence and dependency analysis</b> As each step of the test suite generated by the ATG is already compatible with Kaki, the tool can automatically check for consistency and derive the dependencies required to run without any human interference. Further details on how the mapping between the ATG and Kaki is done, as well as the process for checking the consistency and verifying the dependencies, are described in Section 3.3.</p>
ZYGON	<p><b>Match test step with an action already stored</b> It is necessary that texts similar to test actions already stored in the database be replaced by them since Kaki CNL was built on the concept of frames, which is a structure to store data about a previously known situation [31]. These frames contain prefixed slots that represent an instance of a specific action. Thus, we will be able to automatically, if they are finite and well defined, in pre-established commands or responses.</p> <p><b>Reuse Action</b> As one of the objectives of our strategy is the reuse, when searching for a test step, there is processing in the text to verify similarity, as described in Section 3.5.</p> <p><b>Capture new action</b> When the test step is not stored, or there is no similarity between other test steps already stored, we have adopted the C&amp;R strategy to perform the automation of the test step. It is an advantageous automation technique especially for people who do not know how to program.</p> <p><b>Store step</b> At the end of the capture, the user can save the test step and can be reused later.</p> <p><b>Store test case</b> A TC is composed of several actions, which in turn can be an elementary action or composed of other actions (see Figure 24). Each action/test step can be stored, as well as the whole case test can too. It is excellent, especially in a mobile context, since there is a diversity of devices to be tested, i.e., the same TC runs several times.</p>

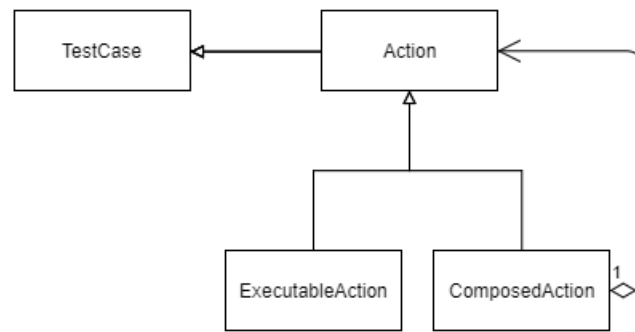


Figure 24 – TestCase structure

**Execute test case** The last component refers to the execution of the case test. This execution can be done soon after the capture of the test steps or *a posteriori*.

#### 4.1 PARSER

The Parser used in this work was the same one used by Gustavo Carvalho for NAT2TEST [25], with a few modifications. It was possible because a version of the Generalized LR (GLR) parsing algorithm [41] was implemented, which allows adapting the grammar and generation of an appropriate parser automatically, without any extra changes required in the code. Unlike the other modules, it was written in Java.

It is a context-free parser, which is responsible for parsing the list of requirements according to the grammar defined and explained in Section 3.1.2. However, before this analysis, it is necessary that each word is inserted in the dictionary, along with its grammatical class. Thus, it can be said that there is also a morphological analysis.

In linguistics, the result of the morphosyntactic analysis defines the POS categories. Similar to this, but in the area of natural language processing, a customization of the POS-Tagger algorithm was implemented in the parser [25]. For each possibility of a match with the rules, a syntax tree is constructed. For this reason, we cannot say that the language is entirely free of ambiguity since the generation of all possible trees allows for lexical ambiguity. For example, up can be a preposition, an adverb, an adjective or a verb. This is the big difference of the parser for a programming language compiler, where for each input only one output is generated, that is, it is deterministic.

In ATG, we use the parser so that as long as the requirement is not parsed correctly, there is no generation of syntax trees. When there is a change in the coloring of the text box and in the text itself (green), the user knows that the sentence is spelled correctly and can proceed, as shown in Figure 25. The user can also create multiple entries with requirements for the same feature, which are processed separately, but, in the end, the generated tests are unified. The *Feature Id* and *Feature Title* fields are required.

EDITOR REQUIREMENTS ALIAS FRAMES ASSOCIATIONS SLOTS FILLERS REALIZATIONS



NEW +

5204

Support different ringtones for each SIM in Dual SIM devices.

Open Settings, AND CASE the phone has:  
 one SIM card THEN set ringtone to "SIM 1", and make a call to "SIM 1";  
 two SIM cards THEN set ringtone to "SIM 1", and set ringtone to "SIM 2", and make a  
 call to "SIM 1", and make a call to "SIM 2";  
 no SIM card THEN the system must not show two options set ringtone, and make a  
 call; END.

TRANSFORM

**Figure 25 – Sentences well-formed**

When entering a word and giving space, the parser analyzes the text written so far and indicates the grammatical classes and/or special words of the expected grammar (see Figure 26), shown just below the input box.

EDITOR REQUIREMENTS ALIAS FRAMES ASSOCIATIONS SLOTS FILLERS REALIZATIONS



NEW +

5204

Support different ringtones for each SIM in Dual SIM devices.

Open the

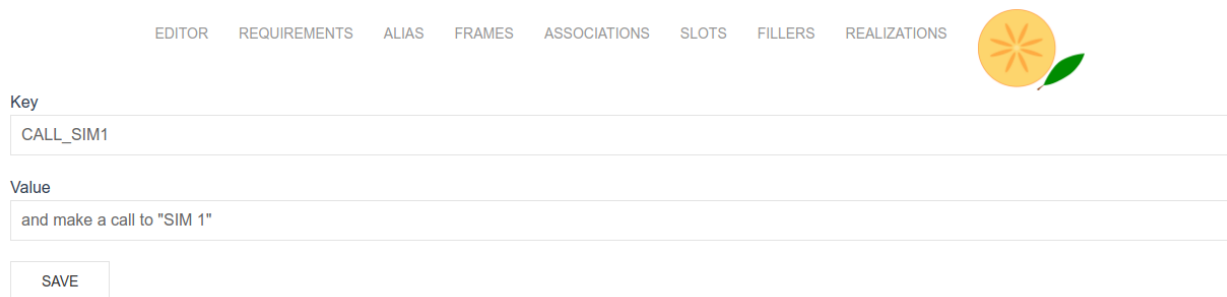
Quotation Marks  
 Adjective  
 Plural Noun  
 Singular Noun

**Figure 26 – Runtime Parser analysis**

If a word is not in the dictionary, the following error message is displayed: *The word 'test' is not defined in the lexicon.* This message appears in the same suggestions field.



As mentioned in Section 3.1.2, the user can create aliases or refer to them in the ALIAS tab (see Figure 27). It is a straightforward procedure: click on the *New* button, fill in the fields with a key and a value. Finally, click the *Save* button.



EDITOR REQUIREMENTS ALIAS FRAMES ASSOCIATIONS SLOTS FILLERS REALIZATIONS

Key

CALL\_SIM1

Value


and make a call to "SIM 1"

SAVE




Figure 27 – How create aliases

## 4.2 USE CASES

Once the requirements are green, one can press the *Transform* button. The following screen (see Figure 28) shows all automatically generated use cases with their respective test steps. Each step contains three fields: initial setup, test step description, and expected results. The initial setup field indicates the conditions that step is conditioned to. If it is empty, there are no conditions. The test step description field is the test action itself. The expected results field is generated automatically, so the tool transforms the test action to passive voice. Any of these fields can be edited, and the test action can be deleted.

EDITOR
REQUIREMENTS
ALIAS
FRAMES
ASSOCIATIONS
SLOTS
FILLERS
REALIZATIONS






### Main Flow

ID	INITIAL SETUP	TEST STEP DESCRIPTION	EXPECTED RESULTS	
1.		open the settings.	the settings was set.	
2.	The phone has one SIM card	set a ringtone to with "sim 1" as descr	a ringtone was set.	
3.	The phone has one SIM card	make a call to with "sim 1" as descript	a call was made.	

---

### Alternative Flow 1

FROM STEP:




ID	INITIAL SETUP	TEST STEP DESCRIPTION	EXPECTED RESULTS	
1.	The phone has two SIM cards	set a ringtone to "SIM 1".	a ringtone was set.	
2.	The phone has two SIM cards	set a ringtone to "SIM 2".	a ringtone was set.	
3.	The phone has two SIM cards	make a call to "SIM 1".	a call was made.	
4.	The phone has two SIM cards	make a call to "SIM 2".	a call was made.	

BACK TO STEP:

---

### Alternative Flow 2

FROM STEP:

ID	INITIAL SETUP	TEST STEP DESCRIPTION	EXPECTED RESULTS	
1.	The phone has no SIM card	Do not show two options.	two options was not shown.	
2.	The phone has no SIM card	set a ringtone.	a ringtone was set.	
3.	The phone has no SIM card	make a call to the tested phone.	a call was made.	

BACK TO STEP:

**Figure 28 - Use Cases screen**

Alternative flows contain the *from step* and *to step* fields that are computed at runtime, based on the connectives of the input requirement, as mentioned in Section 3.2. The value in *fromStep* indicates until which step the main flow will be executed to begin its alternative flow. When the value is *START*, it means that the test run will begin from the alternative flow. The value in *toStep* indicates to which main flow's step the alternative flow will return when it finishes. When the value is *END*, the execution of the test will not return to the main flow.

### 4.3 KAKI INTEGRATION

The test actions are extracted from the requirement(s) and mapped into the Kaki slots (Figure 29), and the respective frames are also created (Figure 30 – Created framesFigure 30).

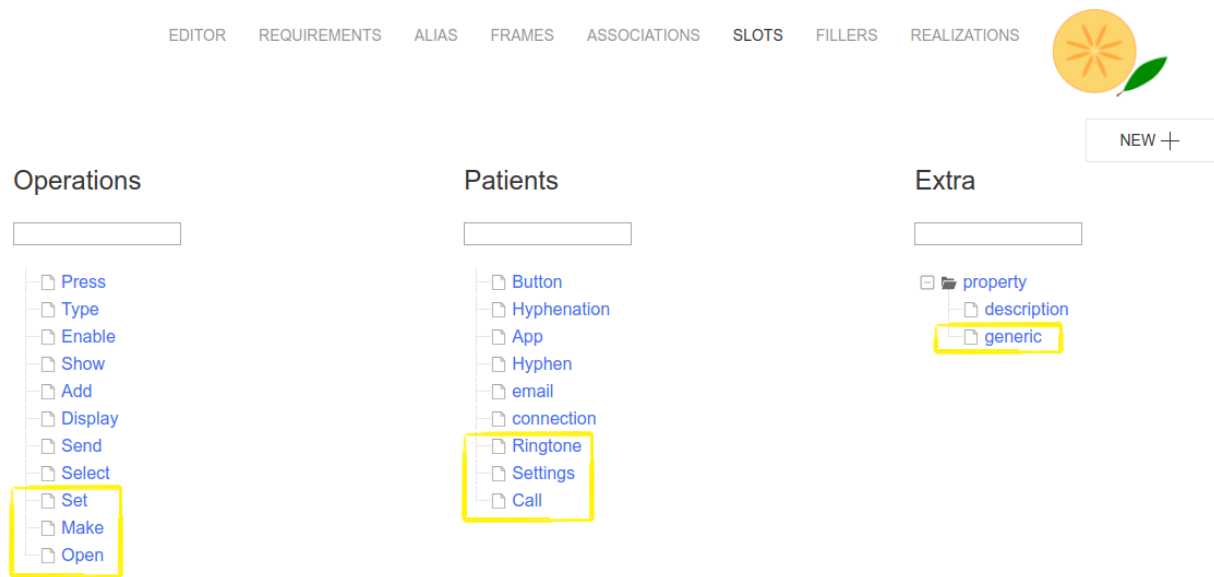


Figure 29 – Created slots

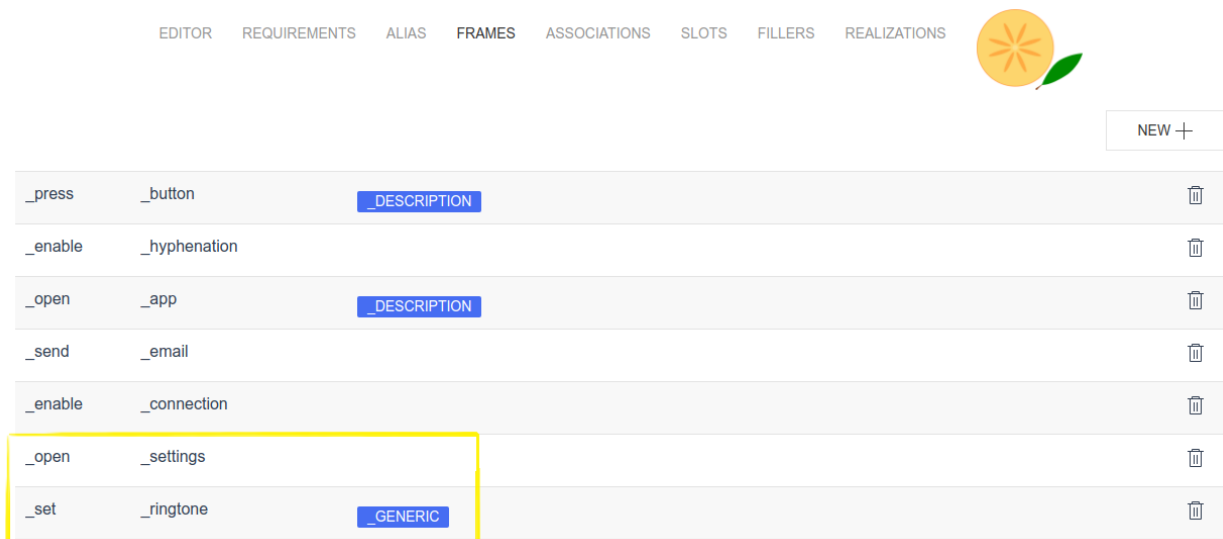


Figure 30 – Created frames

However, the associations (dependencies and cancellations) are not automatically registered. The following is a step-by-step guide on how to do this last step. It is worth noting that once the association is registered, it will serve for future interactions (see Figure 31).

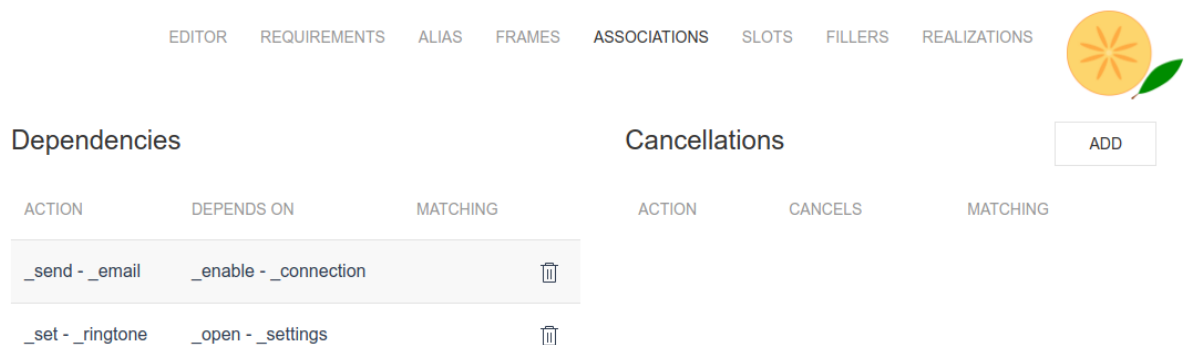


Figure 31 – Associations

When one clicks the *Add* button, a new screen will appear, and the user can choose whether to register a dependency (Figure 32) or a cancellation.

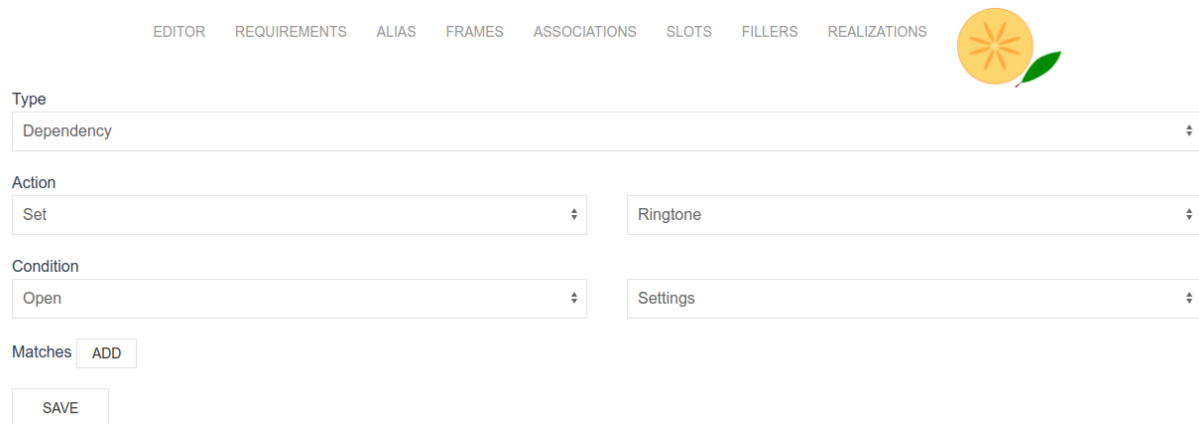


Figure 32 – Add a dependency

Once this is done in the Use Cases screen, the user can see all the use cases and, clicking on the *Save* button, besides saving the information in the database, it also activates the consistency and dependency analysis. If there is any inconsistency, a box with the tool suggestions below the test step will be shown, as already mentioned in Figure 15.

#### 4.4 TARGET INTEGRATION

At the bottom (right) of the Use Cases screen (see Figure 33), there is also the *Generate Test Cases* button. When clicked, the stored information of the UCs is used to create the entry of TaRGeT, transparently to the user. An example of the automatic created input to TaRGeT (XML file) is given in Figure 33 – Input TaRGeT

```
<?xml version="1.0" encoding="UTF-8"?>
<phone xmlns="user-view.target.v20071129">
  <feature>
    <featureId>5204</featureId>
    <name>Support different ringtones for each SIM in Dual SIM devices</name>
    <useCase>
      <id>UC_01</id>
      <name></name>
      <description></description>
      <setup></setup>
      <flow>
        <description></description>
        <fromSteps>START</fromSteps>
        <toSteps>END</toSteps>
        <step>
          <stepId>1A</stepId>
          <action>Open the Settings.</action>
          <condition></condition>
          <response>the settings was opened.</response>
        </step>
      </flow>
    </useCase>
  </feature>
</phone>
```

```

        <stepId>2A</stepId>
        <action>set a ringtone to "SIM 1"</action>
        <condition>the phone has one SIM card</condition>
        <response>the ringtone was set.</response>
    </step>
    [...]
</flow>
<flow>
    <description></description>
    <fromSteps>1A</fromSteps>
    <toSteps>END</toSteps>
    <step>
        <stepId>1B</stepId>
        <action>set a ringtone to "SIM 1".</action>
        <condition>the phone has two SIM cards</condition>
        <response>the ringtone was set.</response>
    </step>
    <step>
        <stepId>2B</stepId>
        <action>set a ringtone to "SIM 2".</action>
        <condition>the phone has two SIM cards</condition>
        <response>the ringtone was set.</response>
    </step>
    [...]
</flow>
<flow>
    <description></description>
    <fromSteps>1A</fromSteps>
    <toSteps>END</toSteps>
    <step>
        <stepId>1C</stepId>
        <action>Do not show two options.</action>
        <condition>the phone has no SIM card</condition>
        <response>two option was not shown.</response>
    </step>
    [...]
</flow>
</useCase>
</feature>
</phone>

```

Figure 33 – Input TaRGeT

ATG runs TaRGeT in background TaRGeT with this input, and the TCs are generated. The output of TaRGeT was modified to a JavaScript Object Notation (JSON) to facilitate processing of the data as well as its display since the project was written in JavaScript.

## 4.5 TEST CASES

Finally, the test cases are displayed on the screen (see Figure 34). From there it is possible to do two activities: (I) export an XLS file, which is to export to Dalek; (II) automate the TCs using Zygon.

EDITOR

REQUIREMENTS

ALIAS


FRAMES

ASSOCIATIONS

SLOTS

FILLERS

REALIZATIONS



TEST CASE NAME:

5204\_MM\_Func\_001

CONDITIONS:

1) THE PHONE HAS ONE SIM CARD

ID	TEST STEP DESCRIPTION	EXPECTED RESULTS
1	open the settings.	the settings was opened.
2	set a ringtone to "SIM 1"	a ringtone was set.
3	make a call to "SIM 1".	a call was made.

dual sim x settings

settings

TEST CASE NAME:

5204\_MM\_Func\_002

CONDITIONS:

1) THE PHONE HAS TWO SIM CARDS

ID	TEST STEP DESCRIPTION	EXPECTED RESULTS
1	open the settings.	the settings was opened.
2	select a ringtone to "SIM 1".	a ringtone was set.
3	select a ringtone to "SIM 2".	a ringtone was set.
4	make a call to "SIM 1".	a call was made.
5	make a call to "SIM 2".	a call was made.

Type a label...

TEST CASE NAME:

5204\_MM\_Func\_003

CONDITIONS:

1) THE PHONE HAS NO SIM CARD

Type a label...

EXPORT TO XLS

Figure 34 – Test cases screen

Underneath each case test there is a *Type a label* field. It can be filled with new labels or reuse old ones and serves to signal which areas belong to that test and filters future searches.

# 5

## EXPERIMENTAL EVALUATION

This chapter presents how we conducted the planning of an experimental evaluation to measure the feasibility and effectiveness of using our strategy to generate test cases from requirements written in natural language (Section 5.1) as well as the results obtained (Section 5.1.4). Section 5.3 shows the threats to the validity of our experimental study.

### 5.1 PLANNING

We followed the planning model described in [42] to evaluate our work. Some experimental evaluations were conducted with the purpose of validating whether it is possible to write requirements using the CNL proposed by us and if the generated test cases cover those manually generated, and, finally, whether it is possible to automate these test cases using Zygon. In this way, we compare the use of ATG with our particular scenario (Motorola project).

Besides, we performed a tool usability analysis with Motorola Mobility employees. After discussing the objectives of the ATG's evaluation, we present some research questions with their respective metrics.

#### 5.1.1 Definition

The objective of this evaluation was structured according to the GQM (Goal, Question and Metric) approach [43], described below:

The main purpose is to analyze the practical use of ATG to generate test cases from requirements, comparing the manual creation of test cases and the automatic creation using the tool, concerning the impact on the process of automation of tests. From the point of view of the test engineers, the evaluation also involves analysis respect to the ease of use of the tool, as well as the ability to generate test cases automatically in the Motorola Mobility context.

#### 5.1.2 Ethical Concerns

It is important to report that this research take care of the ethical issues, guaranteeing the rights of the participants, always being guided by Resolution 466/12 of the *Conselho Nacional de Saúde* (CNS, Brazilian National Health Council).

General and specific information are on the Consent Form (Appendix 0), which deals with permission and use of captured data, formalization of study participation, study objectives, investigators, procedures, data collection, confidentiality of records, risks and/or discomforts, costs and declaration of consent.

The evaluation has been carried out by collaborators of the CIn-UFPE/Motorola cooperation project. Participation in the experimental evaluation is voluntary and participants and may request that their data not be used for analysis.

### 5.1.3 Research questions

The research questions that the present paper tries to answer are the following:

- [Q1] Is it feasible to write requirements using the CNL defined by us using the GUI?
  - Metric I: Number of questions to specialist consultations after training.
  - Metric II: Rate acceptance of interface usability.
- [Q2] Does our approach generate test steps that are automatable?
  - Metric III: Rate of automated test steps using Zygon.
- [Q3] Is there a reduction in time for generating test cases?
  - Metric IV: Average of time spent to create a set of TCs disregarding the execution and preparation time.
- [Q4] Is the strategy able to automatically generate the manually designed test cases?
  - Metric V: Percentage of generated tests with respect to the total number of manually designed tests.

### 5.1.4 Participants

Fifteen (15) Motorola collaborators from the CIn-UFPE/Motorola project and a collaborator from Motorola de Jaguariúna participated in this study. The evaluations were carried out in person in the technical areas of the project, and remotely (from Jaguariúna), from December 2018 to January 2019.

### 5.1.5 Procedures and data collection

To carry out this evaluation, the participants underwent training (slide show and tool demo) to use the Auto Test Generator tool, in person in Recife and remotely for the Jaguariúna collaborator.

The experimental evaluation occurred in one of the technical areas of Motorola of the Center of Informatics (CIn) of the Federal University of Pernambuco (UFPE) and in Motorola Mobility of the city of Jaguariúna. Each participant used a computer, with ATG installed. As a guideline for the accomplishment of the experiment, each participant has: (I) a script describing the experimental aspects to be understood and the scenario of the tool to be explored and (II) the slides used in training. A period of 30 minutes was stipulated to read the roadmap and its specification before beginning the execution of the evaluations and then responding to the questionnaire. The application of the experimental evaluation was monitored and guided by a researcher.

The data of the analyzed variables are collected through the questionnaire and notes during the execution of the experimental evaluations. Also, Jira information (requirements and test cases) are summarized.



## 5.2 EXECUTION AND RESULTS

To obtain metrics associated with the research questions, we present a description of how each stage on experimental evaluation was performed and the respective results are detailed as follows. In the first stage on experimental evaluation, a Motorola test engineer rewrote two requirements to become adherent to our CNL. In the second stage, in addition to these two requirements, we considered five others.

### 5.2.1 Experimental evaluation – stage I

The first experimental evaluation was conducted to verify whether using the CNL by test engineers to write requirements would be feasible. Also, with this experimental evaluation, we were able to compare the time spent by the software engineer to write the test suite using the tool with the time spent by the engineer to create the same suite without the ATG. These requirements were chosen according to the following criteria:

- They must be real requirements;
- They had to have previously created test cases;
- They had to be related to features to run in any environment, such as not being specific to the US carrier or needing to use some Bluetooth accessory. Just to make test execution easier.

The (original) requirements chosen to be rewritten were:

- [RQ001]: *This feature is to extend the functionality of Settings – Sound & notification in order to allow user to set different ringtones for each SIM in Dual SIM devices.*
- [RQ002]: *Enable user to directly attach an audio file to a message through the AOSP messaging app. Must support MP3, MP4, 3gp, and .aac formats.*

Right after the choice of requirements, we perform a training with a Motorola test engineer on how to use the ATG. This had to be done remotely, because the person available to help us works in São Paulo. The training begins with the explanation of the purpose of the tool, and then we perform a straightforward example to demonstrate how to use the graphical interface. This training lasts approximately 30 minutes.

After this step, we requested that he recorded the time to rewrite the requirements. From this we were able to collect the metrics I and IV because we counted how many times, after the training, we were asked for some explanation about the CNL during the rewrite of the requirements. This experimental evaluation also contributes data to metrics III and V.

In addition, we describe the settings of the computer with which the experimental evaluation was performed, as shown in Table 6.

Table 6 – Experimental evaluation I - Computer configuration	
Operating system	Windows 10
Processor	Intel i7-6500U 2.5GHz
Memory (RAM)	8GB

### 5.2.2 Experimental evaluation – stage II

Feedback on the usability of the graphical interface of our tool was essential for us. So, for the primary purpose of evaluating the graphical interface, we set up an environment (see Table 7) and requested that fifteen Motorola employees (CIn/UFPE) use the tool, nine from these professionals have used a tool to automate test.

**Table 7 – Experimental evaluation II - Computer configuration**

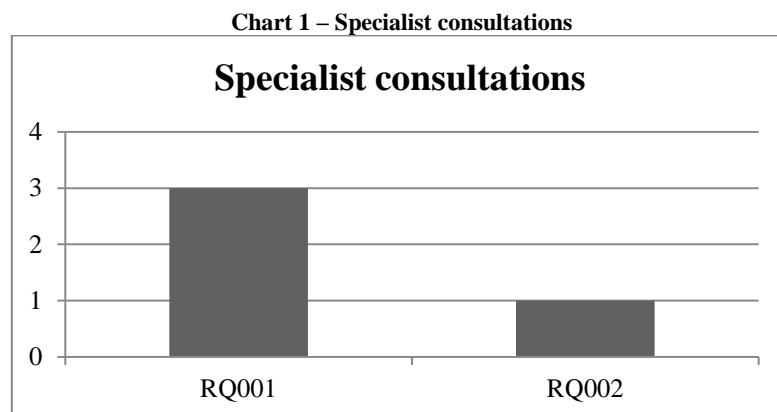
Operating system	Fedora 28
Processor	Intel i7-6500U 2.5GHz
Memory (RAM)	8GB

To obtain metric II, after the use of the tool, each participant was asked to complete a questionnaire. A template similar to this questionnaire is in Appendix 0 and available at this [link](#).

### 5.2.3 Results

Conducting these experimental evaluations, we have achieved some favorable and unfavorable results in our strategy.

The first experimental evaluation provides us with information for measuring metrics I and IV. First, the test engineer rewrites RQ001, and he made us three queries to be able to do it. Already for RQ0002, only one was made. The engineer told us that he felt more comfortable writing the second requirement. Chart 1 below illustrates the metric I.



As reported in the previous section, we asked the test engineer to record the time he needed to rewrite RQ001 and RQ002, with similar complexity. Besides, we analyze the number of test cases generated. Thus, it is possible to obtain the average of time spent to create a set of TCs disregarding the execution and preparation time (metric IV).

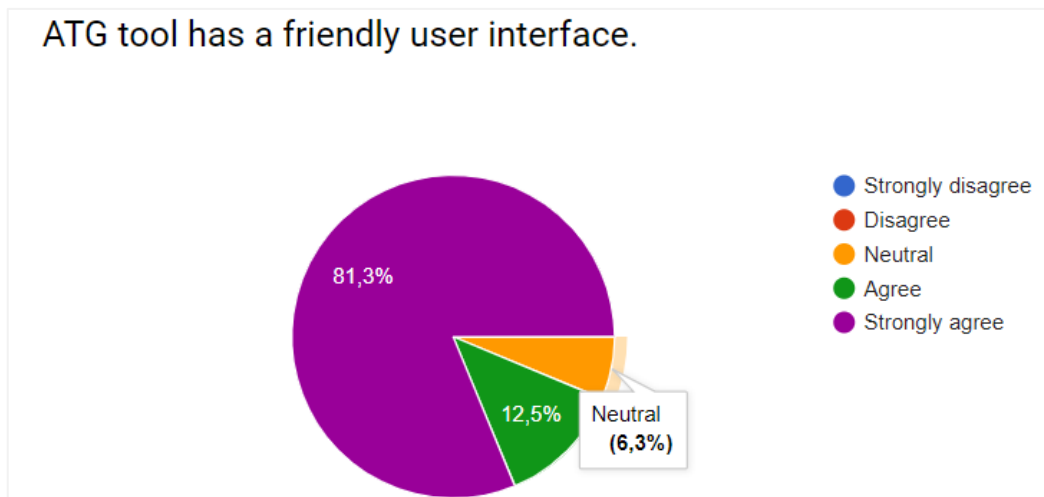
The rewriting of RQ001 and RQ002 (which produced 11 test cases using ATG) has taken approximately 10 hours. From this information, we can calculate the average of creating each test case ( $A_{ATG}$ ) which is approximately 0,9h.

The Motorola test team has provided us with a spreadsheet containing all test cases created by the team from May 2018 to January 2019, as well as the time it takes to create them. The creation of this worksheet with 215 test cases has taken 172,5h and from this information, we can calculate the average ( $A_{manual}$ ) that is approximately 0,8h.

Comparing  $A_{\text{manual}}$  with  $A_{\text{ATG}}$ , we perceive that they are quite close. That is, the difference is not significant because it is only 6 minutes. It should be noted that the initial effort may be similar, but our strategy standardizes the requirements as well as generates automated test cases.

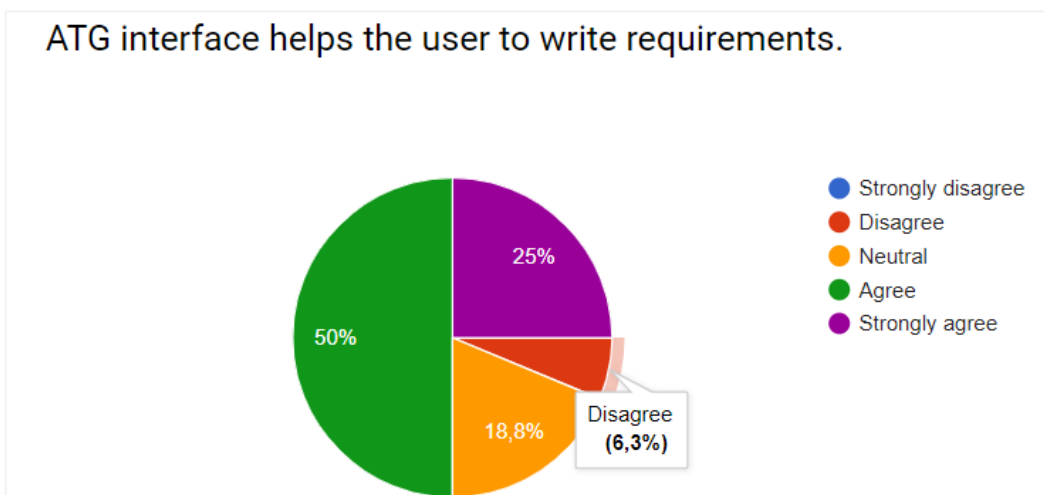
With metric II we evaluated the graphical interface of the tool and got good feedback, as can be seen in Chart 2. There were no negative responses about the tool's interface and more than 80% of the collaborators strongly agreed that the user interface is friendly.

Chart 2 – ATG tool has a friendly user interface



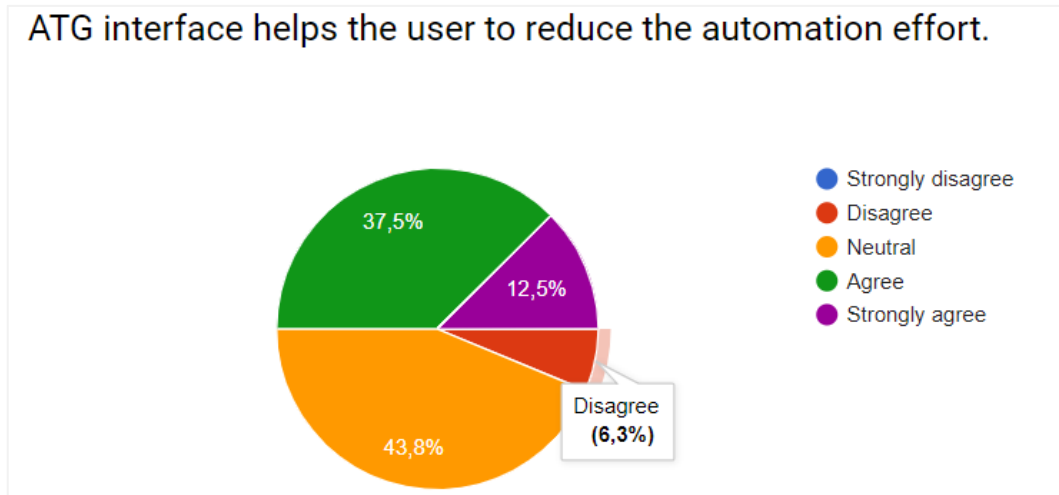
Another good feedback on the usability of the tool is that 75% agree that the ATG interface helps the user to write requirements through the suggestions of the next grammatical classes and/or expected special words, as shown in Chart 3.

Chart 3 – ATG interface helps the user to write requirements



One contributor who disagreed with the statement suggested that a step-by-step tutorial is done. Two others whom both answered the statement with *Neutral* suggested that words be added to the dictionary through the graphical interface. Only half the contributors said there was a reduction in the effort made for automation (see Chart 4). We assume that this is because the initial effort to write the requirement following in a standardized way is similar to the effort to write the test cases associated with this requirement and could be confirmed by the metric IV analysis.

Chart 4 – ATG interface helps the user to reduce the automation effort



However, in general, the graphical interface of the ATG has had positive feedback, needing improvements that we will take into account in our future work.

One of the most important metrics, metric III, validates the number of generated test steps that are automated. Seven requirements were analyzed with 34 test cases in total. From these TCs we counted 151 test steps and only 20 steps could not be using Zygon, i.e., about 86% of the steps were automated. An example of a step that one cannot do is *Resize the widget* because high-complexity gestures are required to perform this type of action. Even a person with little skill feels he may feel difficulty performing this action with his own hands.

Finally, with the metric V, we evaluate if the test cases generated by the tool covered the cases of manual tests previously created. All test cases have been covered, and two additional test cases were generated by ATG, not foreseen by test engineers.

### 5.3 THREATS TO VALIDITY

A concern in any experiment is whether its results are valid. The results are considered to be adequately validated if they apply to the population they want to generalize. The subsections below detail the different types of threats to validity applied to the proposed experimental evaluation and ways to mitigate them.

Regarding the rewriting time of the requirements and the number of queries made to the expert can vary greatly, since previous experiences (learning effect) can interfere in the experiment. Soon, a more experienced test engineer can shorten the rewrite time of the requirements, much like a less experienced test engineer can increase it. Thus, the average TCs/hour and the number of queries may be different. However, we believe that such values will be similar if the level of experience is also similar. In subsequent experiments, we intend to involve more participants and divide them by trial time with tests, in order to evaluate a more realistic scenario composed of results of a larger group.

We perform the analysis with artifacts and personnel of a real project. However, the number of test cases associated with each requirement may vary from project to project. All requirements evaluated could be rewritten, but it might be the case that this is not possible. To mitigate this type of threat, in subsequent experiments, we will increase the number of requirements analyzed.

It is worth mentioning, however, that we only consider tests for the Android<sup>16</sup> platform that could be automated only by interactions on the screen and visually visible.

The conclusion validity is related to the existing challenges to generate a valid conclusion from the relationship between treatment and experiment results. Among other factors, the conclusion validity involves a correct analysis and statistical interpretation of the results, the reliability of the measurements, the heterogeneity of the subjects, among others. We made our evaluation of the usability of the tool with professionals who belong to the same context (all work and/or research in the field of computing), so this can interfere with most of the information. However, the tool was developed for use by test engineers, that is, they are also in the field of computer science.

Threats related to construct validity arise due to human factors, such as incorrect behaviors on the side of participants in general. Timing of the process is a crucial factor in this study. Some guidelines are passed on to the participants, such as moments to start, pause and stop the stopwatch. If the timer is not activated/deactivated the whole experiment will be repeated. During the execution of this experimental evaluation there may be a malfunction both by the installed software and by the machines, which will compromise the end of the experiment. There were no reports of these types of threats at the time of the execution of the experimental evaluations so they did not need to be repeated.

---

<sup>16</sup> <https://www.android.com/>

# 6

## CONCLUSIONS

In this dissertation, we presented the Auto Test Generator strategy and tool to automatically generate test cases from requirements written in natural language, dispensing the need to learn specific notations. Yet, the strategy is based on a controlled natural language, thus reducing ambiguity and standardizing the text. The primary goal of this work was to improve the testing process as a whole, from the requirements writing by test designers to test scripts encoding by developers.

It is known that quality is an increasing core issue in companies, including software industry. These companies maintain specific teams in charge of executing processes that guarantee software quality, among which we highlight software testing. The testing processes and the related artifacts (e.g., test cases) are central to verify the products quality. Test cases are guides in the software verification process, being more critical than other technical documents, since incorrect interpretations may entail risks for the testing process and for the users. Therefore, well-written test cases are essential artifacts in the products quality assurance process.

In this light, ATG offers a strategy for the generation of clear and unambiguous test cases, which are derived from the CNL based requirements. Users are able to write requirements using a CNL specially designed for requirements, counting on a restricted vocabulary and pre-defined grammar formations. Use cases are then automatically derived based on the syntactically correct requirements. Following, these use cases are given as input to the TaRGeT tool, which automatically generates test cases for a variety of scenarios. Note that the test case generation process is transparent to the user, who does not need to get involved with more complex formal specifications.

Unlike other test automation tools, ATG provides dependency and consistency analysis between steps of a test case, and suggests consistent user sequences. Since no other approach mentions similar functionality, we could not compare ATG with other tools with respect to this aspect.

It is worth mentioning that the generated test cases may be automated using Zygon tool. We evaluated our proposal regarding automated test steps considering real examples from our industrial partner Motorola Mobility. As expected, from the selected input requirements, we were able to create TCs with more than 90% of automated test steps using Zygon.

In summary, our contributions include conceptual and design results, software implementation, and empirical assessments. As such, we believe that this research has succeeded in achieving its goal by providing an answer to its primary research question: how to automatically generate test cases from natural language requirements and, in particular, consistent and automated test cases. Following, we point out possible future work.

### 6.1 FUTURE WORK

Our work creates opportunity for several future research directions and improvements.

**Improve integration with TaRGeT** As mentioned in Section 3.4, TaRGeT has several features, among which, we highlight the parameterization. This functionality would be helpful to increase the reuse of the text of the requirements. For example, if the requirement states that a specific functionality should work with a list of applications, the name of those applications could be parametrized. Another exciting feature of TaRGeT is the use of filters for generating test cases. Those filters allow the user to select the test cases according to different criteria, such as requirements, use cases, the purpose of the test and similarity of the test cases. Those filters are very useful since, due to time or budget constraints, it is often not possible to run all the generated tests. More information about TaRGeT can be found in [44].

**Provide integration to other existing tools** Auto Test Plan (ATP) and Auto Test Coverage (ATC) [45] are tools developed within the in Motorola Mobility partnership as well. The ATP determines weights and criteria that are relevant for prioritizing the test cases for a regression campaign through the use of the Z3 solver, based on the historical data of the test cases. ATC is a tool to obtain code coverage on Android devices, without the use of source code instrumentation, through CPU profilers. Promoting the integration of these tools with the ATG would promote several benefits, such as providing guided exploratory tests a guided and more elaborate approach.

**Improve user interfaces** Some improvements to the current ATG user interface are listed below: (I) allow the insertion of words in the dictionary on the home screen; (II) allow insertion or exclusion of alternative flows in Use Cases screen; (III) provide an option to undo the exclusion of test steps; (IV) implement an autocomplete for aliases; (V) persistence of use cases.

**Agreement and regency analysis** Our grammar is not capable of performing an agreement and regency analysis between words, which would be beneficial to identify whether the regency of a given verb requires preposition or not, and in the former case, which prepositions can follow the verb. The dependency between article and noun involves the choice between *a* and *an*, depending on the first letter of the noun that follows the article. In turn, the regency analysis would help to identify which prepositions can follow a given transitive verb.

**Perform more experiments** We intend to consider more requirements, particularly involving Motorola's test engineers and design a controlled experiment to compare the current Motorola practice with. This is an important step towards effectively deploying the tool.

## REFERENCES

1. **Globo Cultura**. Disponível em: <<https://oglobo.globo.com/cultura/ariano-suassuna-tarefa-de-viver-dura-mas-fascinante-9343371>>. Acesso em: 31 Março 2019.
2. SCANNIELLO, G. et al. On the Effect of Using SysML Requirement Diagrams to Comprehend Requirements: Results from Two Controlled Experiments. **Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)**, 2014.
3. SOMMERVILLE, I. **Engenharia de Software**. 6ª. ed. [S.l.]: Addison Wesley, 2003. ISBN 85-88639-07-6.
4. OMG - Object Management Group. Disponível em: <<https://www.omg.org/>>. Acesso em: fevereiro 2019.
5. BITTNER. **Use Case Modeling**. Boston, MA, USA.: Addison-Wesley Longman Publishing Co., Inc., 2002.
6. HEUMANN. Generating test cases from use cases. **The rational edge**, v. 6, n. 1, 2001.
7. RAMLER, R.; WOLFMAIER, K. **Economic perspectives in test automation - balancing automated and manual testing with opportunity cost**. Proceedings of the 2006 International workshop on Automation of Software Test (AST). New York, NY, USA: ACM. 2006. p. 85-91.
8. BURNSTEIN, I. Practical software testing: a process-oriented approach. **Springer Science & Business Media**, 2003.
9. BALCER, M.; HASLING, W.; OSTRAND, T. Automatic Generation of Test Scripts from Formal Test Specifications. **ACM**, New York, NY, USA, v. 14, n. 8, p. 210–218, 1989.
10. BERTOLINO. **Software testing research: Achievements, challenges, dreams**. 2007 Future of Software Engineering (FOSE'07). [S.l.]: IEEE Computer Society. 2007. p. 85–103.
11. LEUNG, H. K.; WHITE, L. **Insights into regression testing [software testing]**. Software Maintenance, 1989., Proceedings., Conference on. [S.l.]: IEEE. 1989. p. 60–69.
12. GRAVES, T. L. et al. An empirical study of regression test selection techniques. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v. 10, n. 2, p. 184–208, 2001.
13. CHANDRA, R. et al. **Towards Scalable Automated Mobile App Testing**. Technical Report MSR-TR-2014-44. 2014.
14. LEOTTA, M. et al. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. **20th Working Conference on Reverse Engineering (WCRE)**, Koblenz, Germany, October 2013. 272–281.
15. HARTMAN, A. Model based test generation tools survey, AGEDIS Consortium, Tech. Rep., 2002.
16. BORBA, P. et al. TaRGeT - Test and Requirements Generation Tool. **Motorola's 2007 Innovation Conference (IC'2007), Software Expo Session**, Lombard, Illinois, USA, October 2007.
17. ARRUDA, F. M. C. **Test Automation from Natural Language with Reusable Capture & Replay and Consistency Analysis**. MA Thesis. Computer Science: UFPE. 2017.
18. SCHWITTER, R. English as a Formal Specification Language. **Proceedings of the 13th International Workshop on Database and Expert Systems Applications**, 2002.
19. BODDU, R. et al. RETNA: from Requirements to Testing in a Natural Way. **Proceedings of the RE**, p. 262-271, 2004.
20. SANTIAGO JUNIOR, V.; VIJAYKUMAR, N. L. Generating Model-based Test Cases from



- Natural Language Requirements for Space Application Software. **Software Quality Journal**, v. 20, p. 77-143, 2012.
21. ESSER, M.; STRUSS, P. Obtaining Models for Test Generation from Natural-Language like Functional Speci. **International Workshop on Principles of Diagnosis**, p. 75-82, 2007.
  22. BARROS, F. A. et al. The ucsCNL Tool: A Controlled Natural Language for Use Case Specifications. **Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011)**, p. 250-253, 2011.
  23. NOGUEIRA, S.; SAMPAIO, A.; MOTA, A. **Test generation from state based use case models**. Formal Aspects of Computing. [S.l.]: n.3. 2014. p. 441–490.
  24. WONG, E. et al. Dase: Document-Assisted Symbolic Execution for Improving Automated Software Testing. **IEEE/ACM 37TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING**, v. 1, p. 620–631, 2015.
  25. CARVALHO, G. et al. **NAT2TEST Tool: From Natural Language Requirements to Test Cases Based on CSP**. Software Engineering and Formal Methods. [S.l.]: [s.n.]. 2015. p. 283-290.
  26. FILLMORE, C. J. **The Case for Case**. In: BACH; HARMS (Ed.). Universals in Linguistic Theory. New York: Holt, Rinehart, and Winston. 1968. p. 1–88.
  27. CARVALHO, G. et al. NAT2TESTSCR: test case generation from natural language requirements based on scr specifications. **Science of Computer Programming**, v. 95, p. 275–297, 2014.
  28. CARVALHO, G. et al. **Model-Based Testing from Controlled Natural Language Requirements**. In: Artho, C., Ölveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, Communications in Computer and Information Science. Madrid: Springer. 2014. p. 19–35.
  29. CARVALHO, G.; SAMPAIO, A.; MOTA, A. **A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification**. In: Formal Methods and Software Engineering. In: Formal Methods and Software Engineering, LNCS. [S.l.]: Springer Berlin Heidelberg. 2013. p. 148–164.
  30. WYNNE, M.; HELLESØY, A. **The cucumber book: behaviour-driven development for testers and developers**. [S.l.]: Pragmatic Bookshelf, 2012.
  31. MINSKY, M. A framework for representing knowledge. **The Psychology of Computer Vision**, 1975.
  32. KUHN, T. A principled approach to grammars for controlled natural languages and predictive editors. **Journal of Logic, Language and Information**, v. 22, n. 1, p. 33–70, 2013.
  33. ALLEN, J. **Natural Language Understanding**. California: Benjamin/Cummings, 1995.
  34. CRYSTAL, D. **A Dictionary of Linguistics and Phonetics**. 6th. ed. [S.l.]: Wiley-Blackwell, 2008.
  35. CHOMSKY, N. **Aspects of the Theory of Syntax**. Cambridge, Massachusetts: MIT Press, 1965.
  36. FREEMAN, E. et al. **Head First Design Patterns: A Brain-Friendly Guide**. [S.l.]: O'Reilly Media, 2009.
  37. TARGET Product Line. Disponível em:  
<<https://twiki.cin.ufpe.br/twiki/bin/view/TestProductLines/TaRGeTProductLine>>. Acesso em: 10 jan. 2019.
  38. ARRUDA, F.; SAMPAIO, A.; BARROS, F. Capture & Replay with Text-Based Reuse and Framework Agnosticism. **Software Engineering and Knowledge Engineering (SEKE)**,

- San Francisco, California, USA, 2016. 420-425.
39. SAMPAIO, A.; ARRUDA, F. **Formal Testing from Natural Language in an Industrial Context**. Brazilian Symposium on Formal Methods (SBMF). Salvador: [s.n.]. 2016. p. 21-38.
  40. JACKSON, D. **Software Abstractions: logic, language, and analysis**. [S.l.]: MIT press, 2012.
  41. TOMITA, M. Efficient Parsing for Natural Language. **Kluwer Academic Publishers**, 1986.
  42. JURISTO, N.; MORENO, A. M. **Basics of Software Engineering Experimentation**. Norwell, MA, USA: Kluwer Academic Publishers, 2001.
  43. BASILI, V. R.; CALDEIRA, G.; ROMBACH, H. D. The Goal Question Metric Approach. **Encyclopedia of Software Engineering**, 1994. 528– 532.
  44. FERREIRA, F. et al. TaRGeT: a Model Based Product Line Testing Tool. **Congresso Brasileiro de Software (CBSOFT)**, Salvador, Bahia, Brasil, 2010.
  45. MAGALHÃES, C. et al. Evaluating an Automatic Text-based Test Case Selection using a Non-Instrumented Code Coverage Analysis. **Proceedings of the 2Nd Brazilian Symposium on Systematic and Automated Software Testing**, Fortaleza, 2017.

## APPENDIX A – TARGET EXAMPLE

Table 8 – Original input TaRGeT - Word

### Feature 1111 – My Phonebook

**OBS.:** The information presented here does not correspond to a real Motorola application. These use cases were only created in order to test the TaRGeT tool.

#### Use Cases

#### UC 01 – Creating a New Contact

##### Description

This use case describes the creation of a new contact in the contact list.

This is the use case main setup.

##### Main Flow

Description: Create a new contact.

From Step: START

To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_10 1]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

##### Alternative Flows

Description: Insert extended information to the contact.

From Step: 3M

To Step: 4M

Step Id	User Action	System State	System Response
1A	Go to context menu and select Extended Information.		The extended information form is displayed. [TRS_111166_102]
2A	Fill some of the extended information fields.		Some of the extended information form is filled.
3A	Press OK softkey.		The phone goes back to New Contact form. It is filled with the extended information.
Description: Cancel the new contact creation.			
From Step: 2M, 3M			
To Step: END			
Step Id	User Action	System State	System Response
1B	Press Cancel softkey.		The phone goes back to My Phonebook application menu.
Description: Cancel the insertion of extended information			
From Step: 3A			
To Step: END			
Step Id	User Action	System State	System Response
1C	Press Cancel softkey.		The phone goes back to My Phonebook application menu.
<b>Exception Flows</b>			
Description: There is no enough memory.			
From Step: 3M, 3A			
To Step: END			
Step Id	User Action	System State	System Response
1D	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS 111166 103]

2D	Select OK softkey.	The phone goes back to My Phonebook application menu.
----	--------------------	---

## **UC 02 – Searching a Contact**

### **Description**

This use case describes the searching of a previously created contact.  
This is the use case main setup.

### **Main Flow**

Description: Searching for a contact.  
From Step: START  
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the Search Contact option.		The Search Contact form is displayed.
3M	Type a string of a previously inserted contact.	There is at least one contact in the My Phonebook application.	The Search Contact form is filled.
4M	Select Search softkey.	There is enough phone memory to insert a new contact.	The list of matched contacts is displayed. [TRS_11111_104]
5M	Select Back softkey.		The phone goes back to My Phonebook application menu.

### **Alternative Flows**

Description: Open contact details.  
From Step: 4M

To Step: 5M

Step Id	User Action	System State	System Response
1A	Select a searched contact.		A searched contact is selected. [TRS_111166_105]
2A	Go to context menu and select Detail Contact.		A form containing all information related to the selected contact is displayed.
3A	Select back softkey.		The phone goes back to the list of matched contacts.

Description: There is no contact in the My Phonebook application.

From Step: 2M

To Step: END

Step Id	User Action	System State	System Response
1B	Type a string to search for any contact.	There is no contact in the My Phonebook application.	The Search Contact form is filled.
2B	Select Search softkey.		A dialog is displayed informing that no contact was found. [TRS_11111_106]
3B	Select Back softkey.		The phone goes back to the Search Contact form.

Table 9 – Original input TaRGeT - XML

```
<?xml version="1.0" encoding="UTF-8"?>
<phone xmlns="user-view.target.v20071129">
  <feature>
    <featureId>1111</featureId>
    <name>My Phonebook</name>
    <useCase>
      <id>UC_01</id>
      <name>Creating a New Contact</name>
      <description>This use case describes the creation of a new contact in
the contact list.</description>
```

```

<setup>This is the use case main setup.</setup>
<flow>
  <description>Create a new contact.</description>
  <fromSteps>START</fromSteps>
  <toSteps>END</toSteps>
  <step>
    <stepId>1M</stepId>
    <action>Start My Phonebook application.</action>
    <condition>My Phonebook application is installed in the
phone.</condition>
    <response>My Phonebook application menu is displayed.</response>
  </step>
  <step>
    <stepId>2M</stepId>
    <action>Select the New Contact option.</action>
    <condition />
    <response>The New Contact form is displayed.</response>
  </step>
  <step>
    <stepId>3M</stepId>
    <action>Type the contact name and the phone number.</action>
    <condition />
    <response>The new contact form is filled.</response>
  </step>
  <step>
    <stepId>4M</stepId>
    <action>Confirm the contact creation. [TRS_11111_101]</action>
    <condition>There is enough phone memory to insert a new
contact.</condition>
    <response>A new contact is created in My Phonebook
application.</response>
  </step>
</flow>
<flow>
  <description>Insert extended information to the
contact.</description>
  <fromSteps>3M</fromSteps>
  <toSteps>4M</toSteps>
  <step>
    <stepId>1A</stepId>
    <action>Go to context menu and select Extended
Information.</action>
    <condition />
    <response>The extended information form is displayed.
[TRS_111166_102]</response>
  </step>
  <step>
    <stepId>2A</stepId>
    <action>Fill some of the extended information fields.</action>
    <condition />

```

```

        <response>Some of the extended information form is
filled.</response>
    </step>
    <step>
        <stepId>3A</stepId>
        <action>Press OK softkey.</action>
        <condition />
        <response>The phone goes back to New Contact form. It is filled
with the extended information.</response>
    </step>
</flow>
<flow>
    <description>Cancel the new contact creation.</description>
    <fromSteps>2M, 3M</fromSteps>
    <toSteps>END</toSteps>
    <step>
        <stepId>1B</stepId>
        <action>Press Cancel softkey.</action>
        <condition />
        <response>The phone goes back to My Phonebook application
menu.</response>
    </step>
</flow>
<flow>
    <description>Cancel the insertion of extended
information</description>
    <fromSteps>3A</fromSteps>
    <toSteps>END</toSteps>
    <step>
        <stepId>1C</stepId>
        <action>Press Cancel softkey.</action>
        <condition />
        <response>The phone goes back to My Phonebook application
menu.</response>
    </step>
</flow>
<flow>
    <description>There is no enough memory.</description>
    <fromSteps>3M, 3A</fromSteps>
    <toSteps>END</toSteps>
    <step>
        <stepId>1D</stepId>
        <action>Confirm the contact creation.</action>
        <condition>There is no enough phone memory.</condition>
        <response>A dialog is displayed informing that there is no enough
memory. [TRS_111166_103]</response>
    </step>
    <step>
        <stepId>2D</stepId>
        <action>Select OK softkey.</action>

```



```

        <condition />
        <response>The phone goes back to My Phonebook application
menu.</response>
    </step>
</flow>
</useCase>
<useCase>
    <id>UC_02</id>
    <name>Searching a Contact</name>
    <description>This use case describes the searching of a previously
created contact.</description>
    <setup>This is the use case main setup.</setup>
    <flow>
        <description>Searching for a contact.</description>
        <fromSteps>START</fromSteps>
        <toSteps>END</toSteps>
        <step>
            <stepId>1M</stepId>
            <action>Start My Phonebook application.</action>
            <condition>My Phonebook application is installed in the
phone.</condition>
            <response>My Phonebook application menu is displayed.</response>
        </step>
        <step>
            <stepId>2M</stepId>
            <action>Select the Search Contact option.</action>
            <condition />
            <response>The Search Contact form is displayed.</response>
        </step>
        <step>
            <stepId>3M</stepId>
            <action>Type a string of a previously inserted contact.</action>
            <condition>There is at least one contact in the My Phonebook
application.</condition>
            <response>The Search Contact form is filled.</response>
        </step>
        <step>
            <stepId>4M</stepId>
            <action>Select Search softkey.</action>
            <condition>There is enough phone memory to insert a new
contact.</condition>
            <response>The list of matched contacts is displayed.
[TRS_11111_104]</response>
        </step>
        <step>
            <stepId>5M</stepId>
            <action>Select Back softkey.</action>
            <condition />
            <response>The phone goes back to My Phonebook application
menu.</response>

```

```

        </step>
    </flow>
    <flow>
        <description>Open contact details.</description>
        <fromSteps>4M</fromSteps>
        <toSteps>5M</toSteps>
        <step>
            <stepId>1A</stepId>
            <action>Select a searched contact.</action>
            <condition />
            <response>A searched contact is selected.
[TRS_111166_105]</response>
        </step>
        <step>
            <stepId>2A</stepId>
            <action>Go to context menu and select Detail Contact.</action>
            <condition />
            <response>A form containing all information related to the
selected contact is displayed.</response>
        </step>
        <step>
            <stepId>3A</stepId>
            <action>Select back softkey.</action>
            <condition />
            <response>The phone goes back to the list of matched
contacts.</response>
        </step>
    </flow>
    <flow>
        <description>There is no contact in the My Phonebook
application.</description>
        <fromSteps>2M</fromSteps>
        <toSteps>END</toSteps>
        <step>
            <stepId>1B</stepId>
            <action>Type a string to search for any contact.</action>
            <condition>There is no contact in the My Phonebook
application.</condition>
            <response>The Search Contact form is filled.</response>
        </step>
        <step>
            <stepId>2B</stepId>
            <action>Select Search softkey.</action>
            <condition />
            <response>A dialog is displayed informing that no contact was
found. [TRS_11111_106]</response>
        </step>
        <step>
            <stepId>3B</stepId>
            <action>Select Back softkey.</action>

```

```
                <condition />
                <response>The phone goes back to the Search Contact
form.</response>
            </step>
        </flow>
    </useCase>
</feature>
</phone>
```

Table 10 – Original output TaRGeT - HTML

## Test Cases

**Test Case ID: 1111\_MM\_Func\_001**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.  
**Objective:** None.

**Use Case References:** 1111#UC\_01  
**Requirements:** TRS\_11111\_101  
**Setups:** This is the use case main setup.

**Initial Conditions:** 1) My Phonebook application is installed in the phone. 2) There is enough phone memory to insert a new contact.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the New Contact option.	The New Contact form is displayed.
3) Type the contact name and the phone number.	The new contact form is filled.
4) Confirm the contact creation.	A new contact is created in My Phonebook application.

**Final Conditions:** None.  
**Cleanup:** None.  
**Notes:** Under Development

**Test Case ID: 1111\_MM\_Func\_002**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.  
**Objective:** None.

**Use Case References:** 1111#UC\_01  
**Requirements:** TRS\_111166\_102, TRS\_11111\_101  
**Setups:** This is the use case main setup.

**Initial Conditions:** 1) My Phonebook application is installed in the phone. 2) There is enough phone memory to insert a new contact.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the New Contact option.	The New Contact form is displayed.
3) Type the contact name and the phone number.	The new contact form is filled.
4) Go to context menu and select Extended Information.	The extended information form is displayed.
5) Fill some of the extended information fields.	Some of the extended information form is filled.
6) Press OK softkey.	The phone goes back to New Contact form. It is filled with the extended information.
7) Confirm the contact creation.	A new contact is created in My Phonebook application.

**Final Conditions:** None.  
**Cleanup:** None.  
**Notes:** Under Development

**Test Case ID: 1111\_MM\_Func\_003**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.  
**Objective:** None.

**Use Case References:** 1111#UC\_01  
**Requirements:** TRS\_111166\_102  
**Setups:** This is the use case main setup.

**Initial Conditions:** 1) My Phonebook application is installed in the phone.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the New Contact option.	The New Contact form is displayed.
3) Type the contact name and the phone number.	The new contact form is filled.
4) Go to context menu and select Extended Information.	The extended information form is displayed.
5) Fill some of the extended information fields.	Some of the extended information form is filled.
6) Press OK softkey.	The phone goes back to New Contact form. It is filled with the extended information.
7) Press Cancel softkey.	The phone goes back to My Phonebook application menu.

**Final Conditions:** None.  
**Cleanup:** None.  
**Notes:** Under Development

**Test Case ID: 1111\_MM\_Func\_010****Regression Level:** na**Execution Type:** Man**Description:** None.**Objective:** None.**Use Case References:** 1111#UC\_02**Requirements:** TRS\_11111\_106**Setups:** This is the use case main setup.**Initial Conditions:** 1) My Phonebook application is installed in the phone. 2) There is no contact in the My Phonebook application.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the Search Contact option.	The Search Contact form is displayed.
3) Type a string to search for any contact	The Search Contact form is filled.
4) Select Search softkey.	A dialog is displayed informing that no contact was found.
5) Select Back softkey.	The phone goes back to the Search Contact form.

**Final Conditions:** None.**Cleanup:** None.**Notes:** Under Development**Requirements Traceability Matrix - Requirements**

Requirement	Use Case
REQ - TRS_11111_101	UC (UC_01)
REQ - TRS_11111_104	UC (UC_02)
REQ - TRS_11111_106	UC (UC_02)
REQ - TRS_111166_102	UC (UC_01)
REQ - TRS_111166_103	UC (UC_01)
REQ - TRS_111166_105	UC (UC_02)

**Requirements Traceability Matrix - System Test**

Requirement	System Test Case
REQ - TRS_11111_101	TES (1111_MM_Func_001, 1111_MM_Func_002)
REQ - TRS_11111_104	TES (1111_MM_Func_008, 1111_MM_Func_009)
REQ - TRS_11111_106	TES (1111_MM_Func_010)
REQ - TRS_111166_102	TES (1111_MM_Func_002, 1111_MM_Func_003, 1111_MM_Func_004)
REQ - TRS_111166_103	TES (1111_MM_Func_004, 1111_MM_Func_006)
REQ - TRS_111166_105	TES (1111_MM_Func_009)

**Requirements Traceability Matrix - Use Case**

Use Case	System Test Case
UC - 1111#UC_01	TES (1111_MM_Func_001, 1111_MM_Func_002, 1111_MM_Func_003, 1111_MM_Func_004, 1111_MM_Func_005, 1111_MM_Func_006, 1111_MM_Func_007)
UC - 1111#UC_02	TES (1111_MM_Func_008, 1111_MM_Func_009, 1111_MM_Func_010)

**Test Case ID: 1111\_MM\_Func\_007**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.  
**Objective:** None.

**Use Case References:** 1111#UC\_01  
**Requirements:** None.  
**Setups:** This is the use case main setup.

**Initial Conditions:** 1) My Phonebook application is installed in the phone.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the New Contact option.	The New Contact form is displayed.
3) Press Cancel softkey.	The phone goes back to My Phonebook application menu.

**Final Conditions:** None.  
**Cleanup:** None.  
**Notes:** Under Development

**Test Case ID: 1111\_MM\_Func\_008**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.  
**Objective:** None.

**Use Case References:** 1111#UC\_02  
**Requirements:** TRS\_11111\_104  
**Setups:** This is the use case main setup.

**Initial Conditions:** 1) My Phonebook application is installed in the phone. 2) There is at least one contact in the My Phonebook application. 3) There is enough phone memory to insert a new contact.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the Search Contact option.	The Search Contact form is displayed.
3) Type a string of a previously inserted contact.	The Search Contact form is filled.
4) Select Search softkey.	The list of matched contacts is displayed.
5) Select Back softkey.	The phone goes back to My Phonebook application menu.

**Final Conditions:** None.  
**Cleanup:** None.  
**Notes:** Under Development

**Test Case ID: 1111\_MM\_Func\_009**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.  
**Objective:** None.

**Use Case References:** 1111#UC\_02  
**Requirements:** TRS\_11111\_104, TRS\_111166\_105  
**Setups:** This is the use case main setup.

**Initial Conditions:** 1) My Phonebook application is installed in the phone. 2) There is at least one contact in the My Phonebook application. 3) There is enough phone memory to insert a new contact.

Steps	Expected Results
1) Start My Phonebook application.	My Phonebook application menu is displayed.
2) Select the Search Contact option.	The Search Contact form is displayed.
3) Type a string of a previously inserted contact.	The Search Contact form is filled.
4) Select Search softkey.	The list of matched contacts is displayed.
5) Select a searched contact.	A searched contact is selected.
6) Go to context menu and select Detail Contact.	A form containing all information related to the selected contact is displayed.
7) Select back softkey.	The phone goes back to the list of matched contacts.
8) Select Back softkey.	The phone goes back to My Phonebook application menu.

**Final Conditions:** None.  
**Cleanup:** None.  
**Notes:** Under Development

**Test Case ID: 1111\_MM\_Func\_004**

**Regression Level:** na  
**Execution Type:** Man  
**Description:** None.

**APPENDIX B – EBFN NOTATION**

<b>Symbol</b>	<b>Symbol name</b>	<b>Purpose</b>
→	arrow	Term definition
	pipe	Or operator
+	cross operator	One or more occurrences
*	star operator	Zero or more occurrences
?	interrogation operator	Zero or one occurrences
;	semicolon	End of production
( )	left and right parentheses	It delimits a list of options

## APPENDIX C – PARTICIPANT CONSENT FORM

### Study title

Comparative evaluation between manual TCs generation x automatic generation of TCs using the ATG, evaluation of TCs automation, and evaluation of the usability of the ATG tool.

### General instructions

It is important that you carefully read the information on this form. This consent form provides you with all study information, such as purpose, procedure, data collection, privacy, costs, risks and additional information. Once you have understood the study, you will be asked to sign and date this form. If you need further clarification on any of the items mentioned here, or need information that has not been included, please ask the experimenters. Before being informed about the study, it is important that you become aware of the following:

1. Their participation is due to the fact that this study / experiment is a partial requirement of the evaluation of the master's thesis of Thaís Melise Lopes Pina;
2. You may request to leave the study at any time for any reason, and also that all data provided by you must be discarded.

You must clearly understand the nature of your participation and give your written consent. Your signature will indicate that you have understood all of the information regarding your participation and that you agree to participate.

### Study Purpose

Comparative evaluation between manual TCs generation x automatic generation of TCs using the ATG, an analysis of the possibility to automate TCs with Zygon tool, and an analysis of the usability of the ATG tool.

### Researches

Thaís Melise Lopes Pina is a master's student of the computer science center (CIn) of the Federal University of Pernambuco (UFPE), and this study is part of his research for the conclusion of the master's degree. Its advisor and co-advisor are respectively the teachers Augusto Cezar Alves Sampaio and Flávia de Almeida Barros.

### Procedures

Participants will be subject to a training of the tool the Auto Test Generator and soon after will answer a questionnaire on the usability of the tool.

### Data collection

The data will be collected through specific surveys and from JIRA (requirements and test cases).

### Confidential Records Character

The information obtained from participating in this study will be kept strictly confidential, since any material will be referenced only by an identifier. All results presented in the MSc thesis or in



scientific publications will be anonymous. However, to safeguard the researches who are conducting this study, all participants must provide their name and sign this consent form.

### **Risks and/or discomforts**

There is no possibility of risks or discomforts associated with the collaboration of any study participant.

### **Costs**

No participant will be charged or payed to participate in this study with their collaboration in the study.

### **Declaration of consent**

I declare that I have had sufficient time to read and understand the information contained in this consent form before signing it. The objectives and procedure have been explained, as well as what will be required of me as a participant. I also received answers to all my questions. I understand that I am free to request that my data not be used for analysis, without the application of any penalty. I also confirm that I have received a copy of this consent form. I give my consent to participate in this study.

---

Participant

---

Date

I attest that I have carefully explained the nature and purpose of this study. I believe that the participant received all the necessary information, which was described in a suitable and understandable language.

---

Thaís Melise Lopes Pina

---

Date

## APPENDIX D – ATG SURVEY

*The required fields are flagged with an asterisk (\*)*

**Name:\***

**Age:\***

**What your function at Motorola Mobility?\***

**How long time you work with test automation?\***

☐ 0 - 1 year      ☐ 1 - 2 years      ☐ 2 - 3 years      ☐ more than 3 years

**Do you use any tool to automate test?\***

☐ Yes      ☐ No

**If yes, what?** \_\_\_\_\_

*Next, you are answer your agreement level with the sentences*

**ATG tool has a friendly user interface.\***

Strongly disagree ( )    Disagree ( )    Undecided ( )    Agree ( )    Strongly Agree ( )

**ATG interface helps the user to write requirements.\***

Strongly disagree ( )    Disagree ( )    Undecided ( )    Agree ( )    Strongly Agree ( )

**ATG interface helps the user to reduce the automation effort.\***

Strongly disagree ( )    Disagree ( )    Undecided ( )    Agree ( )    Strongly Agree ( )

**What do you think needs to change/improve?** \_\_\_\_\_

---



---



---



---



---



---



---