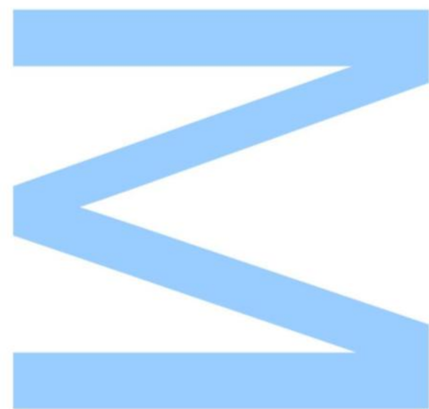
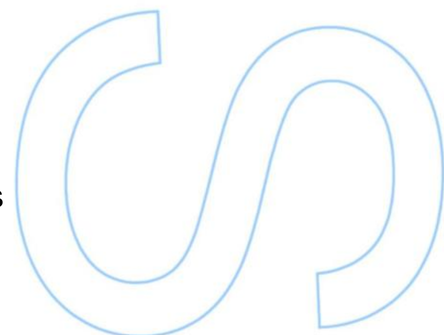
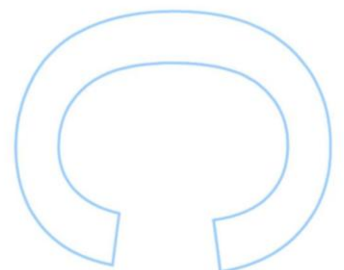
Artificial Intelligence Applied to Software Testing

João Salvado
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2019

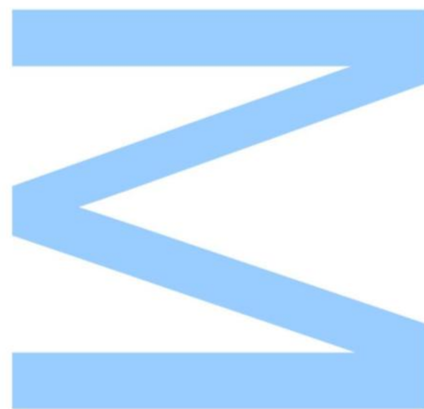
Orientador
Inês Dutra, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

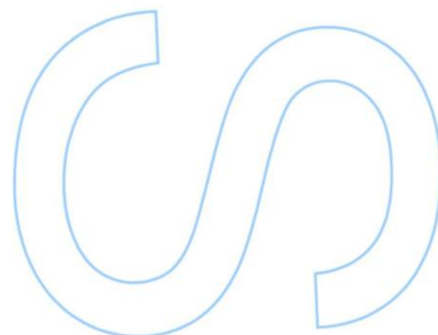
Coorientador
Vitor Conceição, Principal Engineer, CRITICAL Software

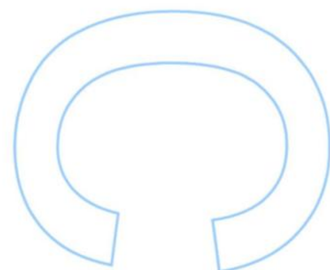
Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________







Abstract

The world we live in is undergoing a great and fast technological evolution, which leads to larger and more complex systems. The costs and resources associated with these systems are therefore increasing, which leads to the search for solutions that can mitigate this problem. Artificial Intelligence has, in recent years, taken a surge as a potential facilitator for this issue. That is why, taking advantage of this emphasis given to Artificial Intelligence, and aiming to reduce the resources associated with the development of safety-critical systems, this dissertation focuses on the automation of the Software Testing process using Artificial Intelligence. The main goal is to be able to generate C++ test code from test specifications using Natural Language Processing. We have proven in this dissertation that this is possible, as we present a parser of test specifications and also a new set of grammar rules to be used by software engineers when writing test specifications. Both of these contributions allowed for the generation of test code for up to three-quarters of the total test specifications. This dissertation exposes the whole process of solving the problem at hand, presenting the difficulties encountered, the decisions taken, and the results obtained.

Resumo

O mundo em que vivemos experiencia uma grande e acelerada evolução tecnológica, que se traduz em sistemas cada vez maiores e mais complexos. Os custos e recursos associados a estes sistemas são, portanto, cada vez mais acentuados, o que leva à busca por soluções que possam mitigar este problema. A Inteligência Artificial tem, nos últimos anos, sido uma das grandes apostas neste departamento. É por isso que, aproveitando este destaque dado à Inteligência Artificial, e tendo como objetivo reduzir os recursos associados ao desenvolvimento de sistemas críticos, se pretende nesta dissertação automatizar parte do processo de teste de software recorrendo à Inteligência Artificial. O grande objetivo é conseguir gerar código de teste a partir de especificações de teste utilizando processamento de linguagem natural. Conseguimos provar nesta dissertação que este objetivo é alcançável, mesmo em sistemas complexos. Propomos um *parser* de especificações de teste que, juntamente com um conjunto de regras de escrita definidas por nós, a serem usadas por engenheiros de software, permitem gerar código de teste para até três-quartos do total das especificações de teste. Esta dissertação aborda todo o processo feito para resolver o problema em questão, as dificuldades encontradas, as decisões tomadas, e os resultados obtidos.

Acknowledgements

First of all, I would like to thank my University mentor, professora Inês Dutra, and my corporation mentor and colleague Vitor Conceição for their invaluable advice, input and empathy towards me. Their help and guidance were essential for this dissertation's quality and organisation and I could not have done it without them.

Second, I would like to thank everyone of my family, particularly my parents, for their unwavering support which always kept me going.

And finally, I would like to thank all my friends and colleagues for all the support during this year. A special thanks to Paulo Gomes from Critical Software whose expertise in Artificial Intelligence was essential in pointing the right path. Also, an honourable mention to Patrick Grümer for helping to assemble and review the dissertation.

João Salvado, Porto, September 2019

Contents

Abstract i

Resumo iii

Acknowledgements v

Contents viii

List of Tables ix

List of Figures xi

Listings xiii

Acronyms xv

1 Introduction 1

1.1 Goals 2

1.2 Organisation 2

2 State of the Art 3

2.1 Artificial Intelligence in Software Testing 3

2.2 Automatic Programming 4

2.3 Natural Language Processors 4

2.4 Models 4

2.4.1 Word2vec 4

2.4.2 GloVe 5
2.5 Tools 5
2.5.1 Natural Language ToolKit (NLTK) 5
2.5.2 SpaCy 5
2.5.3 NLTK vs. SpaCy 5
3 Automatic Test Specification Generation 7
3.1 Data Explanation 7
3.1.1 Test Specifications 8
3.2 Manipulation and Analysis of the test specifications 8
3.3 Test Code Generation 10
3.4 Test Specification Writing Rules 11
4 Results 15
5 Conclusions 17
5.1 Future Work 17
A Grammars 19
B Code Generation Script 25
Bibliography 27

List of Tables

3.1 Most common Input actions, out of 21673 entries (66%) 9
3.2 Most common Expected Output actions, out of 23951 entries (54%) 9

List of Figures

1.1 Verification and Validation (VV) stream. This dissertation focuses on automating the section marked in red. 2

3.1 Organisation of the test specifications 8

5.1 A possible future application on the VV stream, marked in red. 18

5.2 A possible future application on the Development stream, marked in red. 18

B.1 Excerpt of the test code generation script 26

Listings

3.1 Example of Test Code for a simple "Set" action of the BPCU project 11

Acronyms

AI	Artificial Intelligence
CFG	Context-Free Grammar
NLP	Natural Language Processing
NLTK	Natural Language ToolKit
VV	Verification and Validation

Chapter 1

Introduction

We live in a world currently going through an accelerated technological evolution, which demands increasingly larger and more complex information systems. In order to be sustained, these information systems require a vast amount of human and material resources, as well as an extended development cycle time. However, these measures are not always enough to keep up with the above-mentioned technological growth, hence inspiring the search for new approaches to this problem.

One of these approaches focuses on the mitigation of the problem by automating certain parts of the development cycle. This method not only reduces the development cycle time, but also lessens the human error factor from the development cycle, which tends to escalate with the complexity and size of the systems. Since this approach reduces the workload on humans, prone to fatigue and distractions, the overall quality of the product will increase, proving it to be a valuable asset for tackling the problem.

In what concerns Safety Critical Systems, the Verification and Validation (VV) activities represent a significant part of the development process and therefore any automation will be greatly beneficial in terms of cost and quality [2]. One of the means of applying this approach to this particular type of software consists in automating Continuous Integration-runnable tests, allowing for quicker and automated regression testing, and simplifying the development process in general.

That being said, automating tests can be the solution to follow in order to improve the critical systems development scene. And since in the last few years Artificial Intelligence (AI) has taken a leap in terms of evolution and is being applied to a great number of products and services, the combination of these two can be a major step to improving the critical systems development process with good chances of success.

1.1 Goals

The main goal of this thesis is therefore to combine test automation and AI, rendering the VV process faster, finer, and better suited to keep up with the evolution and complexity of today's systems.

The ultimate objective is to be able to generate C++ test script code from existing test specifications written by software engineers in natural language. This process is a standard part of the VV procedure in safety critical software development, as seen in Figure 1.1.

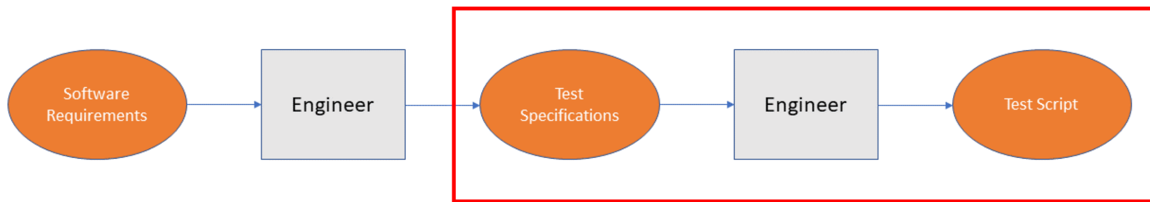


Figure 1.1: VV stream. This dissertation focuses on automating the section marked in red.

The test specifications provide a step-by-step description for thoroughly testing the software in hand and should be used primarily, along with whatever other project-related information is available (such as requirements or data dictionaries), to produce functioning test script code with a certain degree of reliability. This can be achieved using Natural Language Processing (NLP), a sub-field of AI, to handle the natural language written test specifications.

1.2 Organisation

This document is organised as follows: Chapter 2 is dedicated to the state of the art, where details are given about the research made, and the existing models and tools that were found related to the theme. In Chapter 3, the performed work and contributions are described in detail, starting o with an initial section which introduces the provided data, with particular emphasis on the test specifications; the manipulation and analysis of those test specifications are then described, followed by an explanation of the process of generation of C++ Test Code using Natural Language ToolKit (NLTK) to parse the test specifications. The obstacles found, and the solutions we came up with are also presented in this chapter. Subsequently, the obtained results are presented in a chapter of their own and, finally, the last chapter is dedicated to the conclusions and future work.

Chapter 2

State of the Art

This chapter presents the state of the art for the themes at hand for this dissertation, that is, Natural Language Processing (NLP), software testing, and code generation. The models and tools presented below are the results of an initial search resorting to the Google search engine (introducing the topics that serve as basis for this dissertation), and of a latter article search using Google Scholar (which works as a search engine for academic literature) and arXiv (a repository of scientific articles from various fields), both of which were used for bibliographic review.

The search in these databases was done using keywords related to this dissertation's subjects, such as: "natural language processing", "word embedding", "code generation", "automatic programming" and the names of the models and tools found in the initial search ("word2vec", "nltk", etc.).

The articles found provided a strong foundation for the subject under study, and helped to expand the knowledge and can assist in future research and challenges that may arise.

2.1 Artificial Intelligence in Software Testing

With the rise of Artificial Intelligence (AI) over the last decades, its application began to be studied in various areas. Software Engineering was no exception [4, 15], and Software Testing in particular, as one of its core activities, has received a major surge of study cases to try and optimise the resource usage in this area [2]. Since all of the data used in modern Software Testing can be parsed by a machine, AI has proved to be a valuable ally in a lot of di erent departments of this activity [3, 7, 9].

2.2 Automatic Programming

The ultimate objective of this dissertation is to generate C++ Test code automatically, so a few notions of automatic programming are essential to better understand the problem we face. The concept of automatic programming has existed for decades, and the general consensus defines it as the generation of a computer program by a machine, with the main goal of saving time and resources for humans [1, 15]. However, since that machine requires human input, automatic programming has also been described as simply an euphemism to programming at a higher level, letting successive machine iterations generate increasingly lower-level code, down to machine code [13]. And in a way, that is exactly what is proposed in this dissertation: to parse human-written Test Specifications, which will act as an extremely high-level programming language, to generate lower-level C++ Test Code.

2.3 Natural Language Processors

Although it has only recently been heard of them, Natural Language Processors have been around for a while, appearing in the mid-1950s in the form of rudimentary translation systems. However, it was with the advent of machine learning techniques in the 1980s that this area underwent a revolution that would change the way these processors operate. Originally, these were driven by complex sets of rules formulated by humans (by defining grammars, for example), but over the past few years a preference for statistical-based processing has emerged, supported by developments in the area of Machine Learning [8]. Thanks to these developments, new methods of NLP have emerged, which, although not used in this dissertation, can be valuable for future enterprises in the area of critical systems.

The following sections present the found models and tools relevant for this dissertation.

2.4 Models

This section presents two word embedding models that were considered to be used in this dissertation as a means to relate the Test Specifications and the Test Code in a word vector representation. However, as the work progressed, and given the nature of the test specifications, they ended up not being used in this dissertation.

2.4.1 Word2vec

Word2vec was created in 2013 by a Google team led by Tomas Mikolov, which is one of the most popular vector word representation models (known as word embedding) [12]. This predictive model, based on neural networks, allows you to associate words by analysing sets of

texts, organising them into a vector space where each word corresponds to a vector in space. This representation has semantic properties that make Word2vec especially powerful for text comparison, sentiment analysis or machine translation, for example [11].

2.4.2 GloVe

GloVe is another word embedding model developed at Stanford University, but unlike Word2vec, it relies on counting individual words rather than trying to predict them. It is best-suited for finding synonyms, and relating words such as companies and products, or zip-codes and cities [14].

2.5 Tools

This section presents the two candidate NLP open-source tools to be used during this dissertation, and the reasons for choosing one over the other.

2.5.1 Natural Language ToolKit (NLTK)

NLTK is a set of natural language processing libraries and programs for English language. It is especially aimed at teaching and research and provides an extensive manual written by its creators that introduces the user to NLTK and NLP [10].

2.5.2 SpaCy

SpaCy is, like NLTK, a natural language processing software, but which is more directed towards commercial software production. It is written in Python and Cython, and their developers claim to use the best and latest NLP algorithms, and vow to update them as the state of the art progresses[5].

2.5.3 NLTK vs. SpaCy

While NLTK is basically a string-processing library, SpaCy uses an object-oriented approach. Since, personally, we would rather work with strings than with objects, NLTK had the upper-hand on this department. The advantages of SpaCy over NLTK are better performance, and a focus by their developers on having the latest and greatest problem solving algorithms. NLTK, on the other hand, features a much wider range of functions and algorithms, ideal for experimentation. This was, therefore, the biggest argument in favour of using NLTK over SpaCy in this dissertation.

For the reasons mentioned above, NLTK was regarded from the start as a strong candidate

to be used for experimentation, and ended up being the one used to solve the problem presented in this dissertation.

Chapter 3

Automatic Test Specification Generation

This chapter describes the contribution for tackling the problem. After a brief introduction on the nature of the provided data of study, the performed work is described by chronological order of events, accompanied by examples whenever necessary.

3.1 Data Explanation

The primary goal of this thesis consists in generating C++ code from existing test specifications. Therefore, it is necessary to have a deep understanding of both the specifications and the respective code that implements them, in order to efficiently manipulate them.

The data provided consists of two sets of test information from two different Critical Software projects, both containing four different types of data:

1. **Requirements under Test** - The features of the software that the testing process is supposed to evaluate
2. **Test Specifications** - The procedure to be followed in order to evaluate the correctness of the implemented features/requirements
3. **Test Scripts** - The implemented script responsible for the automated execution of the tests
4. **Traceability** - The correspondence between the Requirements under Test and the Test Specifications, which indicates what requirements each test specification is covering

The test specifications, undeniably the most important data chunk for this thesis, are arranged in a similar manner in both projects, which was a big facilitating factor for manipulating the data. The test specifications' organisation is detailed below.

3.1.1 Test Specifications

A Test Specification usually consists on a set of single sentences written in natural language by engineers, which describes a certain step to be taken in order to test the software in question. There are no guidelines that concern the writing of the test specifications itself, other than clarity in the actions described. This means that the engineers are free to write as they find best, and so each set of test specifications will have its own writing style.

Each test specification sentence always belongs to one of the following categories:

- **Inputs**, which describe actions or causes for which certain e ects should occur
- **Expected Outputs**, which describe the expected results or e ects for the tested conditions in accordance to the features/requirements

There is always at least one Input entry before any Expected Outputs. The relation between the number of Inputs and Expected Outputs varies, meaning that there can be any number of Inputs followed by any number of Expected Outputs, successively.

Each of the provided project sets consists of test specifications organised in Test Cases, which are divided in a number of Test Steps, each one containing several single-sentence test specifications. Each Test Case corresponds to an individual Excel file, in which the Test Specifications are arranged by Test Steps. Each Test Action is described in a single cell, and its category is given by its column (Inputs or Expected Outputs), as seen in Figure 3.1.

	Inputs	Expected Outputs
010 A	Boot in diagnostic mode (BPCU_OFP_INT_TEST_5000-0009)	
B	Set AIRCRAFT_SERIAL_NUMBER to 0xBEEF in NVM.	Check that AIRCRAFT_SERIAL_NUMBER is set to 0xBEEF in NVM.
C	Set-up AIRCRAFT_SERIAL_NUMBER (CAN message) so that it's value is the same as the one previously stored on NVM.	
D	Set Discrete input WOW to TRUE.	
E	Set WOW to TRUE in RS485	
F	Execute the necessary procedures to power-on BPCU (BPCU_OFP_INT_TEST_5000-0005)	CAN BIT_STAUS output data must be initialized as FAULT/ FAIL (if flag) or 0 (if counter):
		CAN output data CAN_A_FAULT is set to FALSE.
		CAN output data CAN_A_LOOPBK_FAULT is set to FALSE.
		CAN output data CAN_A_NB_RST is set to FALSE.

Figure 3.1: Organisation of the test specifications

3.2 Manipulation and Analysis of the test specifications

Taking advantage of the fact that the test specifications have identical organisational structure in the two projects (both inside the Excel files, and at the file system level), the first step to this dissertation consisted in creating a script to extract the specifications from the Excel files, and converting them to a format that is both human-readable and easy to parse from a Natural Language Processing (NLP) point-of-view.

At first, that format was decided to be XML, as it has the properties for being both easily parseable by a machine, and being su ciently human-readable. However, as the time passed by, and with the natural work flow, the format that ended up prevailing was plain text, sorting one

sentence per line, and dividing the test specifications in two files: one for Inputs, and another one for Expected Outputs. The reasons for this change are that Natural Language ToolKit (NLTK) parses plain text by default, and that XML ends up being a lot more verbose than plain text, making it more di cult to read by humans. It was also pretty clear that the actions should be separated in their two categories (Inputs and Expected Outputs) since each category would have their own set of specifics consistently throughout the specifications (this will be detailed further below).

A grand total of 45624 sentences from both projects were extracted using a created Python script, from which 21673 were Inputs, and 23951 were Expected Outputs.

Once the data was duly organised, the next stage of the problem ensued: in order to generate code using the test specifications, a careful analysis of both the test specifications and the respective Test Scripts was performed. The ultimate objective was to find common points, i.e., actions described in the specifications that could originate the same snippets of code.

A quick study demonstrated that, for the most part, similar actions resulted in the same code excerpts. To be more accurate, the same test specification almost always used a certain C++ function consistently to perform that action through code. These actions are manifested through certain verbs. For example "Set", "Wait" and "Send" are common actions for Input specifications, while "Verify" and "Check" are recurrent Expected Output actions. The most common verbs occurring in each category are identified in Tables 3.1 and 3.2.

Actions	Occurrences
Set	6045
Wait	2928
Send	2078
Resume	645
Execute	589
Upload	491
Clear	486
Perform	433
Initialise	346
Initiate	337

Table 3.1: Most common Input actions, out of 21673 entries (66%)

Actions	Occurrences
Verify	10086
Check	2928

Table 3.2: Most common Expected Output actions, out of 23951 entries (54%)

After finding the most common actions, we opted to create Context-Free Grammars (CFGs) for each of the most frequent ones, since the test specifications followed recurrent patterns which

could easily be translated into CFGs. A CFG is a set of production rules that describe all possible strings in a given formal language [6]. It always has:

1. a finite set of symbols that form the strings of the language. In our case, the words that form each test specification sentence;
2. a finite set of variables (also called nonterminals), that represent a set of strings;
3. a start symbol, which is a variable that represents the whole language being defined;
4. a set of rules that represent the recursive definition of a language. Each rule is comprised of a variable, followed by a "   ", followed by a string of zero or more terminals or variables. In our defined CFGs, "ϵ" represents a null production, and "|" represents "or".

The created CFGs can be seen on Appendix A.

After presenting the CFGs to my mentors and to Paulo Gomes, the Head of Artificial Intelligence & Machine Learning at Critical Software, it was decided that direct code generation could be of valuable use. Two main aspects derived this decision: 1) NLTK does not allow creation of CFGs with unspecified terminals, and 2) the test specifications under analysis already possess a machine-readable nature. This meant that the code could be easily printed according to each case.

The next logical step would be to attempt to generate the code based on everything that was learned and decided up to this point.

3.3 Test Code Generation

The code generation was the ultimate goal of this dissertation, and, thus far, everything was on track to being able to do so successfully.

However, at this point a big question arised: the two projects had, for the most part, the same type of actions, both for Inputs and for Expected Outputs, but the Test Code for the same actions was radically di erent due to the nature of the projects and the dedicated automated test engines. This led to the decision of focusing on only one of the two sampling projects, henceforth referred to as BPCU (which stands for Bus Power Control Unit). This decision was taken mainly to ensure the quality of the results by being able to concentrate on a single project, while reducing the workload of having to deal with two di erent environments. An example of BPCU test specification and dedicated Test Script can be seen in Listing 3.1.

A script was then created and continuously developed using rules based on the conceived CFGs, and taking the characteristics of the project into account. This script simply uses NLTK's word tokenizer to split each test specification in sub strings (words), and then, based on the key action at hand and the other words' relative position, it tries to generate the appropriate code.

```
// Set STATUS_AUX_EAC_L discrete signal equal to OPEN;
SET_DIGITAL_INPUT(STATUS_AUX_EAC_L, OPEN);
```

Listing 3.1: Example of Test Code for a simple "Set" action of the BPCU project

An excerpt of the Python script used to generate Test Code by parsing the Test sentences can be seen in Appendix B.

The attempt at code generation used the CFG for "Wait <some>ms" as a starting point, because it was the most elementary of the CFGs. It immediately produced very successful results, as expected given its simplicity. Of all the occurrences of this form, 100% were correctly transformed into test code. This represented already approximately 8% of the total 23043 BPCU test specifications being translated to code successfully.

After this initial success, the same process implementation was made for each of the other common forms. This, however, posed a problem: to ensure that the code generation was correct, a manual check was required. And given the size of the BPCU data (23043 entries) that was out of the question.

The solution was to choose one single Test Case from the BPCU project to serve as a sample and facilitate the checking process. This sample would have to be su ciently reflective of the general data, and neither be too small that it could give misleading results, or too big that it would take too much e ort to manually check. Thus a Test Case, hereinafter referred to as BPCU_1003, was chosen that fit this criteria.

Implementing the four di erent syntactic forms in the test generation script and applying it to BPCU_1003 generated mediocre results this time around. Out of a total of 1411 Test actions, code was generated for 689 (approx. 49%), and merely 385 (27%) were completely successful. This meant that, when the script recognised one of the syntactic forms and attempted to generate code, it would only generate perfectly correct code for about a little over half (56%) of those recognised Test actions. It also meant that the overall generation success rate was sitting at approximately 27%, which desperately needed to be improved.

To increase the rates of generation and success, and since the tuning of the test generation script would be technically complicated and extensive, we opted for implementing a solution that was a possibility from the start of the dissertation.

3.4 Test Specification Writing Rules

From the start, one of the possibilities of tackling the problem was the definition of formal rules for the software engineers to follow when writing a Test Specification Standard. These rules' objective is to render the test specifications more parseable, thus improving generation and success rates. With an already good knowledge of the specifications' constitution, a set of rules,

presented below, was created to meet these goals.

Rule No. 1 *- Use the most common forms whenever possible.*

This rule is the most important one, since many of the test specifications can be written in one of the most common forms. It includes, for example, defining a value immediately in situations where it is required to choose a value contained on a certain interval, e.g., "Set x to a value between 0 and 10"   "Set x to 5 - a value between 0 and 10"; using the actual action verbs instead of alternate forms, e.g., "Change..."   "Set...".

Rule No. 2 *- Use variable names whenever possible.*

This rule eases the parsing of the test specifications by directly providing one of the arguments for the function of the Test Code. E.g., "Set the reference analogue value to 5 V"   "Set REF_ANALOGUE_VAL to 5 V".

Rule No. 3 *- Always separate units from values.*

Sometimes the unit of a certain value is glued to the value. This rule aims to impose the practice of splitting them, since they will otherwise be identified as one single token. Their separation in two di erent tokens also allows for the individual manipulation of each. E.g., "Set REF_ANALOGUE_VAL to 5V"   "Set REF_ANALOGUE_VAL to 5 V"

Rule No. 4 *- Split multiple actions into individual ones whenever possible.*

Certain test specifications contain multiple actions in a single cell, or enumerate the arguments for the action in question. This rule ensures that each action is turned into code, avoiding the parsing of only the first action. E.g.:

"Set the following:
x to TRUE,
y to FALSE..."

would be instead

"Set x to TRUE"
"Set y to FALSE".

This set of Test Specification Writing Rules ultimately impacted the quality of the generated code greatly. The correct code excerpts upon recognition of one of the four syntactic forms spiked to approximately 94%. However, the generation rate barely improved, from 689 (49%) to 746 (53%) generated code excerpts, which meant that the overall generation success rate (of the entirety of the BPCU_1003 test specifications) was sitting at around 50%, and that further action was needed to ensure satisfactory results.

With the implementation of the parsing of the four most common forms and the definition of the Test Specification Writing Rules, a very respectable success rate had been achieved, but not

so much for the generation rate, meaning we should focus on improving the latter. We chose to approach this challenge through the reinforcement of our knowledge of the test specifications, by performing an exhaustive study on their constitution and frequency.

This scrutiny consisted mostly in statistical analysis and searches throughout the test specifications for patterns, using primarily basic regular expressions.

During this process of deepening the analysis of the Test actions, new forms were found that would be successfully turned into correct Test Code if the Test Specification Writing Rules were followed. It was also detected that certain Test actions would always produce the exact same code excerpts, which could be easily incorporated into the code generation script by usage of a direct code parser.

With these findings, an already acceptable set of results was produced, which will be presented in the next chapter.

Chapter 4

Results

This chapter describes the obtained results concerning the work performed, specifically the code generation rate and the correctness of the generated code. The results concern solely to project BPCU.

The first attempt at Test Code generation implemented the "Wait <some>ms" Context-Free Grammar (CFG), the simplest of them all, and it produced 100% correct Test Code for all 1852 generations. This represents about 8% of the test specifications of the BPCU project.

The other CFGs were then implemented, and the results, this time concerning only the chosen sample BPCU_1003, were radically di erent. A mere 689 Test actions were identified from the 1411 entries of BPCU_1003, which corresponds to approximately 49%. Moreover, of those, just about 56% (385) generated correct Test Code, which, out of all 1411 sentences, accounted for only 27% successful Test Code generations.

The definition of Test Specification Writing Rules greatly improved the correctness of the generated code, bumping it from 56% to 94%. It also improved, albeit very slightly, the number of generations, from 49% to 53%. At this point, around half of all BPCU_1003 Test actions were being generated correctly. When the Test Code generation script was run in the entirety of the BPCU project, 14635 Test code snippets were generated (around 64% of the 23043 Test actions) which, assuming the rate of success obtained for sample BPCU_1003 of 94%, meant that approximately 60% of the test actions were being successfully turned into Test Code.

With the thorough study performed on the test specifications' constitution and frequency, it was noted that the overall rate of successfully generated code excerpts, i.e., the percentage of code snippets that fully and correctly implement test specifications, was approximately 76%, already taking into account the 94% success rate obtained in BPCU_1003 (note that while the generated code excerpts for sample BPCU_1003 were individually verified against the actual existent test code, the results for the whole BPCU project were not, due to the size of the project; hence the applying of the results obtained for the sample to the whole project).

These 76% represent an already respectable rate: to have three quarters of the data correctly

generated would already greatly benefit the testing process both time- and resource-wise.

The next chapter is the last one of this document, where we present the conclusions of our work and discuss future prospects in the area of this dissertation.

Chapter 5

Conclusions

The obtained results were crystal clear regarding the possibility of generating test code using test specifications as a starting point. It was proven that, given the machine-friendly way that the test specifications are naturally written, as well as its simple structure, it is possible to parse and manipulate those specifications to our content using simple Natural Language Processing (NLP) tools. It can also be concluded that the use of machine learning techniques, which was a possibility from the start, would be over-complicating the problem, and that these would not provide the degree of trust and reliability that safety-critical systems require.

The particular study case approached in this thesis birthed very satisfactory results, with more than three quarters of the test specifications being successfully transformed in functional code. The proposed objectives were, therefore, accomplished, which gives good indicators for future prospects.

5.1 Future Work

The work produced in this dissertation shows good results, opening the path for further investigation and optimisation of these topics with the aim of further improve the safety critical development cycle.

Therefore, a possible next step would be processing software test requirements for the automatic generation of test specifications, as seen in Figure 5.1 (which could then be parsed into test source code, as demonstrated in this dissertation - see Figure 1.1).

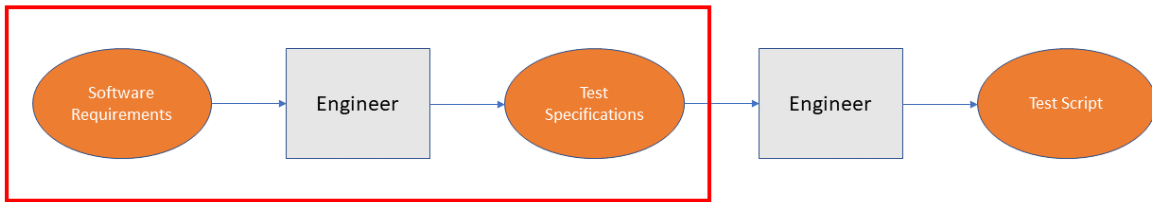


Figure 5.1: A possible future application on the VV stream, marked in red.

Another possible step would be the application of this same concept to other types of requirements, such as low level requirements (detailed design), for the automatic generation of software source code, as seen in Figure 5.2.

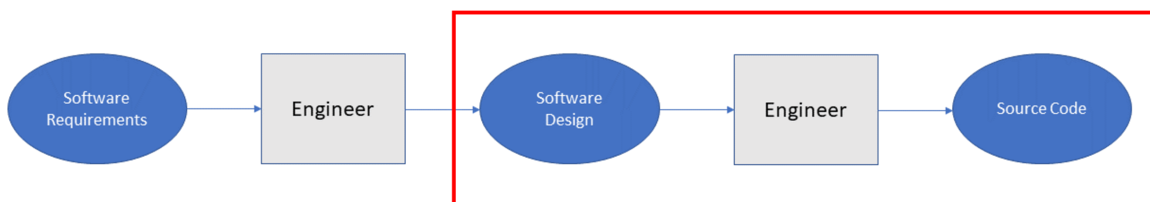


Figure 5.2: A possible future application on the Development stream, marked in red.

Analogously to the test specification writing rules presented in this dissertation, a standard for writing software requirements could be created in order to increase the success rate of such generation.

These two applications would signify the automation of a great part of the developing and testing process, which would save resources on the development aspect, while focusing and reinforcing the process of reviewing. Additionally, new useful features can be added with the help of other project-related data, for example, the script can act as a kind of compiler to detect possible software requirement issues, like variable names or units not aligned with the Data Dictionary. It could also go further and perform a syntactic analysis on the requirements, in order to help the engineers to write in a manner that improves the code generation rate.

In this dissertation the easily parseable requirements were taken advantage of by directly generating the code according to the words and their position in each test specification. Although it was concluded, for the reasons mentioned above, that machine learning is not a viable solution for safety-critical systems, it could be of valuable use in software where reliability is not of utmost importance. Machine learning classifiers could be used to try and teach a parser how to identify di erent types of patterns in non-safety-critical software requirements, and to generate the respective code accordingly.

Appendix A

Grammars

This appendix contains the Context-Free Grammars that were defined for the four most frequent Test actions found in the Test Specifications.

Simple “SET” input action grammar:

Simple most common form:

*“Set” <**some_variable**> “to” <**some_value**>*

Which translates to:

SSETG -> SET Var TO Val
SET -> “Set”
TO -> “to”
Var -> *(one or more tokens)*
Val -> *(first token immediately after “to”)*

Simple “WAIT” input action grammar:

Simple most common form:

“Wait” <some_milliseconds>

Which translates to:

SWTG -> WAIT Ms

WAIT -> “Wait”

Ms -> “<time>ms”, where <time> is an integer

Simple “SEND” input action grammar:

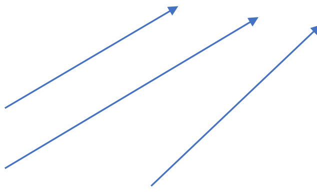
Simple most common form:

*“Send a message by” <**some_mean**> “with” <**some_variable**> “as” <**some_value>***

Which translates to:

SSDG -> SMB Mean WITH Var AS Val
SMB -> “Send message by”
WITH -> “with”
AS -> “as”
Mean -> *(one or more tokens)*
Var -> *(one or more tokens)*
Val -> *(first token immediately after “to”)*

These are probably uppercase



“Verify/Check” expected output action grammar:

Most common form:

*“Verify/Check in/via” <**some_place**> “that” <**some_variable**> “is set to” <**some_value**>*

Which translates to:

VCHG -> VC Var IST Val
VC -> VeCh Opt THAT |
VeCh -> “Verify” | “Check”
Opt -> *(one or more tokens of additional info)* |
THAT -> “that”
IST -> IS | ST
IS -> “is”
ST -> “set to” |
Var -> *(one or more tokens)*
Val -> *(first token immediately after “to”)*

Appendix B

Code Generation Script

This appendix contains an excerpt of the Python script used to generate the C++ Test Code based on the Test actions found. The script starts by tokenizing the Test Specification sentence, and then parses it and generates the Test Code based on the tokens' position, determined by the defined Context-Free Grammars (CFGs).

```python
    # tokenize entry
    tokens = word_tokenize(ws.cell(row = i, column = 6).value)
    tokens = [w for w in tokens if w != '.']

    # case 'Wait <some_time>ms'
    if tokens[0] == 'Wait':
        if tokens[1].endswith('ms'):
            outfile.write('\t\t// TCGEN: The code bellow was automatically generated\n')
            outfile.write('\t\tWAIT_MILLISECONDS(' + tokens[1][:-2] + ', FROM_SPEC);\n')

    # case 'Set <some_variable> to <some_value>'
    if tokens[0] == 'Set' and 'to' in tokens:
        var_name = tokens[1]
        j = 1
        while tokens[j] != 'to':
            if re.match(r'^[A-Z0-9_]*$', tokens[j]):
                var_name = tokens[j]
                break
            j += 1
        set_val = tokens[tokens.index('to')+1]
        outfile.write('\t\t// TCGEN: The code bellow was automatically generated\n')
        outfile.write('\t\tSET_SOMETHING(' + var_name + ', ' + set_val + ');\n')

    # case 'Send a message by <some_mean> with <some_variable> as <some_value>'
    if tokens[:4] == ['Send','a','message','by']:
        var_name = tokens[6]
        set_val = tokens[-1]
        if tokens[4] == 'RS485':
            outfile.write('\t\t// TCGEN: The code bellow was automatically generated\n')
            outfile.write('\t\tSET_LEFT_RS485_MSG_VALUE(' + var_name + ', ' + set_val + ');\n')
        if tokens[4] == 'CAN':
            outfile.write('\t\t// TCGEN: The code bellow was automatically generated\n')
            outfile.write('\t\tSET_RGCU_CAN_MSG_VALUE(' + var_name + ', ' + set_val + ');\n')

    # remaining cases
    else:
        outfile.write('\t\t// Generation not possible, write the test code bellow\n')

# Expected Outputs
# (...)

    # tokenize entry
    tokens = word_tokenize(ws.cell(row = i, column = 7).value)
    tokens = [w for w in tokens if w != '.']

    # case 'Check/verify...'
    if (tokens[0] == 'Check' or tokens[0] == 'Verify') and 'that' in tokens and 'is' in tokens:
        var_name = tokens[tokens.index('that')+1]
        set_val = tokens[-1]
        for k in range(tokens.index('that')+1,tokens.index('is')):
            if re.match(r'^[A-Z0-9_]*$', tokens[k]):
                var_name = tokens[k]
                break
```

Figure B.1: Excerpt of the test code generation script

Bibliography

[1] Robert Balzer. A 15 Year Perspective on Automatic Programming. Technical Report 11, Institute of Electrical and Electronics Engineers (IEEE), 1985.

[2] Jonathan Bowen and Victoria Stavridou. Safety-critical methods and systems, formal standards. *Software Engineering Journal*, 8(4):189–209, 1993. doi:10.1049/sej.1993.0025.

[3] Anurag Dwarakanath and Shubhashis Sengupta. Litmus: Generation of test cases from functional requirements in natural language. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7337 LNCS, pages 58–69, 2012. ISBN: 9783642311772. doi:10.1007/978-3-642-31178-9_6.

[4] Robert Feldt, Francisco G. de Oliveira Neto, and Richard Torkar. Ways of Applying Artificial Intelligence in Software Engineering. *RAISE '18 Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2018.

[5] Matthew Honnibal. SpaCy - Industrial-Strength Natural Language Processing in Python, 2015.

[6] John E. Hopcroft, Rajeev Motwani, and Je rey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition, 2001. ISBN: 0201441241. doi:10.1145/568438.568455.

[7] Hussam Hourani, Ahmad Hammad, and Mohammad Lafi. *The Impact of Artificial Intelligence on Software Testing*. 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), 2019. ISBN: 9781538679425.

[8] John Hutchins. The history of machine translation in a nutshell, 2005.

[9] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic Automated Software Testing. In *Implementation of Functional Languages: 14th International Workshop*, pages 84–100. Springer, 2003. doi:10.1007/3-540-44854-3_6.

[10] Edward Loper and Steven Bird. NLTK: The Natural Language Toolkit. In *Proceedings of the {ACL} Interactive Poster and Demonstration Sessions*, pages 214–217. Association for Computational Linguistics, 2004.

[11] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Je rey Dean. Distributed Representations of Words and Phrases and their Compositionality. *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 3111–3119, 2013.

[12] Anupiya Nugaliyadde, Kok Wai Wong, Ferdous Sohel, and Hong Xie. Enhancing Semantic Word Representations by Embedding Deeper Word Relationships. In *ACM International Conference Proceeding Series*, pages 155–160. Association for Computing Machinery, 2017. ISBN: 9781450348034. doi:10.1145/3029387.3029392.

[13] David Lorge Parnas. Software Aspects of Strategic Defense Systems. *Communications of the ACM*, pages 1326–1335, 1985. doi:10.1145/214956.214961.

[14] Je rey Pennington, Richard Socher, and Christopher D Manning. GloVe: Global Vectors for Word Representation. Technical report, Leland Stanford Junior University, 2014.

[15] Charles Rich and Richard C. Waters. The Programmer's Apprentice Project: A Research Overview. *Computer*, 21(11):10–25, 1988. doi:10.1109/2.86782.