Master Thesis in Computer Science
with Specialization in Software Engineering

From Requirements Specification to
Test Scripting:
Towards Automated Support

Robert Gustavsson
rgn09003@student.mdh.se
School of Innovation, Design and Engineering
Mälardalen University

Dimitrios Kostopoulos
dks12001@student.mdh.se
School of Innovation, Design and Engineering
Mälardalen University

Company Supervisor:
Manuel Palmieri
manuel.palmieri@se.bombardier.transportation.se
Bombardier Transportation AB

Academic Supervisor:
Antonio Cicchetti
antonio.cicchetti@mdh.se
Mälardalen University

Examiner:
Ivica Crnkovic
ivica.crnkovic@mdh.se
Mälardalen University
Abstract

Testing represents one of the major concerns in nowadays system development, since it provides both confidence in the product and enriches the quality of the software itself. In particular, system level testing allows to verify and validate the adherence of the developed product to the expectations specified in the requirements. However, specifying system level test cases and ensure that all requirements have been covered appropriately is time consuming and expensive. This is due to the gap between requirements specifications, typically expressed in non-formal notations, and tests.

This thesis presents an investigation to close such a gap and provide automatic test case generation based on requirement specifications. Industrial constraints in general do not allow the introduction of radical changes in the established product development processes, therefore three progressive adoption steps are suggested. Each step is a refinement of the previous one, with the first step establishing a controlled natural language in requirement specification. The second one moving on specification of requirements as functions, and the third one taking advantage of model-based testing, creating a model of the system and after traversing it deriving test cases.

The concept has been partly applied to the industry with the help of Bombardier Transportation in Västerås. A controlled natural language has been developed and applied in the test specification within the company, which allowed the generation of test scripts with the help of a self-developed parser and test script editor. Moreover, a small internal case study has been conducted aiming at verify the feasibility of introducing a controlled natural language for requirements definition and its possible benefits. The results not only show a realistic step forward towards an automated system level testing approach, but also that the quality and understandability of the requirement specification can be gradually improved as a side effect of the automation effort.
ACKNOWLEDGEMENTS

First of all we want to thank Bombardier Transportation AB for providing us with this thesis and thanks to all the employees that have helped us with all the questions and discussions. We would like to give a special thanks to our company supervisor Manuel Palmieri for his endless support and encouragement to reach new levels, to Luciana Provenzano for the extensive work she did on understanding the state-of-the-art and providing us with valid input and to Ola Sellin for the opportunities and discussions. Secondly, we would like to thank our academic supervisor, Antonio Cicchetti, for the limitless support during the thesis and the super-fast responses on our thoughts.

I, Robert Gustavsson, would like to give a special thanks to my parents and siblings for your encouragement during my studies, your life-long support and for your love. Finally, I would like to dedicate this thesis to my daughter Lo, I love you.

I, Dimitris Kostopoulos would like to express my gratitude first to my family that supported my studies, gave me the chance of gaining more knowledge and meet people from all over the world. Special thanks to all the people I met in Västerås for the nice time that we spend together. Last but not least, I would like to thank all my friends in Greece, Nasos, Kyriakos, Iosif and a lot more, for their interest and support.
INDEX OF FIGURES

Figure 1: Visualization of the V-Model and its phases ........................................ 6
Figure 2: Visualization of the parallel development model .................................. 7
Figure 3: Visualization of the evolutionary development model .......................... 7
Figure 4: Requirement decomposition in levels starting from a higher level on top [8] 10
Figure 5: Division of Software Requirements Engineering. .............................. 11
Figure 6: A comparison between white-box and black-box testing ...................... 15
Figure 7: Driver script in capture/playback approach [11] .................................. 20
Figure 8: Driver script of data-driven approach [11]. ....................................... 21
Figure 9: Input data file of data-driven approach [11] ..................................... 21
Figure 10: Driver script of keyword-driven approach [11] ............................... 22
Figure 11: Input data file of keyword-driven approach [11] ............................... 23
Figure 12: Ontograph with the legend explaining the language to the right and a description of mini world to the left [30] ........................................... 26
Figure 13: Normal requirements and their representation in template form, then FRL notation and lastly model illustration [14] ........................................... 27
Figure 14: Scenario format of requirements as defined in [29] ............................ 27
Figure 15: Flow of actions to generate Use Case sequences [20] ......................... 29
Figure 16: Individual requirements and their respective SDL model [35] ............... 30
Figure 17: Sample contracts of two use cases [15] ........................................... 32
Figure 18: An extracted UCTS [15] ............................................................... 32
Figure 19: Sample CTS [16] ........................................................................... 33
Figure 20: Sample IFA [17] ............................................................................ 34
Figure 21: Test paths acquired after the generation algorithms for the sample IFA [17] 34
Figure 22: Scenario sample taken from [22] .................................................... 35
Figure 23: Steps used to automate test cases from requirements in [42] ............... 36
Figure 24: Steps used to automate test cases from requirements in [51] ............... 37
Figure 25: Requirement model [18] ................................................................ 38
Figure 26: Overview of Bombardier Transportation life-cycle model .................... 41
Figure 27: A set of units inside of the TCMS .................................................... 42
Figure 28: The actual hardware of a CCU-O ................................................... 43
Figure 29: Example of a test cases written in the regression test specification ....... 48
Figure 30: Architecture of VCS and communication with Test Object. ............... 49
Figure 31: Hardware components building up Test Station ............................... 49
Figure 32: Desktop view of the VCS ............................................................... 50
Figure 33: GUI of TSE ................................................................................... 51
Figure 34: GUI of TE ................................................................................... 52
Figure 35: GUI of TEM ................................................................................ 53
Figure 36: The application flow of the developed concept. ............................. 55
Figure 37: Pattern template for requirement specification [34] ........................... 56
Figure 38: Categorization of verbs used inside requirement sentences [34] ........... 56
Figure 39: Comparison of the new and old architecture of the TSE ................. 60
Figure 40: New GUI of TSE ......................................................................... 61
Figure 41: Print screen of failure of parsing a test specification, faults shown with letter E. .................. 64
INDEX OF TABLES

Table 1: Summary of the found research approaches and mapping of their content to valuable comparison attributes.................................................................................................................................................. 39
Table 2: Refinement of TRS into SRS .................................................................................................................................................................................................................. 44
Table 3: Ambiguous Requirement ......................................................................................................................................................................................................... 45
Table 4: Usage of the different keywords for the same condition ........................................................................................................................................................................ 45
Table 5: Usage of the different keywords for the same state ........................................................................................................................................................................ 46
Table 6: Overlapping requirements ............................................................................................................................................................................................................. 46
Table 7: A CNL developed to be used inside of a test specification ..................................................................................................................................................... 63
Table 8: Lamp test function before applying the concept ................................................................................................................................................................. 66
Table 9: Lamp test function after applying the concept ................................................................................................................................................................. 67
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>Aktiebolag</td>
</tr>
<tr>
<td>ACL</td>
<td>Attempto Controlled Language</td>
</tr>
<tr>
<td>BT</td>
<td>Bombardier Transportation</td>
</tr>
<tr>
<td>CIM</td>
<td>Computer Independent Model</td>
</tr>
<tr>
<td>CNL</td>
<td>Controlled natural language</td>
</tr>
<tr>
<td>CPL</td>
<td>Computer Processable Language</td>
</tr>
<tr>
<td>CSP</td>
<td>Content Service Provider</td>
</tr>
<tr>
<td>DCU</td>
<td>Drive Control Unit</td>
</tr>
<tr>
<td>DOORS</td>
<td>Dynamic Object Oriented Requirements System</td>
</tr>
<tr>
<td>EFSM</td>
<td>Extended Finite State Machine</td>
</tr>
<tr>
<td>FRL</td>
<td>Formal Requirement Language</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>IDU</td>
<td>Interface Display Unit</td>
</tr>
<tr>
<td>IFA</td>
<td>Interaction Finite Automaton</td>
</tr>
<tr>
<td>IO</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IOD</td>
<td>Interaction Overview Diagram</td>
</tr>
<tr>
<td>IUT</td>
<td>Implementation under test</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-driven engineering</td>
</tr>
<tr>
<td>MVB</td>
<td>Multifunction Vehicle Bus</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>PCU</td>
<td>Passenger Communication Unit</td>
</tr>
<tr>
<td>PENG</td>
<td>Processable English</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>SCENT</td>
<td>Scenario-based Validation and Test of Software</td>
</tr>
<tr>
<td>SDL</td>
<td>Specification and Description Language</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>SOLID</td>
<td>Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion</td>
</tr>
<tr>
<td>SRS</td>
<td>Software requirement specification</td>
</tr>
<tr>
<td>SSD</td>
<td>System Sequence Diagrams</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modelling Language</td>
</tr>
<tr>
<td>TAF</td>
<td>Test Automation Framework</td>
</tr>
<tr>
<td>TCMS</td>
<td>Train control and management system</td>
</tr>
<tr>
<td>TE</td>
<td>Test Engine</td>
</tr>
<tr>
<td>TEM</td>
<td>Test Execution Manager</td>
</tr>
<tr>
<td>TLG</td>
<td>Two-Level Grammar</td>
</tr>
<tr>
<td>TRM</td>
<td>Testable Requirements Model</td>
</tr>
<tr>
<td>TOTEM</td>
<td>Testing Object-oriented Systems with the Unified Modeling</td>
</tr>
<tr>
<td>TRS</td>
<td>Technical Requirement Specification</td>
</tr>
<tr>
<td>TSE</td>
<td>Test Script Editor</td>
</tr>
<tr>
<td>UCTS</td>
<td>Use Case Transition System</td>
</tr>
<tr>
<td>UML</td>
<td>Unified model language</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>VCS</td>
<td>Vehicle Control Simulator</td>
</tr>
<tr>
<td>VDM</td>
<td>Vienna Development Method</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction .................................................................................................................. 3
   1.1 Contribution .............................................................................................................. 3
   1.2 Bombardier Transportation AB ............................................................................... 4
   1.3 Thesis Outline ......................................................................................................... 4

2. Background .................................................................................................................... 5
   2.1 Software Development Lifecycles ........................................................................... 5
      2.1.1 Sequential Models ......................................................................................... 5
      2.1.2 Incremental models ...................................................................................... 6
      2.1.3 Evolutionary models ..................................................................................... 7
      2.1.4 Iterative Development Methods .................................................................. 8
   2.2 Requirements Engineering ..................................................................................... 9
      2.2.1 Requirement Development .......................................................................... 11
      2.2.2 Requirement Management .......................................................................... 12
      2.2.3 Requirement Characteristics ....................................................................... 13
      2.2.4 Requirement Specification Notation ............................................................ 13
   2.3 Software Testing .................................................................................................... 14
      2.3.1 Testing Techniques ....................................................................................... 16
      2.3.2 Manual Testing ............................................................................................ 17
      2.3.3 Automated Testing ....................................................................................... 18
      2.3.4 Automation Framework Approaches ............................................................ 19

3. State of the Art .............................................................................................................. 24
   3.1 Research Method ................................................................................................... 24
   3.2 Natural Language .................................................................................................. 25
   3.3 Model-based ......................................................................................................... 29
   3.4 Scenario Based ..................................................................................................... 31
   3.5 Formal Specification Language ............................................................................ 35
   3.6 Model-driven Engineering ................................................................................... 37
   3.7 Conclusions .......................................................................................................... 38

4. State of the Practice ..................................................................................................... 41
   4.1 Bombardier Transportation ................................................................................... 41
      4.1.1 Train Control and Management System ....................................................... 42
4.1.2 Requirements Engineering ................................................................. 43
4.1.3 Testing ............................................................................................... 46
4.1.4 Test Environment ............................................................................. 49
5 Concept ................................................................................................. 54
  5.1 Initial Automation Step ......................................................................... 55
  5.2 Second Automation Step .................................................................... 57
  5.3 Final Automation Step ......................................................................... 58
6 Case Study - Bombardier Transportation AB .......................................... 59
  6.1 Towards the Adoption of the Concept ................................................. 59
    6.1.1 Preparation of the Test Script Editor ............................................ 59
  6.2 Applying the Concept ......................................................................... 61
    6.2.1 Applying Step One ......................................................................... 62
    6.2.2 Applying Step Two ......................................................................... 64
    6.2.3 Applying Step Three ...................................................................... 65
    6.2.4 Example of Applying the Concept .................................................. 65
7 Discussion ............................................................................................... 68
8 Concluding Remarks .................................................................................. 69
9 Bibliography ............................................................................................. 70
1 Introduction

Companies producing software are trying to adapt to the market demands that request early delivery of a working product. Hence, different methods are applied having as a purpose to boost the development process, like code generation from design elements or automating the testing phase. More related to automated testing, which mainly concerns the current thesis, companies of the industry are moving forward by including test automation frameworks and adjusting their testing processes around them. Those frameworks allow the creation of test scripts (test cases), their execution on the System Under Test (SUT) and the analysis of the execution results. The benefits acquired from such a method seem to be twofold, as they affect both the cost spent on testing and the product itself. The cost is reduced as the time spent on test execution is limited, especially during regression testing, when the tests should evaluate if any change of code resulted in unintended behavior. The product is improved as the confidence for its correct behavior is raised, as more defects can be found and more coverage can be achieved.

However, the automation is built normally on top of the actual code, thus it relates the testing with the implementation phase. Therefore, the proof that the delivered product as a whole acts as expected, is missing. In order to achieve such a proof system level testing is required, with the aim of verifying the correct implementation of requirements. By automating test cases based on the requirements, without any manual steps, the time-to-market could be decreased. The missing connection between requirements and testing inside the context of a test automation framework is the problem of the thesis that is further explained in the following subsection.

1.1 Contribution

This thesis provides an analysis of the state-of-the-art on the topic of requirements and testing but also presents realistic results coming from the application of a defined concept inside an industrial partner. The analysis part focuses on answering the research related questions arising on the topic, as follows:

1. What are the available techniques that enables generation of test cases targeting system level from requirements?
2. How should the requirements be specified to allow the usage of such methods?
3. In what amount does the manual work of the tester reduce?
4. In what manner can the techniques be used in an industry?
5. With the gathered knowledge, what is the most suitable way for an industry to move towards automatic generation of test cases targeting the system level from requirements?

After answering those questions from a theoretical point of view, we investigate their introduction inside Bombardier’s industrial environment, as a proof of concept. Thus, the applicability of research methods is verified against a real industrial scenario in order to investigate if the industrial environment is mature enough to adopt one of the proposed approaches. As a consequence, in case of immaturity in
particular areas we investigate smooth ways for introducing appropriate techniques enabling the automated generation of tests from requirement specifications.

1.2 **Bombardier Transportation AB**
Bombardier is the company in which the thesis was carried out from middle of January to the middle of June. It is a Canadian worldwide company accepting and building projects in the areas of aerospace and transportation. Specifically the two authors spent those five months in the rail equipment division called Bombardier Transportation (BT) in Västerås, where they analyzed and got familiar with the internal processes. The company’s development process, documentation and working environment constituted the inputs, which led the focus and guided the results.

1.3 **Thesis Outline**
The thesis starts in section 2 by providing the needed knowledge to understand the upcoming sections of the thesis, namely Background. After that, the current state-of-the-art is presented in section 3, while in section 4 the state-of-the-practice is presented in terms of way-of-work and techniques exploited inside Bombardier. Section 5 describes a general concept to go from requirements to test cases divided into three steps and section 6 is a case study where the concept is applied to Bombardier. Eventually, section 7 draws a discussion on the work presented in this report and possible future investigation direction, while concluding remarks are presented in section 8.
2 BACKGROUND

Products relying on software are increasing rapidly in the industry and therefore the quality and time-to-market demands are getting higher and higher. Therefore, rapid development processes are needed while ensuring that the quality of the product is preserved when not even improved. Furthermore, there exists a widespread necessity such that the requirements shall be written in an understandable way, so that stakeholders can understand and agree on the characteristics of the resulting SUT. To be more precise, as stakeholders we refer to everyone who can have an impact on the requirements definition, like the customer, users, managers or external stakeholders like unions or authorities. This section provides the information to understand the basics when it comes to development lifecycles, requirement engineering and testing and it will be useful to make the understanding of the upcoming sections in this thesis.

This section first gives an introduction of the software development lifecycles, and then discusses a broader view of requirement engineering topic, and the testing subject.

2.1 SOFTWARE DEVELOPMENT LIFECYCLES

When developing software a conceptual model is used to describe the lifecycle of the development. The model can be seen as a framework that describes the activities done during a project and it can be described in different manners, either sequential, incremental, evolutionary or a mixture between two or more. The main activities carried out during a project are the same for all models and they involve description of the software, extraction of the requirements, documentation, system representation, code testing and finally maintenance.

2.1.1 Sequential Models

Sequential models rely on the fact that the problem to solve can be described and designed before the implementation starts and is going left-to-right and only once. These models were the first software development lifecycle models, based on best practices and are still commonly used inside some large organizations. The most famous models that pertain to the sequential category are the Waterfall model and the V-model [6].

The waterfall model is the primary model of the sequential model and it can be seen as a waterfall because it starts with the requirement specification and one step at the time it goes downwards to the next phase of the project. When a phase is ended it should not be redone, this however is often impossible because of the knowledge gained in the upcoming phases.

In industry one of the most commonly used model is the V-model, since it clearly specifies the testing activities that are performed during the development and it is easy to understand. The model promoted the inclusion of testing activities in an early state of the process and this leads to minimization of project risks and improvement of the product quality [6]. The V-model is an extension of the waterfall model in the sense that instead of ending at coding it goes upwards again and into the testing part of the process. As seen in Figure 1 the left hand side and the right hand side have a connection in the sense that tests to be performed in the development process are planned when traversing a level on the left hand side and are then performed when that level is reached on the right hand side. Therefore, the left side can be seen as the software development phases and the right side as the testing phases.
2.1.2 Incremental models

Incremental models are sequential models in the sense that all the requirements are known from the beginning, but then the project is divided into subsystems called incrementals [6]. Each incremental is a small set of the final product and contains a sub part of the requirements. For each incremental more and more requirements are added until all of them are met. In the end of each incremental a small release can be made which enables early feedback from the customer. Because of this, the quality of the product can be increased since the customer can at an early state comment on what is good and what is not [6]. The division into incrementals is usually done based on the risks, architecture and/or the requirements. Two famous models are the Stage Delivery Model and the Parallel Development Model.

The parallel development is relying on the fact that you define the requirements and then the design. After these phases the system is divided into subsystems. As opposed to stage delivery model, that develops each subsystem sequentially, the parallel development model prescribes to develop them in parallel. As seen in Figure 2, when each subsystem is developed the integration test is done and then validation test is executed.
2.1.3 Evolutionary models

Evolutionary models are different from the incremental and sequential in the sense that requirements do not have to be completely known before the project can be started. These kinds of models are good to use in research or when a technology is developed since often it is just produced a prototype. The evolutionary models are also done in increments but after each increment the requirements are defined or refined. One of the most known evolutionary model is the Spiral Model and it can be seen as a spiral, starting in the center and each cycle corresponds to one iteration of the development process. This model often uses a sequential model as backbone and for each cycle a prototype is created. This prototype can then either be accepted and the next cycle will then be built upon it, or thrown away and a new prototype is created from scratch in the upcoming incremental.

Figure 2: Visualization of the parallel development model

Figure 3: Visualization of the evolutionary development model
2.1.4 Iterative Development Methods
Inside each phase of the development life-cycle there are methods to fulfill. Best practices in software engineering often suggest to adopt the iterative development method [2]. It can be used with any of the models described in this sections but it is commonly used inside of the incremental development model. It is worth noting that iterative methods and incremental models are not the same thing. In fact, the iterative method is a way to refine and improve software over time while the incremental model is an approach to develop a specific part of the system for a known amount of time. Organizations have refined the iterative methods and created their own types, two of these are the agile and lean software development.

2.1.4.1 Agile Software Development
Agile development methods are a typical examples of the combination of iterative methods and incremental models. These methods are more and more used in the industry and in particular for small or medium size projects\(^1\). Agile is focusing more on the implementation than the documentation, as the code is considered as the documentation. The agile method has been specified by the agile manifesto and carries by twelve principles that shall be followed [47]. After its introduction, several methods have been proposed as implementing the agile approach, like Scrum, Kanban and Extreme Programming (XP) [6].

2.1.4.2 Lean Software Development
The term lean was created in mid 1980’s in manufacturing vehicles at Toyota [7]. However, in the 1990’s lean was “converted” to software development instead of product manufacturing. This was possible because the creator was seeing lean as a set of principles and not a set of practices. Lean software development prescribes seven principles:

- **Optimize the whole** means that a deep understanding in the customer needs is important and that the process must be seen as a whole, it is not only the development that can be improved.
- **Eliminate waste** stands for that everything that does not bring added value to the customer or needed knowledge is seen as waste and effort shall not be spent on that task or feature.
- **Build quality in** associates to continuous integration which in itself means that instead of integrating the software component at the end, it is done throughout the project, repeatedly.
- **Learn constantly** is reached using two ways. The first is to explore different solutions for the project like programming language, architecture etc. and then choose and the second one is about having the minimum setup from the beginning and continuously add what is needed during the project.
- **Deliver fast** signifies that the software shall have a short time range between the releases.
- **Engage everyone** intends to keep the information technology department connected to the business unit and they should not be seen as a separate departments in the sense of project vision and choices.
- **Keep getting better** implies that the company shall always aim to become better. It is for example good practice to start with Scrum or XP and then move towards a Lean development.

\(^1\) Projects that lasts approximately 1 - 5000 hours
This software development method is sometimes considered as an agile method but according to [7] there are differences. One major difference is the scope of the method, agile deals with software development only while lean is a more general vision of to approach the whole process. Also, the roles are somehow different. Agile typically has a customer / product owner that is steering the software development in the desired direction while in lean there exists a role that is more like “an entrepreneur for a startup” that handles the product development [7].

2.2 REQUIREMENTS ENGINEERING

“The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed” [5]

A requirement [4] is:

- (1) A condition or capability needed by a user to solve a problem or achieve an objective.
- (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
- (3) A documented representation of a condition or capability as in (1) or (2).

Requirements is the common language and the simplest intermediate that engineers use in order to connect the customer demands and needs with the different project members like managers, developers, etc. Since various audiences need to share the same knowledge, many levels and types of requirements are used, as shown in Figure XX.. In general, there is no agreement on requirements decomposition between research community and industry. Furthermore, different industrial domains may present variations in the decomposition due to domain-specific constraints. Having in mind the multiple criteria that could be used to handle requirements, notably process based, through technical analysis and product based, requirements can be classified in disparate ways. In the scope of this report we will only consider the different types of requirements related to the development process. In particular, we will deal with business, user and functional requirements, which correspond to different phases of the product development, i.e. the feasibility study, the system analysis and the system design, respectively.

Business requirements describe the reasons that led a company to desire the future developed system. The reasons can be expressed as goals, objectives or benefits that the organization wants to reach or achieve. In other cases the reasons can be defined as needs that the organization has or problems that have to be solved. Business requirements are valuable also as input to the feasibility study conducted in the early phases of the project. During the feasibility study the engineers aim to find out if the system can be implemented and if it is worth being implemented for the customer’s sake.

After the first level of requirements comes the user oriented requirements. This level aims to identify the tasks that the user can perform in the system in order to reach a goal. User requirements are usually expressed as use cases or user stories, which can be understood by the customer. This type of requirements is also referred as stakeholder requirements, since except the users other involved stakeholders express their needs. During user oriented requirement handling, engineers also find out
characteristics, called quality attributes, with the purpose to provide added value to the user satisfaction.

The last level as it can be seen in Figure 4, contains the functional requirements. As "functional" suggests, these requirements deal with a more detailed description of the system behavior. This detailed description gives enough information to the developer to realize user tasks. However, in order to express functional requirements more inputs are expected than the user requirements. One of the required additional inputs are external interfaces, which imply other software or hardware systems affecting and being affected by the system. Additional inputs are business rules related to constraints in the development process or standard compliance and more general constraints like timing or statements of stakeholders. All the additional inputs to the functional requirements like business rules, quality attributes, external interfaces and constraints establish the definition of non-functional requirements. Last but not least are the system requirements, which compose functional requirements towards the representation of a more complex behavior.

The whole process of identifying, studying, documenting and maintaining the services and constraints is known as requirements engineering. Requirements engineering process consists of two different activities called requirements development and requirements management, as seen in Figure 5. Both of those activities are further separated to reach a more precise and simpler process. Thus, requirements development is divided into four phases, i.e. elicitation, analysis, specification and validation, while requirements management is divided into planning and changing management.
2.2.1 Requirement Development

Requirement development is done usually in the beginning of the whole development process, since requirements are a description of the system to implement and companies prefer to start with the requirement phase. This is valid for both sequential models and incremental models but not for the evolutionary ones. Moreover, since it is hard to specify a system in its completeness from the start the requirement phase is usually gone through several times before a system is released. This is typically due to the fact that the requirements were not clear enough from the beginning or a request of change arrived from the customer.

Elicitation is the phase when the identification of the requirements is completed. This can be achieved by continuously interacting with system stakeholders in order to reveal the system users and their goal, as well as the environment under which these goals are fulfilled. The system engineers use a variety of methods to discover the requirements, like interviews with stakeholders, creation of scenarios and prototyping.

During the second phase, named analysis, the engineers should obtain a better perspective of each requirement, which can lead to a more detailed description. A better description can reveal dependent requirements that can be grouped together and mapped directly to sub-systems. This decomposition and allocation is also the reason why requirements engineering cannot exist without relating its outcomes to system architecture. Moreover, during the analysis phase the identified requirements should be prioritized having in mind customer needs and their importance inside the system, but also carefully negotiated with the stakeholders to remove conflicts. Finally, the quality attributes should be taken into account.

Figure 5: Division of Software Requirements Engineering.
So far the previous processes did not mention anything about documenting the findings of the engineers. This is the objective of specification phase, when the different levels of requirements are written and result in the requirements document. The requirements document also known as software requirement specification (SRS) should contain user and system requirements. The SRS should be understandable by different audiences like system users, developers, testers and customers, thus extra care should be taken in order to correctly present the different requirement levels. There exist also cases for big projects where the number of requirements is massive and the user requirements have to be presented in a separate document.

Requirements can be specified using different styles. The most common style to represent requirements is to write them in natural language like English. However, there are other ways to represent requirements, for example by using unified model language (UML) to represent them graphically in different kinds of charts; by making an abstract model of the system itself and showing its behavior in a graph-based model; or by specifying them using a logical or algebraic form, such as a formal language.

The last phase of the requirements development is named validation. Throughout this process the engineers verify that the specified requirements represent the functionality of the product desired by the customer and also if the correct principles were followed in the specification. Numerous techniques can be used to support the validation like requirements reviews and test-case generation to check the specification, and prototyping to confirm the compliance with the demands of the customer. The validation process is very important due to the increased cost devoted to a requirement changes, which results to a design and an implementation change but also demands a new testing phase [8].

2.2.2 Requirement Management

In the majority of software projects the initial specified requirements are never the same as the requirements at the end of the project. The simplest scenario explaining requirement changes is when there are conflicts between stakeholders. Conflicts can be created when the customer is not the actual user, thus after the system installation or the alpha testing the users demand more or different features. There are also occasions when the customer constraints change over time, for instance when the external interfaces are modified.

The development team needs to cope and prepare for requirement changes with a process called requirements management. As first step in the management process, the engineers should define two main sub-processes, management planning and change management. The former is composed of steps that set the infrastructure of the management process. This infrastructure defines a policy for requirement identification and traceability both between requirements and also between requirements and software design. Although traceability becomes a lot easier when a requirement management tool is used, some development teams use different ways to maintain the requirements, like spreadsheets.

As far as the change management is considered it consists of three stages illustrating the sequence of actions starting from the proposal for change to its completion. The first stage begins with a requirement change request, which is later reviewed for its credibility. Depending on the analysis results the responsible engineer or committee decides if the request will be realized. Moving on to the second stage, the company evaluates the benefits of applying the change compared to the cost of updating the design and the implementation. One of the important criteria for measuring the cost is traceability of
requirements, which can be used in the identification of their relations and visualize what chain of reactions will occur if one of them is changed. The final stage contains the activities that lead to the realization of the changes in the implementation and design, as well as the proper documentation of them [9].

2.2.3 Requirement Characteristics

When the time comes to document the requirements, the engineers should follow the defined rules set up by the company to write them down. Those rules specify a management tool and policies relating to traceability as mentioned in the requirement management section. However, except the company’s rules the engineers should respect and take into account some informal requirement characteristics, when they report them. A list of those characteristics is the following [8]:

- **Complete**: All the required information should be included in the requirements, letting the reader comprehend fully the intended behavior. If some information is not currently filled in then the abbreviation TDB (to be determined) should be used.

- **Correct**: The engineer should make sure that each requirement specifies a behavior that realizes a stakeholder’s need. Thus, traversing up to business requirements or even contacting the customer will be essential.

- **Feasible**: Having in mind identified product constraints, like usage scenarios, and project limitations as the available budget, each requirement should be specified only if it can be realized.

- **Necessary**: A requirement should be specified only if its functionality is expected from any of the stakeholders or if it adds value as it applies to a regulation or standard.

- **Prioritized**: Each requirement should be prioritized, so the engineers are aware of its importance to the stakeholders and also plan for the functionality of each release.

- **Unambiguous**: Natural language is one of the approaches used in the requirements specification, as it makes it easier for the engineer to specify the requirements and express with understandable words the functionality. However, natural language is also the cause of ambiguity in the occasions when the same sentence can be interpreted differently.

- **Verifiable**: It is crucial for a project’s success to verify the intended behavior of the product. This can be accomplished if the testers can confirm that the developers implemented correctly a requirement. Hence, the requirement should be written in a proper way that facilitates its verification.

2.2.4 Requirement Specification Notation

This thesis work explores the possibilities of building a (partially) automated interconnection between requirements and test cases. Therefore requirements will be considered the only input available to perform functional verification. In this respect, it is worth revealing the most common and preferred approaches to document requirements across companies in the industry. Two surveys were found describing the state of the practice on requirements engineering in general. The first [25] is focused more on small and medium sized enterprises while the second [24] does not address a specific company size. A careful examination of the results presented in the papers shows that both conclude in informal requirement representation as the most preferred with semi-formal as the second. Specifically, [24]
which has the biggest sample reaching 194 responses, claims that 51% uses informal methods (i.e. natural language), 27% semiformal and only 7% formal methods. Trying to find out if the absence of formality is affecting the product quality, the surveyed responded that the customers found the product both easy to use and meeting their needs. The percentage of the responses was varying from 79% for informal to 60% for formal methods.

After identifying that the trend is the usage of informal requirement specification and as [25] supports it is mostly natural language, we should also detect the reasons. In [26] some reasons are gathered, like easiness of communication with stakeholders for both elicitation and specification purposes. Moreover, the requirement engineer is not demanded to learn complex languages, meaning that the education cost spent by the company is less. In addition, since natural language is the normal way of expression for human beings the whole process is more flexible [27].

Even if the requirement specification in natural language sounds beneficial and is proved successful, it has its own deficiencies. These deficiencies explained as attributes according to [27], are incompleteness, inconsistency and imprecision. Since, business analysts write down the requirements in the first place, there is a high chance that domain information will be missing, resulting in ambiguity when it comes to the steps of design and implementation. Requirements can also be incomplete in the manner of missing entity attributes or action constraints. Obviously, this leads in the end to an insufficient delivered product. Considering the misinterpretation that can occur in the normal development flow (requirements -> design -> implementation), the difficulty to jump directly from requirements to testing, a later phase than the implementation, is noticeable.

The issues related to requirements specification discussed so far have gained a lot of attention by the research community. Hence, there exists a relevant literature focusing either on the analysis of the specification or improving its content. In [28] the author tries to limit the misinterpretation arising between the requirement producers and the ones that use them, with the Cognitive Linguistic Elicitation and Representation method. With that method the linguistic analysis of the requirements results in simple and domain specific definitions. The author of [27] both analyses and improves the specification content as he maps information from natural language to models in a manual and in an automatic way, supported by machine learning tools. Lastly, the cases when the specification is just analyzed by algorithms without being improved, is also common. In this respect, [36] and [37] evaluate the content of the vocabulary and the syntax of sentences. If after those analyses the results are violating some rules then the respective requirements are reported as ambiguous.

2.3 Software Testing

“Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item” [3].

Software testing is one of the Verification and Validation (V&V) principles in software practices. Verification is the type of testing that makes sure that the product is built in a correct way and validation makes sure that the correct product is built. Therefore, verification is usually done throughout the project to test the functions behavior while validation is done when the development phase is done to
validate that the implemented products have the specified behavior and that the product fulfills the requirements.

Testing is a way to provide quality to the finished artifact by verifying and validating the implementation of it. It is a very expensive and long phase in the development process and involves implementers, test engineers and test managers. The implementers are responsible for the code that has been written and have the task to find the cause of the fault if a bug is found. The test engineers are planning the tests by designing test cases in a test specification, performing the tests on the system and also verifying that the tests on the system were passed. The test manager has the role to handle the test engineers and is the main responsible that the set of performed test follows a good standard.

Testing can be divided into two main classes, white box testing and black box testing and their definition follows:

**Black box testing** (also called functional testing) is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [4].

**White box testing** (also called structural testing and glass box testing) is testing that takes into account the internal mechanism of a system or component [4].

Since white box testing is basing the test cases on the internal perspective of the application it can be used to test the internal structure or source code. Therefore, the white box testing is usually used when testing lower levels of the system. In contrast to white box testing, black box testing does not have an internal perspective and is therefore used more to test the functionality of the application. Black-box test cases are created based on the requirements or specification, named requirement/specification-based testing, and white-box is creating the test cases based on the internal structure, named code/design-bases testing. The distinction between the test approaches is presented in Figure 6.

![Figure 6: A comparison between white-box and black-box testing.](image-url)
A specification of a test is called a test case, which is a description on how a test shall be performed and the pre- and post-conditions of the system. It also contains a set of inputs to the system and the expected output based on the input. Usually, the test case also contains more information, such as a pass and fail criteria of the test. Multiple test cases can be put together to form a more complex testing scenario to be executed as a whole, typically referred to as test suite. These test cases are often specified in a test specification document which is written throughout the project. The test specification is often divided by sub-system or functions to have a good structure of the tests. When a test is performed, a test report shall be written that explains how the test was carried out and also why it passed or failed. This test report is then a proof that the system has been tested accordingly.

2.3.1 Testing Techniques

To achieve good quality products the organizations are testing the system in different manners. There are various techniques available, but this section will focus only on a selection of them that is important to know about in the scope of this thesis, i.e. unit testing, function testing, acceptance testing, integration testing and regression testing, and give an introduction of what they are, what they aim for and finally what they are based on.

**Unit Testing**

**Definition:** “Unit testing is the testing of individual hardware or software units or groups of related units” [4].

It is a pure white-box testing technique that is based in low-level design and/or code structure. Unit testing usually aims at testing a very small part of the implementation, often just a function. When the tester is performing this kind of testing a set of inputs are prepared for the function and as a pass and fail criteria the expected output is used.

**Function/System Testing**

**Definition:** “System testing is testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements” [4]

It is on the other hand a pure black-box testing technique that is based on the requirement and/or the specification of the system. This kind of testing is carried out with the full implementation of the system and therefore multiple parts of the system can be tested at the same time. System testing is usually performed by someone with an independent perspective of the system to ensure the system is tested as a user without any bias due to internal knowledge.

**Acceptance Testing**

**Definition:** “Acceptance testing is formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system” [4].

It is also a black-box testing technique and is based on the requirement specification. Acceptance tests are sometimes pre-defined by the customer, a certain number of tests shall have passed before the system is handed over. However, since customers usually are not a testing experts, more testing is needed to have confidence in the release.
Integration Testing

Definition: “Integration test is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them” [4].

It can be used as both a white and a black box testing technique and depending on the technique used (white- or black-box) it is based on different parts of the system. If it is used in black box it is based on high-level design whereas if is white box it is based on low-level design. Integration test is a way to ensure that the components that have been created have the correct behavior and communication between each other. In other words, it is useful to test the system behaves as expected when considered as a whole.

Regression Testing

Definition: “Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements” [4].

It can be used as both a white and a black box testing technique and depending on the chosen technique it is based on different parts of the system. If it is used as black box it is based on high-level design and/or any change in the documentation, while if it is white box it is based on changes in the documentation. A regression test shall be run often to reassure that new features or implementations have not changed the behavior of the old ones.

2.3.2 Manual Testing

The strategy followed when testers proceed from one testing level to another is most of the times completed manually. As a consequence, testers create their individual test cases using the appropriate techniques in order to discover defects. Later they execute the test cases in the SUT and verify the results for compliance with the expected outcome. On one hand this procedure lets the engineers get a broader overview of the system and also allows them to include their own interpretation and knowledge in the test cases. On the other hand, when the complexity of the system grows costs and error-proneness become a relevant issue.

There are two different approaches supporting the manual testing, and in particular aiming to facilitate the test case execution process and hence the actual execution of the tests. These two approaches are code-driven testing and graphical user interface testing [13]. For both the approaches the testers or developers (role depending on the development process) take advantage of tools or frameworks that handle the automatic execution.

Code-driven testing/automation is closely related to the white-box testing method, due to the need of verifying the source code’s behavior under specific circumstances. This type of automation approach is used to make sure that different software modules, like functions, classes or libraries run as expected. Depending also on the time at which the test cases are created the tools allow the key principle of agile development test-driven development to be realized. The broadly used frameworks supporting code-driven testing are products of the family xUnit created for various programming languages, like jUnit for Java, NUnit for .Net, pyUnit for Python, etc. The user of those frameworks creates routines, based on
the test cases, which are executed sequentially. Their purpose is to trigger a specific behavior by invoking objects and verify the output or the state of the system.

The second approach, graphical user interface (GUI) testing, has also as a concept the creation of test routines, but differs both in the test creation procedure and the required skills. Firstly as the name implies, it focuses on the graphical user interface of the applications, which implies that the application should be in a runnable state. Most of the tools benefit from a method called record and playback that tracks the actions of the tester and repeats them to verify the state of the system. Lately, the applicability of those tools has been extended to include web pages other than desktop applications. Record and playback method is pretty forward since no coding skills are expected and the tests can be created quickly. However, it affects negatively the maintenance and reusability of tests, in fact if a user interface element’s attribute is changed then the test has to be re-recorded. In order to overcome these problems more tools have been introduced following an approach known as scriptless test automation, where the system is built as a model and the tester creates test cases by adding values to it.

2.3.3 Automated Testing

In the last decade various research institutions tried to measure the economic impact of software failures and one of them, the National Institute of Standards and Technology (NIST), had results for the United States economy. According to NIST software bugs or errors cost annually 59.5 billion dollars, an amount translated as 0.6 of the country’s gross domestic product [44]. Taking into consideration the increasing demand of delivering the products at shorter time to the market, a different approach needs to be considered to support the testing process. A straightforward solution is to introduce some form of automation in the testing phase.

The previous section described the automation of the test cases, but when it comes to automated testing the whole testing process is implied combining test case generation and test result verification [12]. An automated approach can be beneficial for a company and reduce the cost to market, the cost of failure and also strengthen the product reliability [11]. To begin with, automated testing improves the cumulative coverage, which is important when the system evolves through time and becomes more complicated. In this way, the tester can verify and validate how even a small change affected the system behavior. Furthermore, even if the manual testing allows the tester to set up complex test cases more appropriate to reveal defects, automated testing has more chances in creating test cases or scenarios that the tester could not think of either by misinterpretation or by lack of domain knowledge. The wide range of generated test cases can lead to better coverage and identification of defects in earlier stages, thus preventing them from resulting in failures. In numbers less failures imply reduced cost of failure for the company.

So far the automation of testing sounds beneficial, but the real challenge begins when a method should be realized and applied as an automated process. As a solution a tool can be built, like the ones presented in the previous section that provides the testers with the means to conduct tests and produce results. However, there is a chance that each tester will use her/his own interpretation in both producing the tests and reporting the results. Moreover, the maintainability of the whole process flow will be hard to manage. Not to mention the case when the company builds safety critical systems and
must prove that a process is complying with the regulations or authorities. An alternative solution that can be created programmatically too, is to adopt a framework as a solid base for the testing process.

2.3.4 Automation Framework Approaches
An automation framework is an environment for manipulating the testing activity, including test script creation, management and execution. As mentioned previously the execution should not be the only concern, hence the framework should contain libraries for analyzing the executed tests and generating reports with the results. A framework seems valid for resolving the mentioned issues. For instance, the company is ensured that all the testers use common libraries and that the results are presented in a similar way. Reusability of components is an extra advantage, since the testing libraries could be included in different projects [10].

2.3.4.1 Capture/Playback
Capture and Playback is an approach that allows the recording of testing actions and their automatic repetition. The main purpose of the approach is to increase the reliability that the GUI is responding as expected. Therefore, the initial step for the tester is to manually perform the tests while in parallel the automation tool captures the inputs and outputs in the background. When the capturing phase is complete the same steps can be repeated later by the tool and any different response is considered as an error, which should be logged and reported. An example of a driver script (test script) resulting from the capture and playback approach is shown in Figure 7. The driver script contains instructions about the actions followed by the tester and also documents details of the used GUI elements acting as inputs.

In order to use this approach efficiently inside the testing phase of a development process a set of rules should be specified. Those rules guide the distribution of tests and prevent duplication of effort. One rule states the relation of one driver script per requirement and a second one the allocation of separate functions to separate testers. This allows easier allocation of requirements to testers and easier maintenance of driver scripts. If those simple rules are applied the advantages of the approach are observable. To begin, the cost of educating the personnel is low, since the training period can be short and the tools are straightforward to use. Moreover, the creation time of a driver script is short resulting in a shorter testing time that can be spent in a different type of testing or in another activity. Lastly, it supports detailed logging of failures, as for every error the triggering actions are recorded.

Paying attention to the disadvantages of the method, there are serious maintainability problems with the recorded scripts. If by any chance the window under test changes, because of an extension of the GUI elements or their placement sequence, then all the related driver scripts should be recorded again. The same applies when one driver script reveals an error that should be fixed, as the correct behavior should be captured primarily. Moreover, the approach is available for usage only when the application is stable, thus making early testing impossible.
2.3.4.2 Data-driven

Moving on, the next automation approach is called data-driven and it is an evolved method that improves the capture and playback. The structure followed in data-driven requires a manually created driver script as well, to act as the backbone of the approach. However, in this case the input and the output are not fixed. As a result, two different files are created, one containing the driver script (sequence of steps) and one filled with the used inputs and expected outputs. The second file can be referred as the data file. The automation process cannot be completed if either of the files are missing as the test script depends on the data file and vice versa. The figure 8 represents the same driver script shown in the previous method, but transformed as a data-driven testing scenario.

Data-driven has its own specific rules for the test case management, since one driver script can generate different test cases. Each record taken from the data file is considered a unique test case, as with different input different paths are followed inside the scenario. Figure 9 is a sample of the data file that supports the driver script and as it can be seen each row has its own test case id. Comparing with capture/playback method, it is noticeable that the maintainability deficiency is alleviated, since the logic for each test case has to be changed in one place and one time. Furthermore, the test cases can be created before the application reaches a stable state, by filling data files appropriately. Finally, the creation of the data file is quite simple if a program that lets the creation of spreadsheets, tables or even databases is used.

Regarding its handicaps, data-driven demands a small training period for the testers in order to acquire confidence in variable processing and test script creation. Additionally, some of the data files can become really big, meaning that their management can become time-consuming and error-prone.
Select menu item "Chart of Accounts>>Enter Accounts"

Open file “CHTACCTS.TXT”  * Open test data file
Label “NEXT”  * Branch point for next record
Read file “CHTACCTS.TXT”  * Read next record in file
End of file?  * Check for end of file
  If yes, goto “END”  * If last record, end test
Type ACCTNO  * Enter data for account #
Press Tab
Type ACCTDESC  * Enter data for description
Press Tab
Select Radio button STMTTYPE  * Select radio button for statement
Is HEADER = “H”?  * Is account a header?
  If yes, Check Box HEADER on  * If so, check header box
Select list box item ACCTTYPE  * Select list box item for type
Push button “Accept”
Verify text MESSAGE  * Verify message text
  If no, Call LOGERROR  * If verify fails, log error
  Press Esc  * Clear any error condition
CALL LOGTEST  * Log test case results
Goto “NEXT”  * Read next record
Label “END”  * End of test

Figure 8: Driver script of data-driven approach [11].

<table>
<thead>
<tr>
<th>Test Case</th>
<th>ACCT NO</th>
<th>SUB ACCT</th>
<th>ACCT DESC</th>
<th>STMT TYPE</th>
<th>ACCT TYPE</th>
<th>HEADER</th>
<th>MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>100</td>
<td>0000</td>
<td>Current Assets</td>
<td>Balance Sheet</td>
<td>Asset</td>
<td>H</td>
<td>Account Added</td>
</tr>
<tr>
<td>1001000</td>
<td>100</td>
<td>1000</td>
<td>Cash in Banks</td>
<td>Balance Sheet</td>
<td>Asset</td>
<td></td>
<td>Account Added</td>
</tr>
</tbody>
</table>

Figure 9: Input data file of data-driven approach [11].
2.3.4.3 Keyword-driven

The last approach used in test automation frameworks is called keyword-driven and it is an evolution of the data-driven. The biggest limitation of the previous approach is the tight coupling between the driver script that contains the logic and the data file. In more detail, a lot of effort needs to be spent if a new driver script is created to represent a different and more complicated test case, as the data file should be updated with new columns. A solution is given in [45] and [46], known as keyword-driven approach.

In a keyword-driven approach the concept remains the same as the data are still read from external files (Figure 11) when the driver script (Figure 10) is parsed. What really changes is the content of those files. In fact, the external files contain directives recognized as keywords that describe testing operations to be applied to the arbitrary input data. Other than that the method for storing and processing the data is similar as in data-driven. Then, the values can be saved in a database, as spreadsheets or as html tables, and they can be managed by a parser that gets the keywords with the values and maps them with specific actions.

```
Open file TESTDATA
Label "NEXT"
Read file @TESTDATA
   End of file?
      If yes, goto "END"
   Does @WINDOW have focus?
      If no, Call LOGERROR
   Does @CONTROL have focus?
      If no, set focus to @CONTROL
         Does @CONTROL have focus?
            If no, Call LOGERROR
      Call @METHOD
      Call LOGTEST
   Goto "NEXT"
Label "END"
* Open test data file
* Branch point for next test case
* Read next record in file
* Check for end of file
* If last record, end test
* Does the window have focus?
* If not, log error
* Does the control have focus?
* Try to set the focus
* Was set focus successful?
* If not, log error
* Call test script for method
* Log test results
```

Figure 10: Driver script of keyword-driven approach [11].
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Window</th>
<th>Object</th>
<th>Method</th>
<th>Value</th>
<th>On Pass</th>
<th>On Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Account</td>
<td>MAINMENU</td>
<td>CHART.MENU</td>
<td>Select</td>
<td>Chart of Accounts&gt;&gt;Enter Accounts</td>
<td>Continue</td>
<td>Abort</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>Account Number</td>
<td>Enter</td>
<td>100000</td>
<td>Continue</td>
<td>Continue</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>Account Description</td>
<td>Enter</td>
<td>Current Assets</td>
<td>Continue</td>
<td>Continue</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>Statement Type</td>
<td>Select</td>
<td>Balance Sheet</td>
<td>Continue</td>
<td>Continue</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>Header</td>
<td>Check</td>
<td>On</td>
<td>Continue</td>
<td>Continue</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>Account Type</td>
<td>Select</td>
<td>Assets</td>
<td>Continue</td>
<td>Continue</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>OK</td>
<td>Push</td>
<td></td>
<td>Continue</td>
<td>Continue</td>
</tr>
<tr>
<td>Add Account</td>
<td>Chart of Accounts</td>
<td>Message Box</td>
<td>Verify Text</td>
<td>Account Added</td>
<td>Continue</td>
<td>Continue</td>
</tr>
</tbody>
</table>

Figure 11: Input data file of keyword-driven approach [11].
3 State of the Art

This section of the thesis describes what has been found in the state of the art during the research phase. The thesis objective is to develop a concept that realizes automation of test cases from a requirement specification. The goal during the research phase was to find as many available solutions to the problem that is faced to get an extensive understanding of what options are at disposal when it comes to automation of test cases from the requirements.

3.1 Research Method

The goal of the thesis work implies that techniques allowing the automation in testing should be found and documented. Trying to collect all the possible procedures in test automation would result in gathering a huge amount of papers, which cannot be grouped or explained with the same criteria. Thus, the context of the research has been narrowed down by considering the point in time inside the development process when the concept will be applied. The initial point of the thesis is the requirements phase and the actual input is obtained in the form of a requirements document, so the keyword requirement is added in the search pattern. Even though the research area is reduced it is still not satisfying, because it does not make easier the selection of comparison criteria between the different identified approaches. To be more precise the input of the approaches can vary as it was mentioned in section 2.2.4, from informal requirement representation to semi-formal and formal. Moreover, in section 2.3 the different testing levels were specified and can be considered as another differentiating variable in the recognized approaches. This report’s research do not include as a criterion the format of the requirements, and the targeted testing level is limited to the system level.

To summarize, the constraints we applied for limiting the approaches considered in the testing automation research field are the following:

1. The test automation should begin from the requirements specification.
2. The notation used to describe the requirements is not taken into account.
3. The approaches should target the system level testing.

After defining the context of the research, we adopted the following literature review approach. Initially, the proceedings of various conferences and workshops in requirement engineering were studied, like the Requirements Engineering International Conference, Automated Software Engineering International Conference, International Workshop on Automation of Software Test and more. Later on papers from the proceedings were chosen based on the title relevance with the research topic and kept on a Google spreadsheet. Having read carefully the abstracts, introduction and conclusion of the selected papers, the list was decreased in 23 out of 64 promising papers that could be used for test automation. Those 23 papers were the first valuable sample of the research and by looking into their references and citations the list was updated with 107 more papers, promising to be used in different sections of the report, as introduction, state of the art and results. However, out of 107 papers only 64 could be referred as state of the art and out of those 64 only 15 were not violating the constraints defined before for the research context.
The method and thus the research phase was boosted the moment that surveys relating requirements with testing were found [38], [39], [40], [57]. Still, after a careful understanding the most of the approaches were already covered by a portion of the first papers. Nonetheless, the criteria followed from those surveys guided the last phase of the research and also the structure of this section. The first survey focuses on the alignment of requirements specification and testing in general, trying to figure out how those two concepts are connected in existing literature. The remaining three surveys deal with more concrete approaches relating test generation and functional requirements.

When the first set of papers was gathered, references and citations were analyzed to get a broader understanding in that specific subject or solution. After this phase, we could draw the conclusion that there are four different types of solutions out there: by natural language, models, contracts and formal specifications. In addition, all of them contain sub-solutions. The upcoming sections will describe the solutions in a more detailed manner.

3.2 Natural Language

Writing the requirements using natural language is the most common way for specifying them, since this does not involve any extra work and the learning curve is very good. However, when using natural language the requirements can be written in many different ways and therefore the text can also be misinterpreted. To avoid this ambiguity people have tried to introduce different kinds of solutions. One of them that is still related to natural language is the controlled natural language (CNL) [30]. This technique is built on minimizing the set of words that is allowed to be used in the specification of the requirements and also sometimes even a specific syntax has to be followed. With CNL the ambiguity is decreased a lot and also the way others can interpret the requirements, since the set of words that are allowed to be used is well specified and have only one meaning. However, CNL is not removing all problems within the requirement engineering phase, but it limits the faults to occur.

Commonly, CNL’s are divided into two categories, human- and machine-oriented CNL. Human-oriented is mainly developed to improve the communication between humans in technical documentation and at the same time improve the readability and unambiguity. Whereas the machine-oriented approach tries to translate normal language to a language that the machine could understand to be able to improve translatability of documentations and to take out knowledge from text. Regardless being human- or machine-oriented, both the categories are easier to understand for a human than normal natural language [31].

There are plenty of CNLs available in the literature [31] and during the research phase we distinguished some interesting ones. For example, a paper written by Tobias Kuhn [30] tries to overcome problems related to task-based and phrase-based CNL approaches by introducing a diagram-based approach to specify the language. In particular, this paper describes the ontograph notations where different base objects can be created together with a corresponding graphical representation. Then base objects can be refined into sub objects. Moreover, it is possible to define relations between objects together with their graphical representation. When this small requirement definition language has been finalized it is possible to specify objects and relations which constitute requirement sentences. As seen in Picture 12, the legend on the right describes the objects and relations while on the left side the relations between the object are displayed. These ontographs have been used and compared against other ways of
describing the relations using text and the result was that the ontographs are in general easier to understand.

![Ontograph](image)

*Figure 12: Ontograph with the legend explaining the language to the right and a description of mini world to the left [30].*

Two more researchers Esser and Struss identified the need to provide a predefined language template in order to specify the requirements and succeed in test generation [14]. Their aim was to come up with an approach easily integrated inside a car industry’s environment and processes that would facilitate the testing process of a real or simulated car. In their opinion forcing the requirement personal in a strict natural language is more beneficial than trying to educate them in using a formal specification language. As a result they propose a Template Based Natural Language Specification, which is built up by natural language sentences. However, these sentences adhere to some rules that specify the requirements as facts containing a start condition, a consequence and an end condition. Taking those rules into account and combined with domain theory they come up with 15 sentence templates, which can be transformed into Formal Requirement Language (FRL). With FRL a valid test model can be created and by extending the language mutated fault models can be also generated before the test case creation. The different steps of the approach are presented in Figure 13.
Figure 13: Normal requirements and their representation in template form, then FRL notation and lastly model illustration [14].

In [29], Fatwanto Agung emphasizes the problems due to natural language in the specification and tries to reduce them. The author’s proposal is the use of a specific sentence syntax pattern able to be parsed by the Reed-Kellogg sentence diagramming system. The sentence schema is implying that a requirement consists of a subject, a verb, a target and a way, where way complements the action represented by the verb. Having defined a rule for individual requirements, the author was interested in the dynamic aspect of the system. Thus, a scenario format is also introduced, forcing the requirements to be defined as event-state-action, as depicted in Figure 14.

Figure 14: Scenario format of requirements as defined in [29].
Moreover, there are three main approaches of machine-CNLS. Attempto Controlled Language (ACL), which is a subset of the normal English with a special syntax. The words that are allowed in ACL [31] are following these rules:

- Active and passive verbs (and modal verbs).
- Strong negation (e.g. no, does not) and weak negation (e.g. it is not provable that).
- Subject and object relative clauses.
- Declarative, interrogative, imperative and conditional sentences.
- Various forms of anaphoric references to noun phrases (e.g. he, himself, the man).

The set of available words in ACL is not defined and it is up to the users to provide such a specification. Furthermore, ACL is supported by various tools and has a mathematical ground to check the written text. Another approach is the Processable English (PENG) and just like ACL it is built on the English vocabulary [31]. However, this approach has a more light-weight subset of English, which makes it fully traceable. PENG has its own editor that validates the text while the user is providing input and can also suggest upcoming words. Just like ACL, PENG can be processed and validated with the help of a computer to ensure that the syntax and used words are correct. The third and last approach is the Computer Processable Language (CPL) which was developed by Boeing [31]. Differently from ACL and PENG, CPL is not relying on an editor or interpretation rules. CPL only accepts three types of sentences:

- Ground facts.
- Questions.
- Rules.

All of them have specific ways that they can be written in. All these three approaches proposed a way to mitigate the ambiguity in exploiting natural language in requirement specification [31]. Furthermore, they are all somehow related, or can be translated, to a formal specification.

An interesting and promising approach depending on natural language processing is found in [34]. This paper is the first one describing necessary steps having in mind the automation procedure and not just suggesting a way to improve the requirement specification as a document. Since, the test case generation or the test plans (sequence of steps to describe a test case) in their case are related to functionality, the chance of producing redundant tests is high. Hence, their whole approach is taking advantage of requirement clustering techniques, which group the requirements and reduce the possibility of two or more identical test cases. The first step of their approach is the annotation of requirements in an attempt to define the properties required to cluster them. Their conclusion here is the use of linguistic analysis to transform the sentences into syntax trees and obtain semantic attributes like actor, process and object. During the second step the clustering is completed by calculating the relationship among requirements. A requirement is considered strongly related to another when all the semantic attributes are the same; when this occurs the possibility of redundancy is high. However, it is up to the tester to identify the fault. The last step of the approach is the definition of test plans in natural language, but having in mind the correct ordering of tests. Two techniques are facilitating the test plan preparation, one finding condition relations between requirements and one searching for relations between objects and subjects inside sentences.
3.3 Model-Based

The model-based approach is a growing research topic and a systematic review done in 2007 [44] contains model-based approaches available to link requirements with test cases. The approach is about to make an abstract version of the actual system and represent it using a model, usually a state-chart. The requirements are then specified in the model and based on test criteria test cases can be generated to cover the system in different manners, all paths in the model, all requirement paths etc.

Testing Object-oriented systems with the unified Modeling language (TOTEM) is a functional test method proposed to automate the generation of requirement tests and shown in Figure 15. As the name implies TOTEM is used under object oriented software development and it is realized after the end of the analysis stage, when the first artifacts are produced. The method takes advantage of various UML models like use case, activity, sequence and class diagrams. In the beginning use case diagrams are described both with graphical and text notation to summarize the functionality. Then, activity diagrams are created to express the dependencies and interactions between the use cases, thus valid test scenarios can be generated. The dependencies are defined by including input and output parameters to the use cases and by following modeling rules when the activity diagrams are created to separate synchronous from asynchronous functions. However, only use case sequences are produced on this step and the actual test cases are formulated from sequence diagrams. Sequence diagrams actually display the interactions inside each use case having in mind objects coming from the domain application class diagram. Test cases are finally generated as regular expressions when the sequence diagrams are translated [20].

![Figure 15: Flow of actions to generate Use Case sequences](image-url)
In [35] the authors realized the insufficiency of the conventional black-box testing techniques, i.e. equivalence partitioning, boundary value analysis, for use at system level testing. Their proposal is the supplement of requirements in natural language with a formal or semi-formal description language in order to allow the automatic black-box test generation for system level. Another issue that the whole approach tackles is the mapping of individual requirements with corresponding test cases, since in system level the testers are unaware of the adequately tested requirements. The description language of the paper is called Specification and Description Language (SDL) standardized by the International Telecommunication Union. SDL is used to express system states and the transitions leading to each one of them. Initially, the SDL models are built for each individual requirement (Figure 16) and a system level representation is obtained as the combination of SDL fragments into one model. However, the paper does not mention the rules that are applied to get such a system SDL model. Their next step is the automatic conversion to an Extended Finite State Machine (EFSM) that labels all the transitions with guards and requirement ids. When the EFSMs are traversed having in mind different coverage criteria like state, path, transition and constrained path, the test cases are generated. Another advantage of the approach is the use of selective testing in the early development phases, when the testers want to check specific requirements or combination of requirements.

So far only papers considering the whole development process and setting up a complete testing approach were described, but papers were also found defining test generation from just an individual model. The individual model in all the cases is a diagram created under the UML notation, i.e. activity diagram, state chart. Even though the individual models differs, the followed approach is the same, thus an intermediate test representation is created and then coverage criteria are applied. Rivepiboon and Kansomkeat found a way to generate test cases from UML statecharts, by transforming them into Testing Flow Graphs (TFG) [21]. TFG are a simpler representation of statecharts that are traversed having as a coverage criterion state and transitions. Since none of the papers reveals results gained from
coverage techniques or applicability in an industrial domain, the explanation of these additional methods goes beyond the scope of this report. The interested reader can refer to [58], [59] and [60].

Even though models can be used to generate test cases, they are limited when it comes to the execution. The problem arises as the test cases are expressed at a higher abstraction level than the implementation under test (IUT). Two research teams tried to overcome this limitation by proposing methods for adding more information in the models without affecting their complexity. Their results succeeded in mapping actual implementation modules and the test cases. One of the teams focused on creating testable requirements models (TRM) represented with ACL, a contract requirement language. On a later step by using a Microsoft’s Software Development Kit they were able to read managed code and bind model knowledge to implementation specific parts [49]. The second research team took a step back in the development process and proposed a different way of specifying the requirements. They were using User Requirements Notation, an abstract approach for modeling functional and nonfunctional requirements, with Use Case Maps and Goal-oriented Requirements Language respectively. Their second step was similar to the previous team, as they created TRM supported with ACL [50].

3.4 **Scenario Based**

One of the simplest approaches that can be followed by requirement engineers is the creation of scenarios as part of use cases. Use cases are a common and accepted method to document the intended functionality of the system when actors (users, systems etc.) interact with it. A French research team tried to apply a use case contract pattern in order to support the automation of requirement-based testing [15]. The contract pattern, example seen in Figure 17, includes information about use case parameters as well as logical expressions for defining pre and post conditions. The use case parameters are describing the actors or main concepts of the system that the specific use case would manipulate or need. Logical expressions or predicates are added to realize the order of the test creation and execution, but also to describe the properties of the system before and after the execution. Again for this approach in order to produce test sequences a model has to be created, which in this case is named Use Case Transition System (UCTS). UCTS resembles a graph, as shown in Figure 18, with vertices as system states and edges as use cases. By applying different coverage criteria like edge coverage and vertices coverage the authors generate test sequences. However, it is mentioned that in order to step forward to test cases each use case has to be related to a sequence diagram, so that code details would be available.
Another research group also tried to focus on the generation of test paths in the early analysis phase of the software development [16]. Their approach takes advantage of a diagram from the UML standard called Interaction Overview Diagram (IOD), which is a more descriptive model used to outline scenarios included in use cases and further explained in sequence diagrams. The IODs on their own do not contain enough information to illustrate the sequence of operations, thus one more file is needed that contains contracts written in Object Constraint Language (OCL). These contracts are similar as the contract pattern used by the previous research group, as it has use case parameters, pre and post conditions. Using both the model and the OCL information and by defining rules to overcome issues when two contracts have the same pre or post conditions, they are able to create statecharts. The name of the statecharts is Contract Transition Systems (Figure 19) that has as states the ones defined by the contracts and as transitions the operations defined by the sequence diagrams in IOD. In this case the test paths are generated by the traversal of the graph.
A different scenario-based proposal resulting in test path generation was found depending exclusively on models. Researchers from the Electronic Engineering Institute of Hefei in China created a tool named Automated Test Case Generation Tool that receives as input System Sequence Diagrams (SSD) and generates test cases. Unlike the normal sequence diagrams the SSD are describing the flow of events from use cases without referring to internal system objects, but using a whole system instance that the actor interacts with. The authors added more value by distinguishing and categorizing the information of a scenario in eight pieces.

1. Actors using the system.
2. Outputs of the system to users.
3. External systems interacting with the system.
4. Messages sent from the system to external systems.
5. System validations of incoming requests and data.
6. Change of the internal system state by incoming requests or process of data.
7. Step preconditions.
8. Time outs.

Considering that a SSD is still a sum of informal information, a transformation has to be done in a more formal specification. Therefore, a state-based model called Interaction Finite Automaton (IFA) is used in order to aid the automatic test path generation with different test criteria. IFAs (Figure 20) are created by interpreting the different pieces of information in a scenario, as described previously, and mapping to a specific element node for each state or edge for each transition, respectively. As final step by using...
graph traversal algorithms and coverage criteria the test paths, presented in Figure 21, are generated. Still in this case the real test cases could be constructed after the design semantics are supplied [17].

Johannes Ryser and Martin Glinz proposed a method applicable in most of the development lifecycle phases, starting from the requirements elicitation. Their method is called Scenario-Based Validation and Test of Software or shortly SCENT and it adopts the scenario-based description of interactions. The created scenarios are improved during the whole process as a better understanding of the requirements is achieved. When the scenarios are initially created, it is the responsibility of the developer to translate them in a formal representation with state charts, including both the normal, exceptional and alternative flows. An additional model is created during the method called the dependency chart, which defines the relations between the scenarios. The final step of SCENT contains the automatic generation of test cases by choosing a coverage criteria and traversing the paths of the statecharts. The applied coverage criterion resembles branch coverage from structural testing and the aim is to cover all the links of the graph [19].
UCEd is a tool created to prove Stephane Some’s idea of adopting use cases with their text representation and validate them with scenarios (Figure 22) generated from use case simulations. The initial text notation of a use case is changed to include domain elements originating from a domain model. The updated use case is transformed to a state chart model, which is traversed to generate use case simulations and scenarios [22].

![Scenario sample taken from [22].](image)

One more tool was found during the research phase, named STATEST, which focuses on the automatic generation of test cases from informal requirements [23]. The tool accepts scenario descriptions written as part of a use case with normal flow of steps and one or more alternative or erroneous flows. The steps of the scenarios are transformed into State Transition Diagrams that on a later phase are traversed with different coverage criteria and produce the test cases. All the scenarios are part of use case descriptions and all the pre and post-conditions are stored, but the authors do not mention how they benefit from them. Since, the tool records the scenarios and the test cases, it also supports features for keeping track of the requirement coverage.

### 3.5 Formal Specification Language

Few decades ago engineers and researchers identified the need to check the product’s intended behavior against its implemented behavior. Their purpose was to improve the software quality without depending on implementation-based testing, which cannot verify if the functionality matches the user needs. Thus, the next identified step was a formal specification of the functional system behavior to enable the use of implementation-based techniques to generate test cases based on the actual written code with formal specification languages. Since, these techniques are related to a higher level of abstraction than the actual code they are known as specification-based testing [33]. The benefit of specification testing is twofold, as both cases for intended and unintended system behavior can be revealed. Since the aim of this type of testing is to reveal defects of implemented behavior there should be a relation between the formal language and the code. To be counted as formal specification a requirement specification approach shall be built up on the following main constituents [32]:

- Rules for determining the grammatical well-formedness of sentences (the syntax)
- Rules for interpreting sentences in a precise, meaningful way within the considered domain (the semantics).
- Rules for inferring useful information from the specification (the proof theory).

Furthermore, formal methods are often connected with a tool to aid the user in specifying requirements and their constraints in a correct way. Moreover, there are tools that can generate test cases based on the formal specification. Some of these tools are Tveda, TGV and TorX and all of them are based on the same algorithm [43].

Formal specifications are used to specify the software in a way that describes what the code will achieve without putting any constraints on the way it is developed. There are plenty of formal methods available in the state of the art and some of them are B, Z, VDM and ASM [41]. As mentioned in the previous paragraph, formal languages are mainly used to specify the behavior of the implementation and not the system and therefore the test cases generated from them target the correct behavior of the code. In [41], a survey is done to investigate the usage of formal methods in industry. This work reveals that formal methods are mainly used for maintenance to verify that the code has the expected behavior. Nonetheless, one important aspect coming out from that investigation is that the time spent in the beginning to build the requirement specification throughout a formal approach often reflects in a reduction of product development time.

Bryant and Lee [42] are using a formal method to generate object oriented Java code and UML models from the requirements. This is done by first parsing natural language requirements with the usage of contextual natural language processing into a Two-Level Grammar (TLG). Later, the TLG is mapped to VDM++, an extension of VDM that in turn is translated into UML models and Java code using the IFAD VDM toolbox. Since their methods is generating UML models, test paths can be derived using a model-driven approach. The full process is shown in Figure 23.

![Figure 23: Steps used to automate test cases from requirements in [42].](image-url)
Cabral and Sampaio [51] provide a way of going from requirements to formal specification using a tool to force the requirement engineers to specify the requirements using a CNL to create use cases. After that, with the help of another self-developed tool, a CSP specification is built and based on that specification a model can be derived to describe the system behavior. Just like in [42], a model-driven approach can then be used to generate test cases based on the model that was created. As shown in Figure 24, both the requirement and architecture can be used to create the CSP models and in the end be combined for further improvements.

![Figure 24: Steps used to automate test cases from requirements in [51].](image)

3.6 **MODEL-DRIVEN ENGINEERING**

Model-driven engineering (MDE) is a recent development approach based on the creation and management of models representing well-defined abstractions of the system under development [18]. MDE as a method simply sets an abstract concept whose representation can be chosen by the development teams from a broad range of modeling languages, like UML and SysML. Having the opportunity to use different types of models and conclude in more of them after the use of transformation rules, the traceability is lost through the development process. Hence, Object Management Group outlined the different types of models and standardized the Model-driven Architecture (MDA) [48]. MDA specifies four level of models that encapsulate different aspects and details of the product. The first is Computer Independent Model (CIM), a category that catches the company’s business logic and requirements level. The next level is the Platform Independent Model (PIM) capturing the software system without taking into account development platform constraints (.NET or Java), which is achieved by the Platform Specific Model. The final level is the actual implementation of the product, that is, code.

MDA proceeds from one development phase to another by transforming models starting from CIM towards the production of code. The transformation can be done also inside the same level, as going from a PIM to another PIM model. Model transformation is also a quick way to identify that the used models are consistent and follow a syntax. Valid models can be constructed when a set of rules is applied that defines the available modeling elements, their properties or attributes and their relation. The set of rules is called a metamodel and is commonly illustrated as a class diagram. The University of Seville proposed a language for the definition of functional requirements. Such a language enables the
generation of tests by considering CIM and PIM. In particular, they first created use case diagrams described with tables following the metamodel in Figure 25. Later on to add additional value and prepare test paths they transformed the table information to activity diagrams. Furthermore, in order to catch exception paths the tables are transformed to operational models defining operational variables. Lastly, from those two available models test cases could be generated [18].

![Figure 25: Requirement model [18].](image)

### 3.7 CONCLUSIONS

The previous sections gave an overview of the current approaches that the research community offers with respect to automatic solutions connecting requirements with testing. The approaches we found are categorized having as a criterion the format of the requested input. Hence, methods expecting as input individual requirements written in natural language are presented in section 3.2. In section 3.3 all the methods that require models are summarized. Following up, section 3.4 contains approaches where requirements are gathered as scenarios and section 3.5 describes those works where a formal specification is demanded. Whereas section 3.6 describes the usage of techniques that exploit model driven engineering for test generation.

Table 1 shows a summary of the approaches available in the state-of-the-art. The number of the analyzed approaches is seventeen and data of each of them is presented in each row of the table. The contents of the table can be divided into two sections, comparison attributes and paper information. Paper information simply contains the citation number to be referred to in the bibliography. Attributes are five different criteria that are chosen in order to summarize the related work and build a common base to draw conclusions. The chosen attributes are:

- **Format of input**: Standing for the type of the data that each proposed approach requires.
- **Notation**: The type of the intermediate stage that can lead to results.
- **Tool**: Evidence that the results are enhanced with proof.
- **Automation degree**: the proportion of automated steps in comparison to all the tasks required to apply an approach.
- **Aim**: Definition of the goals targeted in the application of an approach.

In particular, thirteen approaches aim and result in test generation, two aim at test execution, one at improving the quality of the requirement specification and one at transforming the requirements into a model. Out of the thirteen approaches that lead to test generation ten are using as notation an intermediate model that is traversed and facilitates the test case generation. What is also interesting is the amount of tools developed in order to allow test generation; as a matter of fact, only 30% of the methods are describing the existence of a tool. Moreover, the availability of those tools and their applicability in industry is unknown. Coming to the format of the input two are the most common ones, functional requirements written in specific format (template, normal human language) or supported with other means (models) and use cases with 41% and 35%, respectively. Moreover, when it comes to the level of automatization of the approaches that lead to test generation, it can be seen that most of them achieve 50 to 75% of automation. There are also two cases where full automatization is realized, but one of them demands as input just a model, whose creation is manual. The second fully automated approach contains an initial manual step that prepares the format of the expected input.

*Table 1: Summary of the found research approaches and mapping of their content to valuable comparison attributes.*

<table>
<thead>
<tr>
<th>Paper Ref.</th>
<th>Format of input</th>
<th>Notation</th>
<th>Tool</th>
<th>Automated steps/Total Steps</th>
<th>Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Functional requirement in template form</td>
<td>Formal requirement language</td>
<td>Unknown</td>
<td>Test generation</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Use case scenarios with contracts</td>
<td>UCTS</td>
<td>No</td>
<td>2/3</td>
<td>Test generation</td>
</tr>
<tr>
<td>16</td>
<td>IOD and OCL contracts</td>
<td>CTS</td>
<td>Yes</td>
<td>2/4</td>
<td>Test generation</td>
</tr>
<tr>
<td>17</td>
<td>Use Cases and SSD</td>
<td>IFA</td>
<td>Yes</td>
<td>2/4</td>
<td>Test generation</td>
</tr>
<tr>
<td>18</td>
<td>Use case diagrams</td>
<td>Activity diagrams and operational models</td>
<td>Unknown</td>
<td>1/3</td>
<td>Test generation</td>
</tr>
<tr>
<td>19</td>
<td>State charts and Dependency charts</td>
<td>State charts</td>
<td>No</td>
<td>Unknown</td>
<td>Test generation</td>
</tr>
<tr>
<td>20</td>
<td>Use cases supplied with UML diagrams</td>
<td>Directed graphs and Sequence diagrams</td>
<td>No</td>
<td>2/3</td>
<td>Test generation</td>
</tr>
<tr>
<td>21</td>
<td>UML state charts</td>
<td>TFG</td>
<td>Unknown</td>
<td>2/2</td>
<td>Test generation</td>
</tr>
<tr>
<td>22</td>
<td>Use Cases and a Domain model</td>
<td>State model</td>
<td>Yes</td>
<td>1/4</td>
<td>Test generation</td>
</tr>
<tr>
<td>23</td>
<td>Use Cases</td>
<td>STD</td>
<td>Yes</td>
<td>2/3</td>
<td>Test generation</td>
</tr>
<tr>
<td>29</td>
<td>Functional requirements with specific syntax</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Requirement Specification</td>
</tr>
<tr>
<td>30</td>
<td>Functional requirement</td>
<td>Ontographs</td>
<td>None</td>
<td>1/1</td>
<td>Model</td>
</tr>
<tr>
<td></td>
<td>with specific syntax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>34</strong></td>
<td>Functional requirements following patterns</td>
<td>Natural language</td>
<td>Yes</td>
<td>2/3</td>
<td>Test generation</td>
</tr>
<tr>
<td><strong>35</strong></td>
<td>Functional requirements and SDL representation</td>
<td>SDL system model</td>
<td>None</td>
<td>1/2</td>
<td>Test generation</td>
</tr>
<tr>
<td><strong>49</strong></td>
<td>TRM models</td>
<td>None</td>
<td>Yes</td>
<td>None</td>
<td>Test execution</td>
</tr>
<tr>
<td><strong>50</strong></td>
<td>URN</td>
<td>None</td>
<td>Yes</td>
<td>None</td>
<td>Test execution</td>
</tr>
<tr>
<td><strong>51</strong></td>
<td>Functional requirement in template form</td>
<td>CNL/FRL</td>
<td>Yes</td>
<td>3/3</td>
<td>Test generation</td>
</tr>
</tbody>
</table>
4 STATE OF THE PRACTICE

The existence of a gap between state-of-the-art and state-of-the-practice in requirements engineering is not something new [24]. To get a good understanding about this gap this section will introduce the state-of-the-practice in Bombardier Transportation together with the processes and tools they are using to keep the desired level of quality of delivered products.

4.1 BOMBARDIER TRANSPORTATION

The department inside of Bombardier Transportation where this thesis was carried out is responsible for the development of the train control and management system (TCMS). Internally the lean development method is used to ensure and improve the quality of the product. Since the TCMS is onboard on every train and is a safety critical system it forces the employees to follow specific guidelines for the whole process.

The general life-cycle model used is the V-model. Bombardier has been using it for several years and has found a mature way to handle the development. However, in a recent project the lean development method has been introduced to speed up the project work and also to ensure the quality of the final product. As seen in Figure 26, the first two boxes, Vehicle Requirements and Vehicle Design, contain tasks that are accomplished before the TCMS department starts its work. The triangles are milestones that shall be fulfilled before the next phase can be started and the diamonds are checks done by the authority to ensure that Bombardier is following the standard guidelines and that the quality is good enough to continue the work.

![Figure 26: Overview of Bombardier Transportation life-cycle model.](image-url)
4.1.1 Train Control and Management System

TCMS is an infrastructure backbone which allows easy integration of the on-board train functions such as controls and communications. It is a real-time system which has a high bandwidth and processing capability for transferring data, both inside the train and with the outside world. TCMS’ s architecture is mainly built upon three networks, Internet Protocol Train Communication (IPTCom), Multifunction Vehicle Bus (MVB) and serial, and with the help of them interacts with all the train subsystems like brakes, doors, pantographs and so on. As shown in Figure 27, the TCMS system location is spread in the train depending on how many cars the train contains. Moreover, the first and last car, usually named cabs, include in most cases the drivers panel, where the driver of the train is located, and the ones in the middle can often be seen as passengers cars. The functionality of the TCMS is then divided by location. For example, the raise pantograph and emergency brake, which are seen as normal actions, are located in the middle cars, while the safety actions are located in the cabs. Moreover, since a train is a vehicle that is transporting humans it is seen as a safety critical system. To mitigate the hazards the TCMS system uses redundancy (two of each unit). To distinguish between safety and non-safety units the colors blue and yellow are used on front of the actual unit, blue for non-safety and yellow for safety.

Bombardier has in the latest times developed a soft version of the TCMS. It is called soft in the sense that no hardware is needed and instead the whole TCMS is simulated using a normal PC. This opens a new world for Bombardier since they will not have to wait for the hardware to be developed, or for the testing environment to be available thus remarkably decreasing testing costs. This also enables each tester and developer to have the whole system on their desks and by that can start the testing much earlier. Then when the confidence is built in the soft TCMS a test can be carried out in the real test
environment. This leads to reduction of the time spent and queues built in the real testing environment. Figure 28 resembles one of the cabins containing the TCMS system.

Figure 28: The actual hardware of a CCU-O.

4.1.2 Requirements Engineering
Bombardier has a mature requirement engineering phase with well-defined steps. The requirements are specified by the customer, standards, authorities and also in house requirements are created. These requirements are then divided by sub system and handled by all system engineers developing the train. From these specifications it is derived which requirements belongs to TCMS and subsequently gathered in a Technical Requirement Specification (TRS). From this document a refinement of the requirements is made and they are further divided based on which subsystem the requirement is a part of. Since Bombardier is developing software for trains, specific safety requirements have to be written and this is also done in between the TRS and Software Requirement Specification (SRS). These safety requirements are gathered in their own documents and contains a more detailed explanation of the behavior of the system.

For “normal” requirements, i.e. non-safety related ones, Bombardier adopts a fully functional decomposition of the system, and as mentioned before requirements are grouped by the sub-system
they pertain to, and not by the functionality they serve. The requirements are written in the tool IBM Rational DOORS and Bombardier is using links to keep track of refinement relationship between requirements. However, there is no specific standard that has to be followed when the normal requirements are written. Thus, Bombardier is trying to follow a common guideline for specification of the requirements. Each requirement has an id, a description and a Safety Integrity Level (SIL) level, even though the requirements are seen as normal they are specified with SIL 0. In Table 2, some examples are given about how the requirements are refined from the TRS to the SRS.

Table 2: Refinement of TRS into SRS.

<table>
<thead>
<tr>
<th>PUID</th>
<th>TRS Requirement</th>
<th>SRS Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SRS-DRS-REQ-0046 (V.1)]</td>
<td>4S_09.03.05.Zefiro-Italy.TRS.2829 TCMS shall do an integrity check on the green loop. TCMS shall compare digital input All doors in train closed with status from all passenger doors and all cab doors on the train. If they differ an event shall be set. Supervision not valid on a bypassed door. Std Pass door the status is received from DCU via IP. Cab door the status is received via hw input.</td>
<td>If the green loop is energized (All doors in train closed) and any door (cab or passenger door) indicate NOT closed an event shall be set. Isolated (indicated by the DCU or isolated by the selective door opening function) or faulty (indicated by the DCU) doors shall not be considered. IP-input: IDoorState (1=Door closed and locked) IO-input: Cab door closed IO-input: “All doors in train closed”</td>
</tr>
<tr>
<td>[SRS-DBC-REQ-0618 (V.1)]</td>
<td>4S_09.03.05.Zefiro-Italy.TRS.2911 &quot;Boost mode 2&quot; shall be set to all PCUs when 4 MCMs in the train are not available.</td>
<td>&quot;Boost mode 2&quot; shall be set to all PCUs when 4 MCMs in the train are not available and boost mode is enabled. MCM is considered not available if signal &quot;MCMx is OK&quot; is not set for 15 seconds. MVB-input: &quot;Status: MCM1 is OK&quot; MVB-input: &quot;Status: MCM2 is OK&quot; MVB-output: &quot;Command: Boost mode level 2&quot;</td>
</tr>
</tbody>
</table>

When it comes to safety requirements, BT cannot enjoy the same level of maturity since safety standards have been introduced only recently in the railway domain. Therefore, they have been adopting specification techniques that require some stabilization and solid acceptance inside the company. Nonetheless, since the trains that are developed have to meet SIL 2 safety standards requirements Bombardier has to follow specific procedures to achieve correct specifications. Bombardier is mainly following the standard EN50128 and is therefore encouraged to use different types of diagrams, such as sequence and use-case, to specify the system behavior and also write them using scenarios. Bombardier is following this standard in a good manner in the safety SRSs. Furthermore, differently from normal requirements safety requirements are also containing non-functional requirements.
To be able to move towards an automated approach to generate test cases from requirements a deeper knowledge of the current state of the requirements is needed. During the thesis most of the requirements in a project were read and a set of common mistakes when specifying requirements were found. These mistakes are important to reveal and mitigate in order to be able to move towards the desired approach. To provide a deeper knowledge in how requirements are specified in a bad manner a set of examples will be provided. Table 3 shows how a requirement that has been specified is written in a very ambiguous manner. This requirement can be interpreted in many different ways and in the end it is unclear what is wanted from the system. In particular, it is not clear what the conditions are and that two different events have to be created, one for the trailer cars and the other for motor cars.

Table 3: Ambiguous Requirement.

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If above speed v &gt; 35 km/h on motor and v &gt; 55 km/h on trailer the 4 stage relay RM/RP supervision input is set (1) TCMS shall set an event undue fourth stage.</td>
</tr>
</tbody>
</table>

There are also several requirements that are using different words for describing the same condition. In Table 4 it is shown an if-condition used in different manners. E.g. ID1 is using when a and quotation marks around serious fault, while ID3 does not use quotations. Moreover, ID2 is using the word if to describe the same kind of event. Furthermore, in ID4 the condition is presented after the actual action, while ID1-3 starts with the condition. Finally, in ID4 context is not mentioned any serious fault, adding further confusion to the reader of the requirements.

Table 4: Usage of the different keywords for the same condition.

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>When a &quot;serious fault&quot; occurs the driver can acknowledge it via hardkey on IDU. If no ack is detected within 15 seconds Traction block shall be set until the acknowledge is done</td>
</tr>
<tr>
<td>2</td>
<td>If a second serious fault occurs when the lamp is already switched on, the buzzed shall start to sound and the new fault shall be acknowledged by the driver.</td>
</tr>
<tr>
<td>3</td>
<td>When a serious fault is active and a new cab is activated, the lamp shall be switched on in the new cab and the fault needs to be again acknowledged by the driver.</td>
</tr>
<tr>
<td>4</td>
<td>The lamp shall extinguish if conditions for rectification are fulfilled, i.e. the fault is bypassed, or fault condition is inactive.</td>
</tr>
</tbody>
</table>

The third example given is the way of describing the same function, or state, using different keywords. In Table 5, the requirements are specifying different lamp states. As shown, depending on the requirement different keywords are used, e.g. ID1 is using the word lit to describe that the lamp is on, ID2 is using extinguish to specify that the lamp is off, ID 3 and 4 are using switched on and switched off to describe these functionalities and finally, ID 5 is using the keyword illuminated to describe the lamp state.
Table 5: Usage of the different keywords for the same state.

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>When new &quot;serious fault&quot; is present the lamp on the driver's desk shall be <em>lit</em> and the buzzer in the cab shall sound. The buzzer shall be muted when the ack is pressed.</td>
</tr>
<tr>
<td>2</td>
<td>The lamp shall <em>extinguish</em> if conditions for rectification are fulfilled, i.e. the fault is bypassed, or fault condition is inactive.</td>
</tr>
<tr>
<td>3</td>
<td>If a second serious fault occurs when the lamp is already <em>switched on</em>, the buzzed shall start to sound and the new fault shall be acknowledged by the driver.</td>
</tr>
<tr>
<td>4</td>
<td>For C3 class faults the lamp shall be <em>switched off</em> soon after the driver acknowledges the alarm.</td>
</tr>
</tbody>
</table>
| 5  | The lamp of the parking brake button shall have following functionality:  
  - *illuminated* if parking brake is applied on all cars  
  - *not illuminated* if parking brake is released on all cars  
  - lamp shall *flash* if parking brake is not released in all not isolated and manual released cars  
  OR minimum applied parking brake amount not reached. |

The last example, shown in Table 6, shows two overlapping requirements, and in particular they are such that one is contained in the other. As shown, the serious fault lamp is specified to be a part of the lamp test, which is a function that is used to make sure that all lamps are working before running the train.

Table 6: Overlapping requirements.

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The serious fault lamp shall be part of the lamp test.</td>
</tr>
</tbody>
</table>
| 2  | The following lamp shall be illuminated:  
  - parking brake lamp  
  - PEB bypass lamp  
  - serious fault lamp |

4.1.3 Testing

When it comes to testing Bombardier is starting early, already in the requirement phase of the development process test specifications are prepared for sub-systems. These test specifications are derived from the requirements and each requirement has at least one test case bound to it. This enables bombardier to have a trust that what is written in the requirement is also what is implemented in the end. Test specifications are provided by means of DOORS, just like the requirements, in order to exploit built-in features of the tool for keeping easily linked requirements and corresponding tests. The specifications are sometimes written in a very generic way and it can be hard for someone without system knowledge to understand them. One reason for this is due to the gap between requirements and test specification, which is often too big for enabling the testers to specify exactly what is going to be tested in the end. In any case the tests defined in the specification are reviewed by another tester to ensure the quality and validity.
When it comes to techniques for the testing Bombardier is using different types depending on the level of testing. For example, they are using white-box when they are testing the code and black-box when they are testing subsystems or the system as a whole. However, the test specification written in the requirement phase is mainly aiming at system or integration testing, since they can be derived from the requirements. Thus, specifications are mixing both black-box and white-box testing within them. Moreover, not all test specifications are written during the requirement phase. For instance, the commissioning test specification is written when most of the requirements are stable and most of the development is done. This document is a test specification that contains tests derived from other test specifications aiming at proving that the train has the most important functionality implemented and that it is working as desired. An example of one of the test cases specified in the commissioning test specification are given in Figure 29.
Figure 29: Example of a test cases written in the regression test specification.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Activate cab and log in as driver. Close all doors and make train ready to drive. Select forward driving direction. Set master controller in coast position. Cut out all MCMs in 3 motor cars so that 25% traction remains. Activate washing speed by pressing the washing mode pushbutton on the IDU.</td>
<td>Check that washing mode is indicated as active on IDU.</td>
</tr>
<tr>
<td>2</td>
<td>Set master controller in traction position.</td>
<td>Check that speed is well regulated to target speed 2 km/h.</td>
</tr>
<tr>
<td>3</td>
<td>Increase speed by setting speed lever in “+” position.</td>
<td>Check that target speed is increased by 1 km/h. Check that speed is well regulated to the new target speed.</td>
</tr>
<tr>
<td>4</td>
<td>Increase speed by setting speed lever in “+” position several times.</td>
<td>Check that target speed is increased to maximum 6 km/h. Check that speed is well regulated to the new target speed.</td>
</tr>
<tr>
<td>5</td>
<td>Decrease speed by setting speed lever in “-” position.</td>
<td>Check that target speed is decreased by 1 km/h. Check that speed is well regulated to the new target speed.</td>
</tr>
<tr>
<td>6</td>
<td>Decrease speed by setting speed lever in “-” position several times.</td>
<td>Check that target speed is decreased to minimum 2 km/h. Check that speed is well regulated to the new target speed.</td>
</tr>
<tr>
<td>7</td>
<td>Set master controller in brake position.</td>
<td>Check that train is braked to standstill.</td>
</tr>
<tr>
<td>8</td>
<td>Cut out one more MCM so that less than 25% traction remains.</td>
<td>Check that washing mode is deactivated.</td>
</tr>
<tr>
<td>9</td>
<td>Remove cut out of all MCMs. Select the event history menu on the IDU.</td>
<td>Check that no incorrect events have been logged during the time the test steps in this test case were performed.</td>
</tr>
</tbody>
</table>

Figure 29: Example of a test cases written in the regression test specification.
4.1.4 Test Environment

The department in which this work has been accomplished is using a set of tools to increase the quality of the final product and these tools are both simulations of the train and frameworks to enable automatic testing. A deeper description of these tools will be given within this section.

4.1.4.1 Vehicle Control Simulator

Since Bombardier is developing trains and wants to have confidence in the developed software before it is actually tested on a real train, a simulation of the train is developed. This simulation is named Vehicle Control Simulator (VCS) and is a sub-system of the test environment that Bombardier has created and uses, in order to control the system under test or test object. As test object the TCMS hardware shown in Figure 28 is used. During the testing phase, the application software that shall be tested, is running inside TCMS’s units. In addition to the test object the VCS test environment contains the test station and the VCS application. The test station is the rack of computer components shown in Figure 31 that powers the test object and enables the communication through buses that are presented in Figure 30. Moreover, it is the part that allows the VCS application to read outputs from TCMS and simulate inputs, while acting as the real individual subsystems like brakes, battery, etc.

Figure 30: Architecture of VCS and communication with Test Object.

Figure 31: Hardware components building up Test Station.
The VCS was built considering reusability between the different projects, thus a separate database is created per project. This database contains, in the form of signals, all the necessary information to simulate the communication to all the subsystems of the TCMS, like IO components, driver panels and physical and electrical modules. The VCS simulator is built on top of the VCS database to graphically present the behavior of the subsystems from lower (MVB busses) to higher abstraction level (simulations), whenever an action is performed on the test object. Simulations are one of the important concepts that guide the development and preparation of the VCS to support the testing process. Their purpose is to gather signals and combine them to create functions of actions. In Figure 32 one graphical layout of the VCS is presented.

![Desktop view of the VCS.](image)

**Figure 32: Desktop view of the VCS.**

### 4.1.4.2 Test Automation Framework

The testing process inside Bombardier is facilitated with the use of the Test Automation Framework (TAF). As specified in section 2.3.3 an automation framework contains functionalities allowing test case generation, execution and validation. Currently, TAF supports test case creation, test case execution and test case validation including the analysis of results and their extraction in reports. The supported functionalities as a whole result in a complex system, hence its design could lead to a complex architecture. However, Bombardier has succeeded in building separate components that have been assigned to different responsibilities. The three components are called Test Script Editor (TSE), Test Engine (TE) and Test Execution Manager (TEM).

Beginning with the normal flow of testing, TSE should be described first as it is the component used for the specification of tests. The created tests are actually instructions specifying what should be done during the test execution. Since VCS is the application that simulates the inputs of the test object referred as TCMS, those inputs should be available inside TSE. The editor needs to collect from the VCS database input stimuli in the form of signals, controls, indicators and simulations. Once the tester retried these inputs it is possible to create test cases related to the SUT.
Two more subjects are missing from the previous description, the structure of the tests and the outputs of the application. TSE can save and manage two types of files, called test scripts and snippets. A test script is the main output of the application and it is considered as a transformation of a test case taken from the test specification into logic for the automated testing process. The structure of a test script resembles the one of the test case shown in Figure 33, so it is built out of steps and each of them contains substeps with actions, reactions and neutrals. The second output is called snippet and allows reuse of a set of operations as it can be used within several test scripts. However, a snippet cannot contain another snippet since the complexity would increase too much. Moreover, the internal structure of the snippet differs from a test script’s, as the former is not consisting of steps. The graphical representation of a snippet can be seen in Figure 33 as the third rectangle. Both outputs are saved in XML files down to the Visual SourceSafe (VSS) database that allows version handling.

![Figure 33: GUI of TSE.](image)

The second component in the automation testing process and the most important one is the TE. This component is responsible for the test script execution, validation and documentation of results. As Figure 34 shows, TE, as TSE, is an application consisting of a graphical user interface. The application initially prepares the SUT and then executes the test scripts. As the test scripts are executing the validation rules, the application takes action and works as an oracle as the expected output is compared with the actual one to determine if a test script has failed or passed. During the validation one more component acting as a controller like VCS is used that is named DCU. This component, whose inner structure is out of the scope for this report, allows the communication with the TCMS and lets the observation of the signal status during validation. The logged signal status lets the documentation of the results by generating reports as pdf or XML files.
The third component known as TEM was developed to add more value in the test execution and fill in a gap introduced by the TE. TE is limited to execute one test script at a time, thus it is time consuming for a tester to just wait for the completion of one execution to start another one. The whole meaning of automation is lost as test scenarios cannot be produced. Hence, TEM allows the creation of test flows, giving also the ability to manipulate their execution, as the tester sets implicitly the sequence of the test scripts. This sequence, named session, is decided during the test execution as it relates to the output validation as passed or failed. Lastly, the application supports the set of iterations for the execution and also conditions for termination. Its GUI it shown in Figure 35.
Figure 35: GUI of TEM.
5 Concept

Defining a mature concept out of the proposed research methods, which should be applicable for general usage in industry, is challenging. One of the main reasons is that the optimal approach from a research point-of-view could rely on a number of preconditions difficult to be met in practical industrial contexts. Notably, there usually exist business level constraints hindering a completely free applicability of available technologies. Examples of those constraints can be categorized as countable and uncountable variables. As countable, metrics like training period of the personnel in order to be notified for a new working approach, cost in realizing tools that can facilitate the approach, are considered. Except the variables translated in numbers, there is also a bundle of more that cannot be measured like the confidence that a newer approach is more beneficial. It should not be forgotten that an approach can be established only if the higher ups in the organizational hierarchy accept it [8].

Another challenging factor that affects the definition of the concept is the ability to judge all the related work, presented in section 3, under a common criterion. In the problem area that was defined for the thesis purpose, it can be observed that most of the efforts and most of the results are gained through model-based testing. An analysis of the state of the art reveals that to be able to automate the testing from requirements an abstraction of the system has to be prepared [52]. This abstraction is in most cases, a model of the system related to its behavior and describing the manipulation of its states [55]. Depending on the chosen model, different test criteria can be set and corresponding test objectives can be generated [15-23]. Thus, the system can be tested when those test objectives can be mapped to actual test cases.

Even though an intermediate system model is created without exception for all the approaches we found, none of them is fully automating the process when having as an input an informal requirement specification. In most of the cases an analysis or an improved version of the specification is manually created [14, 27, 28, 29, 34, 36, 37]. Additional value is also added with the inclusion of models for simplifying the communication with the customer on the top and the designers/implementers on the bottom of the development process [56]. Hence, a lot of effort is spent for transforming the specification into more formal and straightforward instructions that will alleviate the deficiencies coming from natural language. Taking into consideration the discussion provided so far, it is very difficult (if not impossible) to propose a general solution applicable to an arbitrary industrial setting. At least, each industrial domain is characterized by a specific development process and maturity level.

Nonetheless, we can suggest an adoption process that would result in a progressive and smooth establishment of changes. The changes are partitioned into three steps and will lead in the end to the generation of test cases from requirements while at the same time improving the quality of the requirements themselves. As shown in Figure 36, the initial step is mainly about improving the way of writing the requirement by introducing a CNL, the second step manage the way of structuring the requirements by introducing scenarios that describes functions. The final step is mainly about how the scenario-based requirements shall be translated into a model and further be traversed into test cases. As a consequence, even when a company would not fully embrace the proposed approach, it would still benefit of enhanced requirement specifications. The steps shall be applied in a sequential manner, meaning that step one is the initial step, then step two is following and in the end step three shall be applied.
5.1 **INITIAL AUTOMATION STEP**

Natural language is by far the preferred specification method used to specify requirements. Hence, an improvement of the used language seems to be the most appropriate initial step. As mentioned in previous sections, badly written requirements have side effect to different phases of the development process. In this respect, a conversion from a normal language to a CNL is something that can be achieved without putting a lot of effort and its learning curve is steep. Considering that the human language is still used, the engineers will feel comfortable when they are documenting the requirements.

Taking into consideration CNL approaches found in the state of the art, we understand that the main goal is the creation of a limited vocabulary that restricts the usage of words. The resulting vocabulary is filled with words coming from the relevant industrial domain. Each chosen word should represent and should be mapped to either a specific interaction with the system or a change in the system’s state. The procedure for the vocabulary/dictionary creation requires the participation and cooperation of requirement engineers, who should select and refine the most fitting words that are allowed to be used. CNL’s limited vocabulary eliminates the usual phenomenon observed in the content of specification documents, where a lot of synonyms are used to describe the same action. There are two main reasons why several meanings for the same word are used. Firstly, because of the high chance that different requirement engineers are documenting the requirements, thus they use their own writing style and knowledge of the language. As a second reason, the erroneous interpretation of humans that semantic variety results in more interesting specification documents [8].

The definition of a vocabulary succeeds in eliminating the individual interpretation, but does not resolve the ambiguity of the requirements content. Hence, additional value is added in the CNL, by prescribing patterns that well-formed requirement sentences are expected to obey. The number of patterns as well as their actual structure is related to each specific industrial domain. However, regardless the target environment, patterns should exemplify the way of describing the different interactions with the system. Each pattern should represent its own category, like describing activities, actions, events,
system properties and states. This leads to simpler sentences that combined with the limited vocabulary results in more concise requirements [34, 53].

In the case where the industry is creating a simulation of the SUT it is a good practice to use the same keyword to represent the states and transactions in both the simulation and the requirements. In fact, the former could potentially work as a model of the system and be used to test the development against the requirements. So, if the mapping between an abstraction of the system and the requirements is done at an early stage of the adoption of the concept, the upcoming steps would demand less efforts.

Since requirements are usually product specific, it is impracticable to provide a vocabulary or pattern that can be applied as a general solution. Therefore, this will not be a part of the concept that is provided and it is the company's task to specify a proper CNL. However, certain ideas can be provided concerning the structure of the sentences and the categorization of their content. Following the pattern proposed by [34] and taking into account information from [54] we can suggest as a general sentence structure the one shown in the figure below. The important aspects of the sentences are the actor, the process and the object, in order to identify who (actor) is interacting with what (object) and in what way (process). Later on, the process, which is described with the use of a verb, is categorized depending on the meaning it bestows on the sentence. Thus, verbs are classified into Actions that manipulate the behavior of the system and can consist of both domain specific and general verbs. Action verbs can be also accepted in their past tense, when they refer to a completion of an activity. Furthermore, we have verbs belonging to the State category, as they express a static relationship between the system and its properties. Finally, the remaining ones belong to the category called Rule and describe general details of the system that do not belong to States. As a general rule for the sentence grammatical analysis, we can recommend the use of active form, in order to clearly define the performer and receiver of an action.

Figure 37: Pattern template for requirement specification [34].

<table>
<thead>
<tr>
<th>Action (AC)</th>
<th>State (ST)</th>
<th>Rule (RL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>have</td>
<td>support</td>
</tr>
<tr>
<td>personalize</td>
<td>be</td>
<td>conform to</td>
</tr>
<tr>
<td>deliver</td>
<td>comprise</td>
<td>allow</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
5.2 Second Automation Step

By introducing a CNL the specification can be manipulated conveniently and the evolution to different novel test generation approaches, found during research, is smoother [14, 15, 29, 30, 34, 51]. From this step and on, different branches can be followed to further improve the automation process and produce more concise test scripts. One of the approaches is the introduction of use case scenarios, which as steps can contain the CNL description, but containing additional information about the flow of actions (pre and post conditions). This way all the alternative flows can be covered, meaning that the individual interpretation of the tester is eliminated. Moreover, the relation of the use cases with test generation is also based to model testing, thus a model representing the flows of actions is needed. Even though this is an additional step, the benefits are relevant, since negative test scenarios can be also generated [16].

Another approach that can be introduced after CNL is the usage of a formal specification. This specification can be a mathematical expression easily mappable towards models or test cases. There are several approaches described in section 3 which use different well-defined languages and tools to enable automated testing. However, formal specification might be hard to exploit for the assessments done by authorities, since it requires appropriate technical knowledge, and it is often highly coupled with the code itself. Nonetheless, by using formal specification different testing criteria can be chosen, since it is expressed in a logical way.

Out of the two identified approaches, scenario-based and formal specification, the most ideal appears to be the scenario proposal. This approach supports the simulation of the whole system’s behavior through the creation of models that describe the expected sequence of actions. Moreover, when the requirements are divided by functions suspects of their incorrect relation or even their duplication can be cleared, as they are combined inside the body of the scenario. Formal specifications on the other hand, even though they are considered as means to explicitly describe the behavior of functions, are more applicable at a lower level.

As mentioned in the previous step, if an abstraction of the system is developed it is valuable to combine the scenarios and the abstraction. Thus, the simulations shall act as the real product developed and if the mapping is taken into consideration as early as possible, a smoother upcoming approach can be applied.

The use of scenarios solves the problem arising from individual requirements that do not contain any information related to the sequence of the execution. If we try to generate test scripts directly from the refined requirements resulting from the first automation step, we will earn a one test script per requirement. These generated test scripts do not have any value since the tester will be unaware of the proper flow that will influence the test procedure and produce valid results. Therefore, by introducing pre and post conditions around each scenario, a concise system model describing the interactions can be build. The pre- and post-conditions can be structured as logical expressions on predicates as described in [15]. Each predicate can be evaluated to either true or false value depending on the contained expressions. Therefore the value of the predicate can serve as a guard. In this way the execution of the scenario and its internal expressions correspond to states of the system before and after the execution for pre- and post-conditions, respectively. In the case where an abstraction of the system is already built, like a simulated environment, its states can act as predicates.
Having the flow of the scenarios set with pre- and post-conditions, the only missing information is the method resulting in these scenarios. Their creation is linked with the previous automation step and the definition of sentence patterns. If the engineers determine the keywords that add time constraints on individual requirements, then a scenario can be created implicitly. The time constraints can be added as conditions in the proposed sentence patterns of the previous section. It is worth noting that the first automation step is done on purpose in order to both benefit the quality of specification and effect on the automation process.

### 5.3 Final Automation Step

The focus in the last step is to move from a scenario-based CNL to a model that later can be traversed into test cases. There are several approaches that are traversing a model and create test cases, but there is a limited set of approaches that goes from scenarios to models. As described in section 3, some approaches exist. However, what they all have in common is the need of a pre- and post-condition to be able to create a model with states and transactions. When this model is parsed by following a specific path traversal criterion, like all transitions, then test objectives or test sequences at function level are generated. These objectives are the flows the tester should follow in order to trigger the system and collect results.

Now that the system overview is created, the remaining step is to map each scenario’s body with a generated test script. Since the defined CNL language is used for describing the steps and flows inside the scenario, it is possible to have a test case generation. As proof, one specific work is done by [34] where they specify the requirements in a specific format (part of it is proposed in the first automation step) and then generate a model out of them. In their final step they create test cases based on the model. This approach is promising and by following the two previous steps of the concept we believe that the requirement would be in a good state to allow test case generation.
6 CASE STUDY - BOMBARDIER TRANSPORTATION AB

This section discusses the application of the first step of the concept specified in the previous chapter into an industrial setting, namely Bombardier Transportation AB. In order to take this direction, we had to study their existing tools and processes in order to provide a suitable form of introduction. The ideas on proposed adaptation steps practicable by Bombardier have been thoroughly discussed with leaders and managers inside the company to gain a valid, valuable and solid solution fitting their current state-of-the-practice.

6.1 TOWARDS THE ADOPTION OF THE CONCEPT

The test script editor (TSE) provides the opportunity to distinguish inputs of the test case and expected results, as the VCS data are categorized as actions, reactions and neutrals. Actions manipulate the SUT through signals, controls and simulations and can be seen as rectangles with the letter A to the left. On the other hand, reactions expose the effect of the actions in the SUT or validate its state with signal indicators and simulations. Reactions are distinguished from actions and are represented by rectangles with the letter V to the left. Finally, neutrals do not affect the SUT, but they include advises that guide the test script execution, like parallel reactions, comments, timeouts and so on.

6.1.1 Preparation of the Test Script Editor

Towards the adoption of the concept a reinvention of the TSE is needed, because it is not in a good state as the other applications in the TAF. Bombardier put effort in making newer versions of TE and TEM, but the TSE was left out. Bombardier wanted various improvements for TSE such as:

- Improve the architecture with maintainability and scalability as the main properties
- Improve the GUI to make it easier to work with
- Improve the current writing and reading of files
- Improve the performance when using the application and during start-up

To tackle these demands, a new application was built from scratch with a total new architecture. The previous architecture, as seen in Figure 39, was combining logic and graphics which is widely considered as bad practice. This made the code very unclear and a first logical decision was to separate these into two components. Then with the help of the principles of SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) further improvements of the internal structures of the components were made.
Furthermore, to improve the applications performance, threads were introduced to handle calculations, reading of files and database querying. The new application is five times faster to start than the old one and this reduced a lot of annoying delays from the user perspective. To improve the workflow, or usability, major changes were included, such as drag-and-drop, common keyboard commands, search functionality and better organization of the objects to insert in the test script. Additionally to the wanted features, further enhancements have been realized. Some of these are:

- The level of testing can be chosen and depending on the level, different objects are available for creation of test script.
- Read, create and edit sessions within the TSE. This feature was highly desirable for the TEM since a user can easily load a whole existing scenario instead of loading it script by script. The user can then handle the whole session, update the wanted parts and then save it down to a new session and run it with TEM.
- Edit mode is introduced to enable various users to read the same file from the VSS at the same time.
- The parser, to enable reading in a test specification and automatically generate a test script. This will be described deeper in the upcoming subsection.
- Several test scripts open for edit in the same instance of the application. Before the application was just handling one test script but now the application includes tabs, where each of which handling one test script.
- A search tab for quick access of signals or simulations. This enables the user to insert a set of characters and the application will provide all matches on the set. This feature helps the user when a substep shall be added and the name is known, but the location of the wanted action or validation is unknown.

When it comes to the graphical user interface most of the previous design was used. This decision was taken to make the environment recognizable for the user and the way of working similar to previous version of the application. However, as seen in Figure 40 various changes can be distinguished from the
old interface. One of the major changes is the tabs on top, these are the tabs for switching between opened test scripts, and another is that the tabs on the left are now sorted on source and not on type. In those tabs the appearance is also changed, now both action and validation are shown in the same tab. The final major change is the removal of the central part of the GUI, where the user was able to choose what step to show. This feature is now displayed as tabs on top of the substep list instead. In the top-right corner four labels were added to display available valid sources for substeps.

![New GUI of TSE](image)

**Figure 40:** New GUI of TSE.

By implementing a new test script editor on top of the test automation framework provided deep knowledge on how the scripts are formatted and what they contained. Moreover, we gained a good understanding on how the framework itself works, e.g. communication with the VCS, internal structures. This knowledge was helpful to be able to recognize what was needed to be capable of generate test scripts from another source.

### 6.2 Applying the Concept

With the current constraints within Bombardier an automated transformation of requirements to a test script or test specification is not possible at system level. The needed information to build appropriate test scenarios and acquire a broad system level knowledge is missing (i.e. no dependencies between requirements). Therefore, a 1 by 1 requirement coverage can be achieved after the use of controlled natural language (CNL).
6.2.1 Applying Step One

By introducing CNL in both requirements and test specifications most of the present ambiguity can be removed and also parsing from the test specification to test scripts as a first step in the automation process can be enabled. Since the establishment of the concept is a long term process in an industrial setting an alternative path was also introduced, to automate the process from the test specification into test scripts and it is described in section 6.2.

When considering how Bombardier could improve its practices in requirements specification and move towards a controlled natural language, extensive work has to be applied. Nonetheless, thanks to the knowledge gained and the effort devoted in the realization of this work, certain conclusions can be drawn and initial improvements can be suggested, also taking into account projects-specific needs. Since Bombardier is trying to maximize the reuse of solutions coming from previous projects, also the keywords that are going to be specified can be potentially carried by one project to another. This will in the end give a good and mature set of keywords and syntaxes for requirement specification. The improvements that can be proposed right now are:

1. Agree on keywords to use to describe a common function. For example, to describe the state of the lamp just one keyword shall be used.
2. Analogously to the keyword of the state, make an agreement on how conditions shall be written in what order. Should the condition come before or after the action and shall the keyword if or when be used.
3. Avoid specifying requirements that are ambiguous, this is a big challenge, but when the CNL is built it would be mitigated anyhow.
4. Repetitions and overlapping requirements have to be removed.
5. Keep the keywords compliant with the VCS database

In addition, since applying a CNL to a requirement specification would require extensive workload, a CNL was instead created for the test specification that are targeting at system level. To achieve a good understanding on how Bombardier could proceed to create an appropriate CNL in order to enable a test specification, an investigation was made. This investigation was carried out by deeply analyzing the current test specifications, in all testing levels, and several meetings with test leaders and testers were held. What could be found was that the sub-system test specification in general contained a set of keywords and almost a single way of specifying the test cases. However, the same could not be said for tests at higher levels, e.g. commissioning specification. To come up with a CNL that both satisfies the demands from the automation point of view and the usability for a tester, who uses it, was a challenge. At first, a “generated” version of a test specification is made out of the TSE. Thereafter, a mixture between the keywords already used within Bombardier, the keywords needed by TAF and the old test specifications are combined into a new test specification. Later, this new specification was discussed during meetings and refined until the demands were fulfilled and a common CNL was created. The outcome of the investigation is a small CNL that covers the main features of the system and can be used in the regression test specification, as well as for other test specifications. The language is built in a way to optimize the learning curve of the testers, that is by adopting a syntax as similar as possible to the current keywords used in the company. Nonetheless, the current CNL has to be considered as a first language prototype to be validated and iteratively extended/refined by taking into account different projects and usage contexts.
The proposed language is built using a set of keywords strictly linked with the available database of the VCS. This interconnection enhances test specification correctness, since if an operation not present in VCS is used it will be considered as unrecognized. As shown in Table 7, there are five specific patterns where the bold text is keywords and italic text is either a state, or simulation name available in the VCS database, or a numeric value. The underlined text represents optional text that can be used to specify the location of where the simulation can occur, e.g. the keyword all can be used to specify that it shall occur in all available cars. The first three patterns are specifying an operation that shall be either performed or validated on the system under test and the fourth is used to specify that there shall be a snippet available to set the SUT in the specific state. The last one is used to pause the execution for a number of seconds to make sure that the simulator has reached the desired state.

Table 7: A CNL developed to be used inside of a test specification.

<table>
<thead>
<tr>
<th>Id</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Set Simulation Name to value in car/all</td>
</tr>
<tr>
<td>2.</td>
<td>Make sure Simulation Name to value in car/all</td>
</tr>
<tr>
<td>3.</td>
<td>Check that Simulation Name to value in car/all</td>
</tr>
<tr>
<td>4.</td>
<td>S: Snippet Name in car/all</td>
</tr>
<tr>
<td>5.</td>
<td>Wait time in seconds</td>
</tr>
</tbody>
</table>

To enable automation from the specified CNL, a parser was developed as an extension to the TSE, see Figure 41. This extension parses the CNL and creates objects that contain the information needed from the specification. Later, these objects are used to search in the VCS database for matching patterns. If a pattern is matched it is inserted in the test script. On the contrary, if a pattern is not found a warning message is displayed to warn the user that the parsing is invalid. Since the script is shown graphically, with the help of TSE the user can easily understand where the fault is introduced and fix the problem. The graphical representation of the scripts also enhances the opportunities of validating a script when it has been created, since the tester can have a quick look to ensure its validity.
Further changes to be introduced by Bombardier are related to the way the test scripts are currently being created, in the sense of the aim. Currently the scripts are mixing black-box testing with the help of simulations and white-box testing with the help of signals. However, since the level of testing is system, it is enough to use the simulations. Therefore, the commissioning test specification shall not mention any signal names, but only the actual action or validation that shall be performed. This could be not only beneficial in the sense of automation, but would also drive Bombardier in creating test specifications at the correct level and therefore increase the overall quality of the tests. Such a change would even not be demanding in general, since the commissioning test specification is written late in the process when all simulations inside of the VCS are available, thus making the mapping easier, and requirements can be considered as solid.

6.2.2 Applying Step Two

After the realization of the first automation step, proposed in our concept (section 5.1) and a CNL has been built, the company can move into specifying the requirements using a scenario-based approach. Scenarios is an already used way to group the system into functions and describe them as normal, alternative and exceptional flows of steps in the safety related requirements. If Bombardier starts to apply those specification methods observed in the safety requirements also in the normal requirement documents the realization of the second automation step will occur quicker. However, in the safety requirement document we detected some issues in the way they are documented that propagate problems in other development phases. For instance, a set of requirements is usually merged into one requirement, which is something that should be avoided in the normal requirements to facilitate the mapping of requirements and their test cases. Therefore, a way of describing the requirements in a scenario-based way, but using individual requirements has to be established. What has to be kept in
mind is that the requirements shall still be specified using the CNL that was developed in the previous step.

To move towards this scenario-based approach of specifying the requirements, Bombardier probably would need to analyze the way requirements are currently partitioned. Thus, some scenarios would probably reach over several subsystems and therefore it might be useful to be aware that division of the requirements using functions could make the transition easier.

6.2.3 Applying Step Three
Since the long-term goal of the concept is to reach an automated mapping of the requirements into a model that can later be traversed into test cases, Bombardier is expected to spend several years in the progressive improvements of the requirements. However, as mentioned in the state-of-the-practice Bombardier is developing a simulator that is simulating the train to enable early stage testing. This simulation is actually a model of the system, since it contains the states and the gates to go from one state to another. So, if Bombardier would manage to adapt the concept to be compatible with the states and gates available in the VCS a mapping could be done.

The need of a parser that can generate a model, based on the scenarios specified in the requirement specification is a fact. This model shall then be consistent with the development of the VCS. With this model Bombardier can later execute the test cases generated with the help of the TSE. Since the TSE is taking the states and gates from the VCS, a parser, or similar, can be created to automatically create test scripts from the requirements. Moreover, the TSE can be extended to generate a test specification for the assessors. This will not only provide a valid test specification, it will also improve the confidence that the test specification is 100 percent corresponding to the actual test executed on the system.

6.2.4 Example of Applying the Concept
Together with a requirement manager we tried to apply our full concept to a set of requirements, to provide a future vision of what Bombardier could reach if effort was spent on applying the developed approach. For this a set of requirement that describes the function “lamp test” were gathered, these are shown in Table 8. As it can be seen, it is a rather simple function divided into only six different requirements. However, even though the number of requirements that specify the function is little, still inappropriate information can be found. For example, this requirements shall aim on system level, but as shown in Req-2 and Req-5, units and internal communication is already mentioned at this level. There is also ambiguity in Req-0, on whether the test shall last for 5 seconds after the 3 seconds and what happens if the lamp is released after 4 seconds. Further things to pinpoint is how Req-3 says that the test shall begin when the lamp test button is pressed and how Req-4 with 2 sentences just add the word simultaneously as added value.
Table 8: Lamp test function before applying the concept.

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req-0</td>
<td>The lamp test shall be triggered in active cab only if a rising edge of the lamp test push button is detected and the button has been pressed for 3s. The lamp test shall be active as long as the lamp test button is pressed for max 5s.</td>
</tr>
<tr>
<td>Req-1</td>
<td>The following lamp shall be illuminated:</td>
</tr>
<tr>
<td></td>
<td>· parking brake lamp</td>
</tr>
<tr>
<td></td>
<td>· PEB bypass lamp</td>
</tr>
<tr>
<td></td>
<td>· serious fault lamp</td>
</tr>
<tr>
<td>Req-2</td>
<td>The lamp test command shall be forwarded to CCU-O.</td>
</tr>
<tr>
<td>Req-3</td>
<td>Lamp test of TCMS controlled indicators (gauges, lamps) shall be done by manual command from the activated desk. If the lamp test button is pressed, all indicators shall be tested:</td>
</tr>
<tr>
<td></td>
<td>· the gauges shall move from min to max value and back to current value</td>
</tr>
<tr>
<td></td>
<td>· the lamps shall lit in steps of different intensity if applicable</td>
</tr>
<tr>
<td></td>
<td>· the IDU (the primary and the secondary) shall show a white screen testing all pixels.</td>
</tr>
<tr>
<td></td>
<td>· Buzzer shall sound for 3 seconds</td>
</tr>
<tr>
<td>Req-4</td>
<td>When the lamp test button is pressed all indicators shall be activated simultaneously, the software controlled and the hardware controlled buttons. The synchronization of all indicators shall be ensured.</td>
</tr>
<tr>
<td>Req-5</td>
<td>CCU-O shall activate lamp test of CCO-U control lamps and also high the lamp test order digital output</td>
</tr>
</tbody>
</table>

So with this set of requirements gathered and deeply studied we asked one of the requirement managers to try and apply our concept to them. The requirement manager was not restricted to any kind of keywords, so she chose the vocabulary. The only restriction set, was to follow our concept and the five points mentioned in section 6.2.1. The result is shown in Table 9, and the first thing that can be noticed is the amount of requirements needed, now four requirements is used instead of 6. Another important aspect is the way sentences are written, Req-0 starts with the word Begin, which means that this is a preconditions to the function, and then Req-1 explains what the trigger is to enable the lamp test. Req-2 begins with the keyword After, this means that this requirements is going to happen after some trigger has been executed, in this case the press button for three seconds. Req-3 is a condition requirement that starts with the keyword If and it specifies what will happen if the actor is releasing the button from when the test has begun or when the time since the button was pressed is longer than eight seconds. One remarkable thing here is that the time is now changed to always start counting from the time where the button was pressed. Also, in this way of specifying, with the set of keywords, a scenario is built but with the usage of individual requirements and therefore a mapping to a model should be possible.
Table 9: Lamp test function after applying the concept.

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req-0</td>
<td>Before the driver will be able to press the lamp test pushbutton the cab shall be active</td>
</tr>
<tr>
<td>Req-1</td>
<td>The driver shall be able to test the audio/visual indicators by pressing the lamp test pushbutton for at least 3 seconds (on the driver’s desk)</td>
</tr>
</tbody>
</table>
| Req-2 | After the driver presses the lamp test pushbutton, all indicators shall be activated simultaneously:  
  - The gauge shall move from min to max value  
  - All lamps shall be turned on  
  - Both primary and redundant IDU shall show a white screen testing all pixels  
  - The buzzer shall be sound for 3 seconds |
| Req-3 | If the driver releases the lamp test pushbutton (after having it pressed for 3 seconds) OR If the lamp test button is pressed for 8 seconds, all indicators shall be deactivate simultaneously:  
  - The gauges shall move back to the current value  
  - All lamps shall be turned off  
  - Both primary and redundant IDU shall show a black screen  
  - The buzzer shall be muted |
7 Discussion

During this thesis a lot of knowledge and experience was gained. This was our first project that was fully industrial and it provided us with the picture of a real working environment surrounded by professionals employees. During the thesis a lot of information given during our studies could be seen in practice or just acted as the right background to understand technical concepts. We observed that there is a gap between the academic projects and industrial projects, since the constraints of the industry and especially a huge company like Bombardier are guiding process decisions.

Furthermore, since the thesis involves requirements and testing, it provided us with a lot of knowledge on these phases. For instance, we learned the specification rules that result in correct documentation of requirements and test cases and also familiarized with the execution of tests on a system. Going up one level, we also got the overall view of the development process, the deliverables after each phase and the flow of activities between the phases with their relations. Hence, the main parts of many development processes were analyzed together with existing development methods that could be applied. Another additional topic, which enriched our knowledge is the automation framework and the methods used to build and maintain one. Moreover, the value of time management and use of appropriate words during discussions and presentations allowed us to add confidence for our work and support our thoughts. Additionally, during the extensive research phase a lot of knowledge was gained in the research areas of requirement, testing and how these can be combined in an automated manner.

The thesis was mainly about developing a concept and with the help of Bombardier try to recognize what limitations the industry is setting when such a concept shall be applied. With the work done, a conclusion can be drawn that applying such a concept into a large organization, like Bombardier, would require an enormous amount of time and extensive work. However, we believe that by dividing the concept in several steps, letting the industry slowly adapt, would make the process easier. A realization of the concept could not be done, as the required time and effort inside the company would exceed the thesis timeline, which is limited to five months. Even though the concept is not fully applied, taking into account our example transformation in section 6.2.4, a seemingly improvement of the current status of the requirement specification is noticed. This change could improve the quality of the document’s language in a level, which will decrease the introduction period of newly employed engineers. Moreover, if the company that will apply the concept is already using CNL in the requirements, or specifying them using scenarios, the sequence of the steps can change and adapt to the company’s current work process.

Also, if more time was available further industrial partners could have been added to strengthen the definition of a generalized concept and provide more information of the limitations that the state-of-the-practice is setting.
Generating test cases from requirements is a difficult approach that has over the years been improved. However, with all the difficulties that it involves it can be seen as a rather hard goal to fulfill. During this thesis we have made a literature study in the state-of-the-art to find the approaches available trying to achieve the automatic generation of test cases from the requirement specification. Also, with the help of an industrial partner an investigation of the state-of-the-practice has been performed to get the understanding on what might be applicable in an industrial setting. With the help of the gained knowledge a generic concept has been proposed in order to support the industry to move towards this automatic generation. The concept is divided into three steps where the first step is an introduction to a controlled natural language and throughout the steps moves towards a scenario-based CNL by mapping keywords in relation to a model, with its states and transactions. This model is then the origin of the test case generation. To summarize, the concept is about mapping the requirements to a model that later can be traversed into test cases.

The thesis also involves work in partly applying the concept to the industrial partners setting. Work in the requirement specification was made, but also a mapping between the test cases described in the regression test specification and the test scripts were made. To enable the mapping of test cases and test scripts the developed parser extension of TSE is used. For the parser to work, a CNL was specified that can be used to describe the actions and validations that can occur on the SUT. Also, the concept was applied on one system function and this can be used as a futuristic view and it enabled the chance of test the concept. Furthermore, some future thoughts are explained on how they can start to map their current requirements with their developed abstraction of the system. To conclude, the industrial setting in which this thesis work was developed is far from enabling the automatic generation of test cases from the requirement. However, the descriptions of safety requirements is close to what the concept is aiming to and the tools available can help them move quickly towards the generation. As a conclusion, Bombardier is using many of the techniques specified in the concept but in other contexts than specification of normal requirements. If they find a way to apply their current usages in the right context, the generation can be reached with a smaller amount of work.

Further work can be applied for the current work. One of the things that can be done is a feasibility study to clarify how feasible it would be to apply the concept in an industrial setting and another path would be to apply the concept in another domain than the railway transportation to give a broader understanding of the genericness of the concept. Also, an involvement of several industrial partners could be beneficial to provide a better understanding on what limitation the industry is posing when such a concept shall be established. Finally, a realization of the applications mentioned in the concept is needed to fully enable it, since they play an important role when it comes to automate the process between the described steps.
9 Bibliography


[10] Rashmi Mascarenhas, Developing and Implementing an Automation Framework


[17] Lizhe Chen; Qiang Li, Automated Test Case Generation from Use Case: A Model Based Approach, Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference


[53] Christian Kop, Heinrich C. Mayr, Mapping functional requirements: from natural language to conceptual schemata

[54] G. Fliedl, Ch. Kop, W. Mayerthaler, H.C. Mayr, Ch. Winkler: Guidelines for NL-Based Requirements Specifications in NIBA, in M. Bouzeghoub et. al. (eds.) 5th Int. Conf. on Applications of Natural Language to Information Systems (NLDB2000), LNCS 1959, Springer Verlag, 2000, pp. 251–264


[60] Annamariale Chandran. Model Based Testing – Executable State Diagrams, Published in STEP-AUTO 2011, AVACorp Technology, Chennai