Master Thesis in Computer Science
With Specialization in Software Engineering

Enhancing the consistency between requirements and test cases through the definition of a Controlled Natural Language.

May 2015

Candidate:
Valerio Lucantonio

MDH Supervisor: Antonio Cicchetti
Examiner: Jan Carlson
UDA Supervisor: Alfonso Pierantonio
Abstract
Requirements and testing engineering build up the solid base for developing successful software projects. In particular along the development process, testing follows and relies on requirements engineering: an incoherent specification of one of the two can affect the correctness of a project resulting in delays, failures, unhappy customers and other consequences on the project delivery.

Especially nowadays software companies are competing in fast changing markets where the delivery time of the products is the most crucial aspect and it really affects the quality and the success of the product. Given the semantic gap between requirements (typically written in natural language) and test specifications, it is not rare that requirements are misunderstood leading to erroneous tests. This risk is even more relevant when trying to perform early validation, since testing is mainly based on requirements definition.

This thesis work introduces an investigation to close the gap between requirements specification and test cases by providing automatic test case generation. Requirements are written in Natural Language, their subsequent restructuring in a more formal controlled natural language, and the final automatic translation derives test cases. The soundness of the concept is demonstrated through a practical implementation tailored to a previous real project developed at MDH. The implementation not only demonstrates the feasibility of the idea, but also shows interesting results in terms of generated test cases against the ones obtained for the project by hand.
Acknowledgement

I would like to give a special thanks to my supervisor at MDH Antonio Cicchetti for his endless support and encouragement during the whole thesis process. I would like to thank my parents and all my family for backing me up continuously during my life, encouraging me always to do my best. I would like to give a special thanks to my beautiful flower Serena for her patience and goodness during these years spent together. I would like to thank Genew Team for these years studying and supporting each other always together as brothers. I would like to thank all my real brothers and sisters: Tazza, Mirko, Marta, Charlie, Miky, Giuly to be always by my side regardless of the distance between us. Finally, a big thank to my GSEEM’s mates Giulio and Pina for the time spent together fighting against bureaucracy and exams. Last but not the least a special thanks goes to my Erasmus friends: Matej, Miguel, Gina, Betty, Gordo, Benny, Toni, Gabriele thanks for everything we made together, you are the best.
Index Of Figures:
Figure 1: Software Development Life Cycle ................................................................. 14
Figure 2: The V model .................................................................................................. 15
Figure 3: The Spiral Development Process .................................................................. 16
Figure 4: User vs. System Requirements [5] ............................................................... 17
Figure 5: Spiral view of RE process [5] ....................................................................... 19
Figure 6: Data Driven data file [16] ............................................................................ 25
Figure 7: Keyword Driven data file [16] .................................................................... 25
Figure 8: MOF architecture ......................................................................................... 27
Figure 9: Ontograph example [23] ........................................................................... 35
Figure 10: The process of model-based testing [25] .................................................... 36
Figure 11: Concept approach from specification written in NL to test cases (or test scripts) 41
Figure 12: Relation between PENS properties. White dots represent natural languages, black dots represent formal languages and blue dots represent CNLs (or CNLs classes). [20] .......... 43
Figure 13: First specification words and occurrences .................................................. 50
Figure 14: Second Specification parsed words ............................................................ 52

Index Of Table
Table 1 Codes for CNL Properties ............................................................................... 29
Table 2: PENS classes ................................................................................................. 31
Table 3: Classification of CNLs according with PENS classes .................................... 32
Table 4: existing CNLs classified by Kuhn in [20] according with their properties .......... 44
Table 5: Test Cases template of previous project treated as example. .......................... 58
Table 6 Delete Booking: original manual test case of previous project for requirement #2 .. 59
Table 7: Book Simulator: original manual test case of previous project for requirement #1 .. 60
**Abbreviations List:**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>Attempto Controlled English</td>
</tr>
<tr>
<td>BX</td>
<td>Bidirectional Transformation</td>
</tr>
<tr>
<td>CNL</td>
<td>Controlled Natural Language</td>
</tr>
<tr>
<td>DRS</td>
<td>Disclosure Representation Structure</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MDE</td>
<td>Model Driven Engineering</td>
</tr>
<tr>
<td>NL</td>
<td>Natural Language</td>
</tr>
<tr>
<td>PENS</td>
<td>Precision Expressiveness Naturalness Simplicity</td>
</tr>
<tr>
<td>RE</td>
<td>Requirements Engineering</td>
</tr>
<tr>
<td>SDLC</td>
<td>Software/System Development Lifecycle</td>
</tr>
<tr>
<td>SRS</td>
<td>Software Requirement Specification</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>TTM</td>
<td>Time To Market</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>MBT</td>
<td>Model Based Testing</td>
</tr>
</tbody>
</table>
Summary

1. Introduction .......................................................................................................................... 8
   1.1 Motivations ..................................................................................................................... 8
   1.2 Thesis Contribution ....................................................................................................... 9
   1.3 Research Method ........................................................................................................ 10
       1.3.1 Concrete Search Parameters ............................................................................. 11
   1.4 Outline .......................................................................................................................... 12
2 Background .......................................................................................................................... 13
   2.1 Software Development Lifecycle Models .................................................................... 13
       2.1.1 Sequential Models ............................................................................................... 14
       2.1.2 Incremental Models ........................................................................................... 15
       2.1.3 Evolutionary Models ......................................................................................... 16
   2.2 Software Requirements Engineering ........................................................................... 17
       2.2.1 Requirement Characteristics ............................................................................... 17
       2.2.2 Requirements Validation ..................................................................................... 19
       2.2.3 Towards Requirements Formalization ............................................................... 20
   2.3 Software Testing Engineering ..................................................................................... 21
       2.3.1 Testing Techniques .............................................................................................. 22
       2.3.2 Manual Testing .................................................................................................... 23
       2.3.3 Towards Automated Support ............................................................................. 23
   2.4 Model Driven Engineering ........................................................................................... 26
       2.4.1 Models .................................................................................................................. 26
       2.4.2 Metamodels ......................................................................................................... 27
       2.4.3 Transformations ................................................................................................... 27
3 State Of The Art ................................................................................................................... 29
   3.1 Requirements Ambiguity Avoidance .......................................................................... 29
       3.1.1 Controlled Natural Language ............................................................................ 29
       3.1.2 CNL Frameworks ............................................................................................... 34
   3.2 Testing Automation ....................................................................................................... 37
       3.2.1 Model-Based Testing ......................................................................................... 37
       3.2.2 Scenario Based Testing ...................................................................................... 40
   3.3 Related work ................................................................................................................ 41
1. Introduction

Requirements engineering and testing represent the solid base for successful software projects. In software development, testing follows and relies on requirements engineering: an incorrect specification of one of the two can affect the project as resulting in delays, failures, unhappy customers and other consequences on the project delivery. Moreover, feedbacks of testing are often reflected on requirements in terms of refinements, additions or removals. Thus, collaboration and alignment between testing and requirements engineering are more than recommended, especially for complex software system development [3].

Testing represents a fundamental activity in software development, as it assures both reliability of the product and enriches the quality of the system itself. Specifying correct and complete test cases and ensure through their execution to cover all the existing requirements is a time consuming and error-prone task. The main reason is the gap between requirements specifications, typically formulated in non-formal notations, and tests. [4]

If testing cannot rely on well-defined requirements time will be wasted for interpreting and clarifying those requirements, and the mapping between requirements and corresponding test cases will be necessarily performed manually. Manual techniques on the one hand require a lot of human resources, and on the other hand make the testing frequently to be postponed too much during the development lifecycle [4].

In this scenario, specifying requirements through a formal language, different from the Natural Language (NL), would have the potentials of removing ambiguities in the specifications. Moreover, it would disclose the opportunity to create traceability links between requirements and corresponding test cases. These are currently two big challenges both in the software engineering research and in industry [9]. This thesis proposes to adopt controlled natural language (CNL) as an intermediate requirement definition mean, i.e. acting as a bridge between non-formal NL and automatically generated test cases.

1.1 Motivations

Software companies are now more than ever competing in fast changing markets where the demands of product and the delivery time is a crucial aspect and it really affects the quality and the success of the product. On the one hand testing is a fundamental activity which allows to deliver a well-defined and correct product. On the other hand it is a time consuming activity that affects software costs for the company and slows down the delivery of the product. Testing is usually performed at the end of the implementation, when costs for modifications are typically higher than in earlier stages of the development process [4]. In this respect, there has been a general effort targeting the revision of development processes in order to allow a more flexible kind of testing. There is a need of new testing techniques, and industries are slowly moving towards automated testing trying to use frameworks to speed up the testing process. Through these frameworks, it is possible to create test cases and to execute them on the System Under Test (SUT) in order to compare actual results with the expected ones. The
benefits of an automated approach are different, one of the most important is that the time to market is reduced sensitively. Indeed, automating testing can disclose the opportunity of starting the testing activity also before the implementation, and as a result it can allow saving time [9].

A precondition to make testing effective in general, and during the early stages of the development process in particular, is that requirements can be unambiguously interpreted and reflected to corresponding test cases. In turn, more precise requirements permit to introduce enhancements in the management of application evolutions by means of test cases traceability and change impact analysis [23]. Eventually, unambiguous requirements can disclose opportunities to (semi-)automatically derive test cases, thus introducing further consistency control in the development process.

The underlying main motivations for this works are therefore the need to speed up testing and a better quality of requirements that directly reflects in better test cases. In this way, requirements can be specified more formally granting a better test cases formulation, which, in turn, will reflect in increased software reliability as well as a short time to market.

1.2 Thesis Contribution

In order to enhance the consistency between requirements and test cases, this thesis work analyses different approaches for automating the definition of test cases as derivable from requirements specification. In particular, it investigates the traceability support between requirements written in Natural Language (NL), their subsequent restructuring in a more formal controlled natural language (CNL), and the final translation to appropriate concrete test cases. The objective discussed so far entails the following research questions:

RQ1: What are the state-of-the-art techniques to derive CNL from requirements definitions in NL?

RQ1.1 Are there any generic (i.e. non domain-specific) solutions?

RQ2: How is it possible to automate the mapping between requirements and test cases through the exploitation of an intermediate CNL?

After investigating the state of the art on how derive a CNL from requirements written in NL I proceed with an investigation of any possible general approach to the definition of a CNL. Based on the result found at this point I evaluate any possible approach to map requirements, written in CNL form, to test cases in an automatic way. Through the application of the research method described in the next subsection, a clearer landscape of the state of the art is provided. In this respect, there is no an existing attempt that tries to cover the whole activities from requirements definition to test cases. Moreover, partial solutions available in the literature are very often domain-specific.
The knowledge gained in the survey is then exploited to formulate a concept, which aims at proposing a generic process for the automatic derivation of test cases from requirements specified through a CNL. Given the domain-specificity of the problem, it is not possible to propose a generic solution implementing the various steps that build-up the automatic generation. Therefore, the actualization of the concept is illustrated through a real case study: starting from a project requirements specification, I create the mapping solution, that consists of a dictionary with a CNL, and a transformation between requirements written according to the CNL and test cases defined according to a corresponding template. The case study has the scope to show the soundness and feasibility of the concept. Moreover, despite its relatively small size, the example demonstrates concrete relationships between test cases generated through requirements and tests effectively performed during the project.

1.3 Research Method

This thesis work has the goal to analyze current approaches for enhancing the consistency between requirements written in NL and test cases. The proposed solution relies on the definition of a CNL that on the one hand will ensure consistency between requirements and test cases, and on the other hand removes ambiguity in the requirements specification. In this respect, the thesis has involved both a state-of-the-art survey and some implementation work.

For what concerns the research applied to this work, I started with a general search on testing and requirements of software systems through existing publications on international conferences, workshops and journals. This analysis gave me a better understanding on how the two phases (requirements and testing) could overlap in a development process and gave me the idea, developed in the concept, to formulate a general approach to reduce the gap (in terms of development time but also of consistency mechanism) between requirements in CNL and test cases.

Then I proceeded with the analysis of the state-of-the-art of CNLs related to software requirements specification. Through this analysis, I discovered that (to the best of my knowledge) there is no precise answer to the research questions RQ1 and RQ1.1. In particular the open problems related to RQ1 are that most of the examined CNLs are domain specific and not always in software engineering field. Moreover, they are often “embedded” in a tool, thus hindering the possibility of extrapolating general resolution guidelines. This analysis, although not completely successful, allowed me to narrow down the field of research according to precise constraints, defined later in this section. Moreover, it allowed me to propose a concept based on a CNL used in requirements specification as a support to the testing phase, and in particular to keep requirements and test cases consistent.

I investigated the state-of-the-art further in test automation by considering the automation starting from requirements written in NL (instead of CNL, since almost no work was compatible with the constraints applied to the search), and trying to apply/port available solutions in my setting, i.e. substituting NL with a CNL. From this analysis, being a CNL a subset
of a NL, it emerges that all the possible approaches devoted to the generation of test cases from requirements in NL are suitable for the CNL as well. Based on this fact, I could exclude the different kinds of test cases purposes from the concept, since the generation process can abstract from the specific intent as soon as a formal template describing test cases models is provided (analogously to NL).

Eventually, the concept has been formulated according to the material and knowledge gained in the analysis of the previous phases. It is concretized by the implementation of a prototype, based on a previous project done at MDH university. The realized prototype follows state of the art principles and techniques for the NL analysis, CNL template definition, test cases template specification, and CNL parser construction for deriving corresponding test cases.

1.3.1 Concrete Search Parameters

A generic query exploring the state-of-the-art related to requirements and test cases approaches would result in a huge amount of papers, with different scopes and prospective, that would be difficult to categorize in a meaningful way (for the scope of this work). Therefore, the research has to be guided and narrowed by the various aspects that characterize the topic. I introduced the following constraints:

1) Requirements specification has to be written in NL or CNL.
2) Test cases have to be generated from requirements specification.
3) The generation has to be automatic.

The databases chosen for the literature review are Google Scholar, IEEE Digital Library, and ACM Digital. Two queries have been formulated in each of the tree DB, for literature review: For RQ1 and RQ1.1:

("Controlled Natural language" OR "CNL") AND ("create" OR "generate from") AND ("Natural Language" OR "NL") AND ("software specification" OR "software requirements" OR "software requirements specification") For RQ2:

("auto generate" OR "generate automatically" OR "auto derive" OR "automatically derive") AND "test cases" AND "from" AND ("software requirements specification" OR "software specification" OR "software requirements") AND ("natural language" OR "controlled natural language" OR "NL" OR "CNL")

The queries generate 192 and 126 results, respectively. Moreover, articles published after 2005 have been considered at first (except for largely cited articles before 2005), published in English, that are composed of 6 pages or more, and are presented at least in a workshop. The resulting papers have been reduced, in order to select only significant papers, reading abstract, introduction and conclusions and deciding according to the contents if they are significant for the research. Among these results the research have been principally narrowed in the identification of 67 papers. From these papers the research has been guided from two main articles in the field:

• A survey and classification of controlled natural languages. T. Kuhn, (2014): a classification of more than 100 different CNLs representing the state of the art about
CNLs, the most cited publication in CNLs field with 47 cites. Through this article and the constraints defined to narrow the scope, it has been possible to reduce the amount of papers regarding the state-of-the-art of CNLs to 18.

• From Requirements Specification to Test Scripting: Towards Automated Support. R. Gustavsson, D. Kostopoulos (2014): a previous work realized at MDH suggested by the supervisor based on a side topic of this work thesis. Through the literature review in this work it has been possible to reduce the amount of papers for the state-of-the-art in test automation to 29.

1.4 Outline
Section 2 provides a background in software development lifecycle (subsect. 2.1), requirements engineering (2.2), testing engineering (2.3) and model driven engineering (2.4). Section 3 concerns the state of the art of CNLs, as a requirement specification language (3.1), and testing automation (3.2) with related works (3.3). Section 4 describes the concept elaborated thorough the various subsections, from NL specification (4.1), to the definition of a CNL (4.2) and the generation of test cases (4.3). Section 5 contains an example applied to a previous project at MDH, supporting the concept in the various steps. At the end of the paper section 6 contains a brief discussion of the result of this work, and section 7 a small conclusion with possible future works.
2 Background

Contemporary software applications and services are commonly characterized by a competitive market, in which products have to be delivered in a fast way, because the demand of a product hardly depends on the Time-To-Market (TTM). This affects companies that have to modify their assets to be more competitive for the market, and this has been reflected in the adoption of new development processes. In particular, a typical strategy includes the introduction of a new stakeholder, the customer, who can strictly follow the development process and give suggestions and improvements to developers in order to shorten the delivery time and to enhance quality attributes. In such described scenario the importance of precise and well-defined requirements is evident since they affect directly the quality of the final product. Equally important, ambiguities in requirements definition can remarkably impact the delivery time.

This section explains different development lifecycles and illustrates software requirements engineering and software testing engineering.

2.1 Software Development Lifecycle Models

In software development the choice of an appropriate development process is one of the most delicate aspects to be treated. Indeed, the process provides a framework model to follow for all the activities involved in a software engineering project. A development process supports all the lifetime cycle and gives the precise standard and steps for the realization of the software. These steps, depicted in figure 1, can be summarized according to [2] by:

Requirement Analysis and Specification: The first step of this analysis is to get all the requirements from the customers. Requirements are gathered, analyzed and formalized in a Software Requirement Specification (SRS) document that demarks the end of the requirements phase. This can in general also being not true, since, as we will see in the next subsections, requirements can change during the whole lifecycle.

Designing: Gathered all the requirements from the SRS they have to be translated in an architecture with a precise design structure. The output of this phase is the design document.

Coding: The implementation phase, is when the specifications contained in the design document are actually translated in code. The output of this phase is a software prototype.

Testing: The produced code is tested following different techniques, in order to evaluate the correctness and validness of the software.

Evolution and Maintenance: All the modification that can improve the product and all the routines to ensure the correctness of the software system. This phase is responsible of the maintenance of the implemented system and the implementation of new features.
2.1.1 Sequential Models

Sequential models are one of the first development process used to approach software development. The foundation of this process is that every problem can be completely understood and formulated through a design software solution [1]. According to the development process life cycle model, a design formulation follows a requirements analysis and is followed by a design. All the implementation can be done before validation and testing.

The characteristic of this model is that each phase strictly affects, upon its completion, the initiation of the next phase. The most referenced sequential model is the waterfall model, because it magnifies this tidy separation of the stages. The waterfall model is not a flexible development process especially for competitive markets. For this reason, the waterfall model evolved in more flexible models in order to support more competitive development.

This brought to one of the most used models, the V model. The V model is an evolution of the waterfall model, in a V shape as shown in figure 2. It is based on verification and validation of each phase of the process, so from the requirements analysis to the testing. The first part (left-hand side of the V) can be referred as Validation phase. It starts with requirements gathering and ends with detailed software design. At the bottom of the process there is the implementation phase. The right-hand side represents the Integration and Verification of the system components. It starts with unit testing and ends with user acceptance testing. It is worth noting that vertical axis also prescribe the level of the decomposition in modules of the system, and that each subcomponent can be developed using the V model process recursively [3]. In that particular case, as we will see in the next subsection, we can refer to the V model as an iterative development process.

The V model is widely adopted especially for safety critical systems, where it is important to define a strict process that assures that the actual product will meet safety requirements. In this kind of systems the interconnection between requirements and implementation is crucial. In order to have a strict control (traceability) over the activities of requirements definition and development of the system, the development process has to prescribe strict policies.
2.1.2 Incremental Models

Iterative models can be seen as three-dimensional sequential model, where the number of sequential models in the z-axis specifies the number of iterations, or increments, that have to be made in order to improve system functionalities. One of the most important advantages, as Wallin and Land write in [1], is that “With incremental development lifecycle models, risk of developing the wrong thing is reduced by breaking the project into a series of small subprojects (increments)”. This allows to have a faster feedback on system functionalities and at the same time allows to respond faster to changes in requirements. One risk with the incremental approach is that the first releases address such a limited set of requirements that the customer could be dissatisfied, however keeping open the opportunity of fixing wrong or missing requirements on time. Moreover, using an iterative approach allows to involve new stakeholders at different times and iterations of the process, thus making the definition and refinement analysis of a particular functionality more effective.

Examples of incremental models are the staged-delivery and the parallel model. The former is characterized by the construction of a subsystem at a time, while the integration is performed indifferently at the end of the process or release-by-release. Instead, in the parallel development the subsystems are developed in parallel. In this way it is possible to shorten the TTM, although there could be problems in integration. Additionally, this development process doesn’t assure fast response on requirements changes.
2.1.3 Evolutionary Models
The border between incremental and evolutionary model is not really well defined, but we can definitely say that one of the biggest difference between them is that in the evolutionary models not all the requirements are available in the beginning of the process. For the remaining aspects, this process shares most of the characteristics with incremental models. One of the most used evolutionary model is the spiral development model, depicted in figure 3, defined by Boehm in [4] as:

“Spiral development is a family of software development processes characterized by repeatedly iterating a set of elemental development processes and managing risk so it is actively being reduced.”

Another definition, always from [4], that stresses the relationship between spiral and the other development processes is:

“The spiral model is actually a risk-driven process model generator, in which different risk patterns can lead to choosing incremental, waterfall, evolutionary prototyping, or other subsets of the process elements in the spiral model diagram.”

The spiral model is characterized by two properties: is a cyclic approach for incrementally growing the system while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder feedbacks. The evolutionary model’s disadvantage is that it is complicated, indeed it is hard to plan and follow up and it might not be worth the effort using it if the development is manageable enough, with low risks. Another disadvantage is that the architecture of the first version of the system must support the changes introduced in each cycle. Otherwise, you will have to redesign your system completely, even though the experience acquainted from earlier cycles is very valuable input when doing this.
2.2 Software Requirements Engineering

“The requirements for a system are the descriptions of what the system should do, the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose [...]. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).”[5]

Requirements is the earliest phase of the development lifecycle. They state, independently from their implementation, the desired goals and achievements that the future software should reach. The main purpose of requirements engineering is to synthetize user requests in a clear, consistent, and unambiguous set of problem statements. For the sake of understandability and management, requirements are typically distinguished in the following two categories:

User Requirements: are statements written in a natural language with the use of some intuitive diagrams. Their aim is to provide an initial system description in order to allow engineers to model them and to formulate a more formal specification that will be exploited during the entire development lifecycle.

System Requirements: are a more detailed description containing a full formulation of features, services, and constrains that the final system must provide. The system requirements must provide an unambiguous formulation of the system to be implemented.

Figure 4 explains, through an example, the demarked difference between user and system requirements.

![User Requirement Definition](image)

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

![System Requirements Specification](image)

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 4: User vs. System Requirements [5]

2.2.1 Requirement Characteristics

Requirements engineering can be summarized in an iterative process, as shown in figure 5, starting from a feasibility study, followed by collection and analysis of requirements, translation in a formal specification, and ending with a validation of the actual requirements.
The duration of each phase is strictly dependent on the previous phase precision. At the end of the spiral a Software Requirements Specification (SRS) has to be released and each requirement in it can be considered as well-defined if it is:

**Complete**: means that there is not missing or to-be-determined information (also known as internally completeness) and the information defined does not contain undefined referenced entities (or external completeness).

**Correct**: it is formally expressed by a combination of consistency and completeness. Consistency refers to the attribute of not having contradictions while completeness refers to the point above. From a practical point of view, a correct requirement fully satisfies certain business goal [7].

**Feasible**: it is actually possible to satisfy the requirement considering its constrains, scenarios, limitation and current budget.

**Necessary**: a requirement should be specified only if it contains information not contained in other requirements, and can give an added value according to its expected functionalities.

**Prioritized**: each requirement should be prioritized, following a metric, in a way that engineers are aware of its importance, and according to the stakeholders it is possible to make a plan based on this priority.

**Unambiguous**: this is a key aspect in requirements definition. The nature of the word unambiguous does not assure anything to the engineers of the project, however it has to be clear that the requirement should not be understood in more than one way.

**Verifiable**: It is crucial for a project’s success to verify the expected behavior of the product. Also this concept is very hard to apply in practice, as it basically prescribes that each requirement should entail a way to prove its satisfaction by testers, or even better, by test cases as we will see next.

This spiral can give the false perception that after the requirements document specification is released the requirements engineering process is completed. This is something not true in general, because requirements at the end of the product are usually slightly different from the ones in the beginning. As Kotonya and Sommerville say in [6]:

“The notion of ‘completeness’ in requirements definition is problematic. There is no simple analytical procedure for determining when the users have told the developers everything that they need to know in order to produce the system required. Requirements are never stable. Changes in the environment in which the system has to work may change even before the system is installed, due to change in its operational environment.”

This can depend on many reasons, above all due to conflicts among stakeholders; indeed, it is not always easy to reach agreements and to write requirements in a formal way such that they mean the same thing for stakeholders with different backgrounds. Conflicts can depend from
everything: different user demands, changes in the business asset, additional constraints or other thousands of reasons.

Figure 5: Spiral view of RE process [5]

2.2.2 Requirements Validation
At the end of the spiral requirements should be validated in order to check that they actually define the system that the customer wants. The validation process allows to remove errors in requirements definitions, and to limit the cost of corrections during the development. This cost in fact, is much higher than repairing requirements right after their definition. The main reason is that changing afterwards often implies changing design architecture and implementation, and likely testing as well. According to Sommerville [5] the activities that have to be involved in the validation process have to be:

Validity checks: analyzing current requirements to understand if all the features required for the system have to be specified through the specified requirements. In case, identify and add other requirements in order to specify new functions.

Consistency checks: there should be a clear and not-conflictual specification. Contradictions in requirements should be erased and conflicting constrains analyzed and resolved.

Completeness checks: it is always worth trying to verify that requirements are complete, although, as said before, it is not easy and there is no way of proving it formally.

Realism checks: using the state of the art of the technologies, the requirements should be checked to ensure that they can actually be satisfied by a corresponding implementation. These checks should also take into account budget and schedule for the system development.
**Verifiability:** in order to reduce the gap between system and customer requirements, requirements should be somehow verifiable. Therefore, it should be possible to demonstrate that the system released meets the requirements contained in the SRS.

In order to validate requirements different techniques have been proposed. Among them:

- **Requirements reviews:** requirements are reviewed systematically by a team of reviewers who check for errors and inconsistencies.
- **Prototyping:** a prototype is implemented in order to give a real proof of the system. The end user usually tests the system to verify that works according to the expected behavior.
- **Test-case generation:** This is one of the key point of this thesis work, and it will be explained better in the following subsections. The basic idea is that requirements should be testable. Testing the requirements is a good strategy to save efforts in the design phase of the system. Indeed, generating test cases from the requirements often uncovers problems in requirements, notably difficulties in the implementation and/or needs for clarification.

### 2.2.3 Towards Requirements Formalization

Writing requirements in the natural language (NL) is one of the most used approaches, both in research and in industry. The expressiveness offered by a natural language represents an added value to the definition, however it is high probable to have ambiguous, informal, and incomplete specifications. Several approaches have been introduced in order to mitigate the problems mentioned so far, and hence to obtain a more formal specification. Notably, in order to have a more structured approach, standard templates have been proposed such that requirements writers would use a subset of natural language in the writing procedure. This approach can limit the ambiguity of the requirements while at the same time granting expressiveness and understandability of natural language. A feasible template for a more formal requirements specification according with [5] should contain at least:

1. A description of the function or entity being specified.
2. A description of its inputs and where these come from.
3. A description of its outputs and where these go to.
4. Information about the information that is needed for the computation or other entities in the system that are used (the ‘requires’ part).
5. A description of the action to be taken.
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

The problem of having not well defined requirements using NL is well known, as well as the need for more formal designs on the way towards a complete implementation. These formal
requirements would support not only rapid prototyping of the desired software systems but could also provide a standard model upon which all successive implementations would be based on. Since object oriented modeling using UML and similar tools is a de facto standard for software system design, there is a need for a requirements specification language that might play an intermediate role between the original NL specification and a corresponding object-oriented design. One technique of object-oriented requirements engineering proposes to define the objects that will be used in the design by using the same nouns written in the requirements specification, and define the interactions among these objects as well as their operations using verbs and their objects [8]. The mapping between user requirements and system design is problematic and some additional tools can be used to facilitate the process. For a broader description of the approaches dealing with requirements specification the reader is referred to the state-of-the-art section.

2.3 Software Testing Engineering

“Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the programs non-functional attributes.”[5]

One goal of software testing is to prove (to the developers but also to the client) that the software meets its requirements. This means that all the requirements must be tested at least once. A part of testing deals with validation, that is it is devoted to verify that the right system was implemented. In this case, based on the requirements specification, test cases will verify the system against expected inputs to show that it behaves as prescribed. Another goal of system testing it to discover errors, failures, and incorrect behaviors according to the requirements specification. This part copes with defect testing and aims at revealing imperfections. As a consequence, the test cases are built-up on all the possible inputs the system could receive.

It is clear that the border between the two kinds of testing is not really demarked and the activities often overlap. Indeed, during validation testing some errors may appear, and these are reflected to, and originate from, defects that will be discovered by defect testing.

Testing can be divided into two main classes, white box testing and black box testing. Black box testing consists of testing the whole application ignoring the source code and focusing only on the output that a test case gives in response to some input. White box testing, commonly referred as structural testing, is performed on the code, and test cases are conceived by taking into account the internal structure of the software. White box testing is mainly used for testing the code from a structural point of view, looking for dead or unreachable code, and checking all the possible branches and system states. Instead, black box testing is performed to test the features of the developed application.
A test case, as referred before is the definition of how a test can be performed on a desired feature. It contains a set of inputs, a set of (expected) outputs, pre and post conditions, and usually also more information about the failing and passing criteria. A test suite is a set of test cases, put together in order to specify a more complex scenario to test. Usually the test cases are divided into subsystems according to the functions and the structure of the application. Test cases then will be run with different approaches. At the end of the test running phase a document should be delivered, that reports all the information about the executed test cases and their outcomes.

2.3.1 Testing Techniques
Testing techniques are typically tailored to the used development process. We can summarize the different techniques in the following list:

**Unit Testing:** is a pure white-box testing approach based on low-level implementation. Unit testing can be defined as the process of testing program components, such as methods or object classes [5]. Unit testing does not ensure to check all the functionalities of a program, rather it is used to guarantee that building blocks of the software work as a whole, independently from each other. It is performed by testers or engineers during the implementation phase of the software development lifecycle. Typically its goal is to test all the functions and the methods of a certain unit, therefore it usually exploits different inputs. Testing objects and classes implies the coverage of all their features, hence the operations within an object, object attributes (against their possible values), and all the possible statuses an object may acquire.

**Integration Testing:** depending on the used approach it can be either a black box or a white box technique. Integration testing consists of testing more units combined in a larger component. Testing is done mainly on the interfaces of the units plugged together, in order to ensure that the communications between them have the correct behavior. Integration testing is done iteratively until all the units are integrated in a system that works as a whole.

**System Testing:** it is a black-box approach and is performed at the end of the software lifecycle. After all the components are integrated together composing a version of the system, the whole software is tested to check that all the components are compatible and interact correctly exchanging the right input and output across their interfaces. The testing is usually performed by someone that does not know the internal details of the system, in order to ensure an unbiased procedure, closer to end users. This kind of interaction testing should discover bugs that are not revealed by the integration and unit testing. In system testing also performance and non-functional requirements should be proved.

**Acceptance Testing:** it’s a black box approach based on requirements specification and performed at the end of the lifecycle, just before delivering the product. The main purpose of acceptance test is to provide an agreement (with the customer) on the developed system and deliver the final product. Therefore, it is mainly run in order to demonstrate that the system
works as required. The test suite is run with different inputs, and the outcomes are compared with the expected outputs. If the suite meets the expected results acceptance test is successful and the system can be delivered. Otherwise, the system is rejected and discovered bugs have to be fixed previous a new delivery attempt.

*Regression Testing:* it can be used either as a black box or a white box technique. In regression testing a test suite is developed incrementally, while the program is developed. Hence it can be collocated across the implementation phase in the spiral lifecycle, but depending on which testing technique is applied it can be also in the design phase. Essentially, this approach consists in rerunning all the existing test cases together with newer ones, introduced when units are joined together. In this way it is possible to check whether the integration process causes bugs or undesired states of the system.

### 2.3.2 Manual Testing

Manual testing is a widely used approach and consists of manually writing test cases using appropriate techniques. Testers write test cases manually: on the one hand testers’ skills and knowledge positively affects the efficacy of tests. In fact, it is scientifically proved that manually written tests entail very good results in terms of design and execution, especially in small projects. On the other hand, when the size and complexity of the software grow, it becomes more and more difficult to keep an acceptable quality level [10]. There are two main kinds of manual testing approaches [13]:

*Code-Driven Testing:* is a kind of white box testing and aims to test various sections of code to verify if the system behaves correctly. Some frameworks, such as jUnit, xUnit, pyUnit, Gobo Eiffels, have been provided to support automation in test case execution. Test cases are written according to the technique chosen by the testers. The purpose of the test cases is to trigger a particular state or behavior in the object to assure that the outcome will be consistent with the expectations. Code Driven testing is usually used in agile development techniques and runs during the whole development process.

*Graphical User Interface (GUI) Testing:* is a black box approach based on running the software from the GUI interface simulating the user choices. Each time the system is run, the interaction is recorded for comparing the current outcomes with all the previous simulations. There are many tools providing record and playback features to interactively record the simulation. The big advantage of this method is that it does not require any specific knowledge in a testing environment, since it mainly consists of exploring the application from a user perspective. A disadvantage of this technique is that it is not possible to reuse previous test cases, since they are simple recordings.

### 2.3.3 Towards Automated Support

A notorious problem with software testing is its costs, both in terms of human resources and time. In fact, the growing complexity of software applications entails corresponding testing
efforts, and in a competitive market long testing phases collide with the general need of reducing TTM. As a consequence, manual testing appears to be a relevant bottleneck on the way of trying to optimize the validation and verification activities. Automating testing is a relevant topic both in research and industry, and is recognized as a valid approach for alleviating the problem of testing costs. The main idea is to support testing through the whole software development by automating the process. There are many approaches to provide test automation, and also a variety of software frameworks to support the activity. According to [14] three of the most used approaches to test automation are Capture/Playback, Data-Driven, and Keyword-Driven. The selection of the approach to use for automation is based on different parameters, notably testers’ skills, the development phase in which the testing process is placed, and others.

Capture/Playback: This can be seen as the manual GUI testing, but with the big difference that the actions are captured in background by a tool; in this way, it is possible to record also the testing activities to enable test cases reuse. So this approach relies more on test scripts that the GUI approach, although the recording features are performed from a GUI point of view. Therefore, testing is firstly executed manually, to create a test scenario, and then recorded in order to re-execute them for verifying the correct behavior of the system after some changes. Capture and playback tools usually work on a more abstract level of the system to increase portability and enable the definition of test scenarios, i.e. the recording of general browsing actions instead of mouse actions. Nevertheless, this does not guarantee that changing the structure of the GUI will leave the tests valid and reusable without any modifications [15]. As suggested in [13], there are many advantages and disadvantages of using this testing technique: a big advantage is that there are no major requirements of particular technical skills for performing this kind of testing, but at the same time this approach is not fully automatic, since it requires manual emulation (at least once). Another advantage of capture and playback approaches is that tests do not need to be written in advance, indeed they can be developed on the fly. However, they require the system to be stable in order to be testable, otherwise there could be the problem of having to redefine the same test multiple time, even for small changes.

Data Driven: The main idea of data driven testing is to use a capture/playback approach but to test the application with variable input and output. This approach leads to a problem caused by having inputs and outputs together within the test script, which means that every time data need to be updated the test script has to be regenerated or at least changed. This can cause problems especially if who is updating testing data is not aware of the test scripts. For huge systems embedding data into the test scripts is not a viable alternative, therefore they are typically read from another file input of the script. Hence, inputs and expected outputs are contained in an extern file, while the definition of the test script is contained in another file. The file containing the data, as in figure 6, is typically a table or spreadsheet or in a format that is easy for testers to be understood. Among those formats the most diffused are CSV (Comma Separated Values), TSV (Tab Separated Value), HTML tables and others [16]. As a consequence, there exist also a lot of tools for editing raw data in these formats. Another convenient and scalable approach to manage test data is to let the script read them
from a database. In this way, testing data can be shared across the whole organization, or the part of it dealing with testing activities.

![Data Driven data file](image)

**Figure 6: Data Driven data file [16]**

**Keyword Driven**: is based on data driven testing, hence the logic of the test script stays separately from the data. This test aims at alleviating one of the problems of data driven testing, that is some test cases might be very similar, but require in any case different test scripts. By means of the keyword driven approach data can contain also directives to provide some malleability to test scripts. In particular these directives, called keywords (see figure 7), affect the behavior of a test script by controlling the actions required case-by-case.

![Keyword Driven data file](image)

**Figure 7: Keyword Driven data file [16]**
2.4 Model Driven Engineering
When talking about automatic processes it has to be involved at least the notion of Model Driven Engineering (MDE). MDE is based on the principle of abstracting a real phenomenon, i.e. the system under study, in order to reduce the complexity of its development. Indeed, in a code-centric approach the quality of the final software application relies on programmers’ skills. In order to reduce costs and TTM improving the quality of the software, MDE introduced a new vision of software development that shifted the focus from coding to modeling. Therefore, the central idea of MDE is move towards a model-centric approach for realizing software systems. The first immediate benefit of working with models instead of code, is the reduction of complexity thanks to the representation of the system from a more abstract level. Moreover, the use of models discloses the opportunity to automatically generate implementation code from system design, through an automated procedure called model transformation. The MDE vision is based on the definition of three main entities: models, metamodels and transformations.

2.4.1 Models
A model represents an abstraction of the reality, in the sense that it represents those aspects that are relevant for the system under development. In this respect, an abstraction cannot represent all the details of the reality, but only those pertaining to the purpose a model is used for.

In the software engineering field, the most widespread modeling language is Unified Modeling Language (UML). UML offers a way of designing the system using some blueprints called diagrams or models that include activities, entities of the system, interactions among them, and many others. UML, as all the modeling languages, has the purpose to simplify the design process in software development. Usually it is possible to distinguish different kinds of models or diagrams depending on the perspective from which a the system is observed:

*Static Diagrams*: static diagrams point out the structure of the system using objects, attributes and operations on these objects. Static diagrams allow to divide the system in more separated entities that can be subsystems or simply basic entities of the system.

*Dynamic Diagrams*: dynamic diagrams allow to describe the system from a behavioral point of view. This is done specifying collaborations, and relations between the objects defined in the static diagrams. Through dynamic diagrams it is possible to specify how the system entities interact with each other and allow to simulate system’s scenarios specifying possible activities, states and sequence of actions.
2.4.2 Metamodels
In order to build-up a model, it is necessary to specify what aspects of a certain reality it is going to contain. In this respect a key concept is represented by the metamodel. In fact, a metamodel specifies a set of rules that identify how a model should be built for representing a certain real phenomenon. In other words, a metamodel prescribes what are the concepts and the relationships among them to describe a system. A model is said to conform to a metamodel when the former adheres to the rules the latter imposes. Given the flourishing of multiple technological solutions and frameworks for realizing the MDE vision, the Object Management Group (OMG) decided to define a standard framework for MDE. This standard encompasses the UML, and a minimal set of concepts exploited to define metamodels. This set of concepts is technically known as the meta-metamodel, and has been standardized under the Meta Object Facility (MOF) name [17]. The standard also carries by a well-defined metamodelling architecture, represented in figure 8, partitioned in four layers. The highest level, M3, represents MOF, i.e. the tool to build metamodels, which are located on level M2. An example of M2 metamodels is UML metamodel. Metamodels allow to specify models, which pertain to level M1. So in the case of UML M1 models are UML models like class diagrams, use case diagrams and so on. The last layer, M0, corresponds to the real object or system.

<table>
<thead>
<tr>
<th>Meta Level</th>
<th>MOF terms</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Metametamodel</td>
<td>MOF models</td>
</tr>
<tr>
<td>M2</td>
<td>Metametadata, metamodel</td>
<td>UML Metamodel, UML profiles</td>
</tr>
<tr>
<td>M1</td>
<td>Metadata, model</td>
<td>UML Models (e.g. Class diagrams)</td>
</tr>
<tr>
<td>M0</td>
<td>Data</td>
<td>Modeled systems</td>
</tr>
</tbody>
</table>

2.4.3 Transformations
The main principle behind MDE is to move from a code-centric to a model-centric approach. In this respect, it is of critical importance that models can be automatically manipulated by computers to generate other artifacts (at least code that should be derived by hand otherwise). Moreover, the growing adoption of MDE promoted the exploitation of models for other tasks, notably analysis and documentation. MDE prescribes the use of transformations as the mean to operate on models. In other words, a transformation is a program that converts a model, referred as the source, in another model, called target. It is worth noting that the mapping between the source and the target is not established at model level (M0), rather the elements of the source metamodel are mapped towards corresponding elements in the target metamodel. Hence the transformation is specified through a set of rules that operate a conversion from a source to a target model, both of them conforming to their respective metamodels. So the transformation guarantees that a legal input model is mapped towards an appropriate legal output. When the transformation operates only on a specific direction it is referred to as unidirectional transformation.
In some situations there could be the need to perform a mapping in both the directions, so from source to target and vice versa. Bidirectional transformations born with the idea of supporting such kind of needs. Notably, bidirectional transformations (BX) provide a way to ensure consistency between two models, so when one changes (being the source or the target) the other can be synchronized accordingly [18].
3 State Of The Art
This section contains a survey of the most common practices available in the literature about the core aspects of this thesis work. The objective is to give an overview of the CNLs field; in particular, I illustrate those works aiming at a better formalization of requirements for allowing the generation of consistent test cases through automation testing techniques. The intention of this survey is to illustrate existing solutions to the problems involved in the overall goal of this thesis, and hence to provide a general understanding of the issues and related countermeasures in mapping requirement specifications to test cases. In this respect, in the following I first introduce techniques for disambiguating requirements definitions that culminate with CNLs, and then I present available mechanisms for the provision of automation support to testing activities.

3.1 Requirements Ambiguity Avoidance
In defining requirements natural language is among the most common adopted solutions, mainly because it is fully understandable (even by non engineers/technicians), universal and flexible, and it gives to the requirements a high level of expressiveness. Unfortunately, NL is intrinsically ambiguous and requirements can be easily misunderstood if specified with it. This is a big problem especially considering that very often stakeholders are not aware of ambiguity in requirements definition [19]. To alleviate the ambiguity problem engineers proposed different solutions and approaches, based on the general goal of seeking the right tradeoff between higher degree of formality and expressiveness of the language. Although there are different tools and frameworks that help to remove ambiguity in requirements, there is no standard solution to this problem. The introduction of a Controlled Natural Language represents a solution in the direction mentioned above, that is having a more formal requirements definition approach while still keeping high expressiveness of the language.

3.1.1 Controlled Natural Language
One of the solutions proposed to remove ambiguity in requirements definition is the Controlled Natural Language (CNL). CNLs are a set of engineered languages based on vocabularies, expressions, grammatical constructions, and semantic interpretations, written in a natural language such as English [19]. They facilitate human to human communication helping in requirements analysis and documentation over the lifecycle, and human to machine communication specifying for example the interfaces between components or the APIs for a database connection. CNLs are derived from natural languages, therefore specifications can be written in an easy, readable, and understandable format such that they offer more expressiveness than other technical domain specific languages. According to Kuhn [20], a language is called CNL if it respects the following properties:

- It is based on one and only one natural language (basic language).
- It should be the restriction of its base language in terms of syntax and/or semantic.
• It should keep the natural properties of its base language, hence ensuring its usability, understandability also for stakeholders of different domains.

• It is a constructed language, which means that it is well defined on solid constructs always specified. This means that implicit definitions are not accepted, as well as natural processes.

CNLs are supposed in the one hand to solve issues derived from communication among stakeholders (speaking different native languages) providing a natural representation for formal specifications, and on the other hand to give an approach to translate a natural language in a language that machines can recognize, in order to trace requirements documentation. Moreover, a CNL can be useful even to write documentation, and as a medium in verbal discussions. Depending on the usage of the CNL, it can be domain specific, or general purpose, and according to the domain, it can be written in and for different contexts such as academia, industry, or government.

<table>
<thead>
<tr>
<th>Property</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comprehensibility</td>
<td>C</td>
</tr>
<tr>
<td>Translation</td>
<td>T</td>
</tr>
<tr>
<td>Formal Representation</td>
<td>F</td>
</tr>
<tr>
<td>Written Language</td>
<td>W</td>
</tr>
<tr>
<td>Spoken Language</td>
<td>S</td>
</tr>
<tr>
<td>Narrowed Domain</td>
<td>D</td>
</tr>
<tr>
<td>Generated from Academia</td>
<td>A</td>
</tr>
<tr>
<td>Generated from Industry</td>
<td>I</td>
</tr>
<tr>
<td>Generated from Government</td>
<td>G</td>
</tr>
</tbody>
</table>

*Table 1 Codes for CNL Properties*

Table 1, defined by Kuhn in [21], shows the properties of CNLs based on problems they address (comprehensibility hence for humans, or translation hence for automatic generation of artifacts), mode of usability (written or spoken), origins (academia, industry or government), and whether it is general purpose or domain specific CNL. However, these properties are not enough to describe a CNL: indeed, a CNL is based on a natural language, so all the properties of a natural language can be added to a CNL to enrich and characterize it. Furthermore, these properties are more oriented to the domain of application rather than to the language itself. In order to narrow down and shorten these properties a better classification, in which some properties are merged together, has been proposed. According with the authors in [20], CNLs can be characterized by means of the so called PENS classification schema, which is composed by four properties of a language:

*Precision:* it comprises the degree level of ambiguity, predictability and formality. NL does not assure these properties, mainly because in order to understand a sentence or a phrase, its
context has to be specified. On the contrary, formal languages enjoy higher precision, because their meaning is unique and well defined.

**Expressiveness:** it gives the range of expressions that is possible to reach using a language. In other words, it is the size of all the possible concepts that can be expressed using a certain CNL. PENS classification scheme uses the following five features:

1) Universal quantifiers
2) Relations of arity greater than 1
3) Structured rules
4) Negation
5) Second-order universal quantifiers

**Naturalness:** represents the degree of the look-and-feel and understandability of the CNL. In other words, it describes how much the CNL is close to its basic language.

**Simplicity:** represents the simplicity or complexity of a language taking into account its syntax and semantics. It can be considered as the effort needed to write the language syntax in a mathematical algorithm. As indicators of simplicity the PENS scheme can rely on the number of pages needed to describe the language itself.

PENS uses a natural language and a formal language to define a range of possible CNLs created by mixing together the two languages. Each CNL in the range is obtained by giving a class, from 1 to 5, to each of the properties in PENS. The five classes do not overlap each other because they are fully and properly defined. Table 2 contains on the x axis the PENS properties, and on the y axis the five classes. Referring to table 2, for example English language has class 1 for precision and simplicity (P\(^1\) and S\(^1\)), Propositional logic instead is in the opposite class for these two properties, being in P\(^5\) and S\(^5\). For expressiveness and naturalness it happens the contrary: English is in class 5 (E\(^5\) and S\(^5\)) whereas propositional logic is in class 1 (E\(^1\) and S\(^1\)). By following this example it is possible to classify all the CNLs on a scale containing S\(^5\) = 625 classes. Each class is represented by its formula given by the PENS properties classes. Always taking into account the previous example, English is P\(^1\)E\(^5\)N\(^5\)S\(^1\) [20].

<table>
<thead>
<tr>
<th>Precision</th>
<th>Expressiveness</th>
<th>Naturalness</th>
<th>Simplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Imprecise Languages: Vague and ambiguous sentences. Complex sentences are ambiguous.</td>
<td>Inexpressive languages: One or both the features 1) and 2) are missing. Propositional logic belongs to this category.</td>
<td>Unnatural Languages: Heavy use of symbols, characters, brackets, or unnatural keywords. Natural phrases are possible but not required.</td>
<td>Very complex: Have the complexity of NL. Cannot be described exactly.</td>
</tr>
</tbody>
</table>
Due to the lack of standardization caused by several frameworks with different application domains, the literature about CNL is quite huge and although this classification helps, it is still hard selecting the best CNL. In table 3 there are the PENS classes and properties for some well-known CNLs. Theoretically, there are 625 possible PENS classes, but of course some of them are unfeasible ($P^5E^5N^5S^5$) and some others are hard to represent. The idea in selecting a CNL is to look for a couple of the 4 properties identified as more representative and find a good tradeoff with the value of the other properties. Kuhn in [20] analyzed these different classes finding important relations: taking the couples precision-simplicity and expressiveness-naturalness it is easy to see that they are positively related. Instead, expressiveness and naturalness have a strong negative relation with precision and simplicity. These values are consistent with preliminary expectations: a more expressive language is in general less precise but simpler and more natural. Vice versa, a more precise language will have its expressiveness reduced together with its naturalness and simplicity.

<table>
<thead>
<tr>
<th>Class</th>
<th>Properties</th>
<th>Controlled Natural Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Less imprecise:</td>
<td>Low Expressiveness:</td>
<td>Dominant Unnatural:</td>
</tr>
<tr>
<td></td>
<td>Ambiguity and vagueness lower than in NL. Still not much formal for automation.</td>
<td>Have both the features 1) and 2). Description logics belong to this category.</td>
</tr>
<tr>
<td>3 Reliably Interpretable:</td>
<td>Medium Expressiveness:</td>
<td>Dominant Natural:</td>
</tr>
<tr>
<td>Syntax heavily restricted. Not necessarily formally defined.</td>
<td>Have all of the features 1), 2), 3), and a&gt;$Z54). First-order logic belongs to this category.</td>
<td>Natural elements are dominant. Still presence of unnatural elements.</td>
</tr>
<tr>
<td>4 Deterministically Interpretable:</td>
<td>High Expressiveness:</td>
<td>Natural Sentences:</td>
</tr>
<tr>
<td>Full formal syntax Deterministically parsed to formal models</td>
<td>Have all features 1), 2), 3), 4), and 5), but are not ES-languages. Second-order calculus belongs to this category.</td>
<td>Speakers of the NL understand statements. Minor exceptions as text color, indentation, hyphenation, and capitalization.</td>
</tr>
<tr>
<td>5 Fixed Semantics:</td>
<td>Maximal Expressiveness:</td>
<td>Natural Texts:</td>
</tr>
<tr>
<td>Fully formal and specified. Phases have exactly one meaning and can be automatically derived</td>
<td>Can express anything that can be communicated between two humans. All natural languages belong to this category.</td>
<td>Documents are written in a natural style. Dialogs have a natural flow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Very Short Described:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exact and comprehensive description requires exactly 1 page.</td>
</tr>
</tbody>
</table>

Table 2: PENS classes
<table>
<thead>
<tr>
<th>P: E</th>
<th>N: S</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>C S D G</td>
<td>CAA Phraseology, FAA Phraseology, ICAO Phraseology, PoliceSpeak, SEASPEAK</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C W D I</td>
<td>Airbus Warning Language</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F W A</td>
<td>AIDA</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C T W D A I</td>
<td>ALCOGRAM, COGRAM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C T W D A</td>
<td>CLCM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C T W D I</td>
<td>ASD-STE, Bull GE, CTE, CASL, CE at Douglas, DCE, General Motors GE, PACE, Sun Proof</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C T W D</td>
<td>Wycliffe Associates’ EasyEnglish</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C T W I</td>
<td>ICE, SMART Controlled English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C W D I</td>
<td>AECMA-SE, CFE, CASE, CE at Clark, CE at IBM, CE at Rockwell, EE, HELP, ILSAM, KISL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C W D G</td>
<td>Massachusetts Legislative Drafting Language</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C W I</td>
<td>Boeing Technical English, NSE, SMART Plain English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C W</td>
<td>Basic English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T W D I</td>
<td>MCE, Oce Controlled English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T W A</td>
<td>KCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T W I</td>
<td>CLOUT</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C F W D I</td>
<td>SLANG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F S D I</td>
<td>Voice Actions</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>F W D A</td>
<td>RNLS</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F W A</td>
<td>ClearTalk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W I</td>
<td>ITA CE</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>F W I</td>
<td>CPL</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>C F W I</td>
<td>RuleSpeak, SBVR-SE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>F W D A</td>
<td>Drafter Language, MILE Query Language</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>F W A</td>
<td>Quelo Controlled English</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>T F D A</td>
<td>PILLS Language</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>F W D A</td>
<td>Atomate Language</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>Gellish English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>GINO’S Guided English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W I</td>
<td>CELT</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>F W D A</td>
<td>PROSPER CE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>ACE</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>F W D A</td>
<td>ICONOCLAST Language</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>F W D A</td>
<td>CLEF Query Language</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>F W A</td>
<td>Ginseng’s Guided English</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>F W D A</td>
<td>Coral’s Controlled English</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>PathOnt CNL</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>F W A</td>
<td>Sowa’s syllogisms</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>F W D A</td>
<td>TBNLS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>OWLPath’s Guided English, SQUALL</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>F W A</td>
<td>CPE, CLIP, OWL ACE, SOS</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>F W D A</td>
<td>BioQuery-CNL, PERMIS CNL, ucsCNL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>CLonE, DL-English, E2V, Lite Natural Language, OSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W G</td>
<td>Rabbit</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>F W D A</td>
<td>CLM, ForTheL, Naproche CNL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F W A</td>
<td>CLCE, PNL</td>
<td></td>
</tr>
</tbody>
</table>
### 3.1.2 CNL Frameworks

This subsection illustrates some CNLs selected from table 3 according to the purpose of this thesis work, hence based on the following criteria:

- The language has to be suitable for machine translation, so it has to have a high precision according to PENS (greater than P\textsuperscript{2}).
- The language has to have a good level of expressiveness and naturalness because it will be used for writing down requirements. Hence according to PENS classes not less than E\textsuperscript{3} and N\textsuperscript{3}.
- The language has to be used for written purposes.

#### Attempto Controlled English Framework

The Attempto Controlled English (ACE) is a CNL proposed by Fuchs and Schwitter in [22], which according to PENS classification is a P\textsuperscript{4}E\textsuperscript{3}N\textsuperscript{4}S\textsuperscript{3} CNL. Attempto is meant to be used to interactively formulate requirements specifications, and it can be interpreted by a computer still offering a good level of expressiveness and naturalness. Writing ACE requirements does not presuppose expertise but offers to domain specialists an application specific language that “breaks the bottleneck between informal and formal specification” [22]. The specifications written using ACE are text based and give the impression of being informal although the language is machine interpretable. Indeed, ACE accepts textual specifications and translates them either in Prolog or in an understandable representation structure. Attempto parser generates a syntax tree as syntactic representation and at the same time a Disclosure Representation Structure (DRS) as a semantic representation. In order to inform the user about the result of the analysis the parser generates a paraphrase in ACE explaining how the language has been interpreted by Attempto. In this way, the user can accept or rephrase the input to achieve the desired interpretation results. As already discussed in this thesis, there is also the need of a preselected set of words, or vocabulary, to use in order to remove ambiguity and to have a visual style for the communication with stakeholders. Attempto’s vocabulary contains a set of predefined words such as determiners, prepositions, pronouns and conjunctions and a set of domain specific words such as nouns, verbs, adverbs that are added based on user’s needs. In this way, stakeholders can gradually add the needed words to the vocabulary while having at the same time a basic set of words at disposal. The base vocabulary allows to build templates typically suitable for stakeholders without a domain specific knowledge. For what concerns domain specific words, there are three different sets:
• **Nouns**: is itself a set of three subclasses: common, proper and personal nouns. Common nouns contains countable or mass nouns. Countable nouns can be referred with undefined articles or quantity as a number.

• **Verbs**: are the most important words in ACE denoting states and events. Verbs can be used only in an active form and with the present tense. Many advantages are achieved introducing these restrictions: the active forms allow to make casual agents explicit, having only the present tense stakeholders can write chronological events or states only as a list of step-by-step instructions avoiding complexity of temporal references. Modal verbs are forbidden because they can give a false level of vagueness about certain facts depending on the interpretation either of the writer or reader. One word verbs are strictly preferred on the one constructed with the use of particles, indeed particles can overlap prepositions affecting negatively the meaning of a sentence.

• **Adjectives**: semantically they represent properties. The only degree of comparison in ACE is given by the two adverbs *more* and *most*. Modal adjectives are not preferred: *possible, probable, certain, sure* and *necessary* do not add a concrete contribution to the text, rather they give ambiguous dimensions to be avoided in requirements definition.

For what concerns grammar structure of the specifications, the pattern to follow is the *declarative sentence* that is characterized by two aspects: a propositional content and an illocutionary force [22]. Specification texts consist of:

• Declarative sentences: *subject* + *finite verbs* + *complement* or *object*.
• Composite sentences: obtained as sums of declaratives coordinated using *and, or, either-or*, subordinated using *if-then-else, who/which/that*, negated using *nor, neither*.

Sentences can contain:
• subject and object modifying relative sentences
• anaphoric references, e.g. personal pronouns
• coordination between equal constituents, e.g. *and, or*
• ellipsis as reduction of coordination
• negated noun phrases, *no X*
• synonyms and abbreviations

Interrogatives can contain:
• *yes/no* questions
• *wh*-questions

For what concerns rules of how to use this vocabulary, ACE uses different methods to fight structural ambiguity. The language itself does not accept some ambiguous sentences, or it provides unambiguous alternatives. Another mechanism to try to remove ambiguity is realized through a small number of principles associated with syntactic constructions: all the sentences are parsed according to these principles in order to remove ambiguity. As a result, a paraphrase is generated showing all the modified ambiguous sentences. Based on the results, the users can modify the initial sentence in order to re-parse it and trying to obtain a better interpretation [22].
**The Ontograph Notation**

According to Kuhn in [23] the existing approaches to evaluate and differentiate a CNL can be grouped into two categories: task-based and paraphrase-based approaches. The former ones are based on creating a CNL, on a specified task, and emulating this CNL into a given tool. This approach has different problems: first, it tests more the ability to write CNLs rather than the ability to understand its statements, and second it is really hard to understand if the knowledge generated is a result due to the tool or to the CNL itself. The paraphrase-based approach deals with the automatic analysis and comprehension of NL in a requirements specification. The analysis results are then exploited to derive an appropriate CNL. The problems related to this approach are mainly due to the intrinsic ambiguity of NL: if any of the sentences is misunderstood then the transformation towards the CNL will create erroneous support to requirements definition. Moreover, some of the ambiguities carried by the NL specification could be propagated in the CNL itself if not adequately detected.

In order to overcome these problems Kuhn in [23] proposed a solution called Ontograph notation ("a contraction of ontology and graphs"). This notation is based on a graphical notation and is very easy and intuitive. A good advantage of this notation is that it is completely independent from tools. In the ontograph notation each object is created together with a graphical representation of it. A legend introduces the name and the graphical representation of the type and its relations. Objects can then be extended into different types, belonging to the same main class. By looking at figure 9 for example, a person can be extended by traveler or officer. Objects of the same base class can also be combined in new objects. For example, in figure 9 Kate is an officer and a traveler at the same time. As it is possible to see in the figure, objects can have also a name. Relations are represented by circles containing different symbols inside.

![Figure 9: Ontograph example [23]](image-url)
There is no notation to express negation. It can be implicitly expressed by not representing the fact to be negated. Everything that is true is shown in the ontograph. The ontograph has different benefits. The first one is that nothing can be left to the user interpretation because everything defined in it is necessarily true or false. Another fundamental aspect of this notation is that it is not limited on its capability. For example, it is possible to express concepts like “every man loves Mary” in a vague and general way. The only way to figure this out is to represent a love relation between all the instances of the class man towards Mary. This means that every feasible relation has to be expressed through a direct relation. On the other hand, it is not possible to express concepts on an infinite set and it is not practical with sets of huge numbers. So this kind of representation is suitable for easy and relatively small specifications because of its simplicity, but not for specifications that are complicated and in which forbidden concepts, such as negations, have to be expressed specifically. At the same time, it is a representation language very useful for little projects because of its ease to use and steep learning curve.

3.2 Testing Automation
Testing is the activity to be performed in order to validate that the software system conforms to the requirements specification. The testing process involves different activities, among them planning, execution of tests, validation of the results, and debugging. As mentioned so far, testing is also a very expensive activity that can cost up to 50-70 % of the total amount of the whole software development [24]. A possible solution to alleviate the cost of testing is represented by automating the testing process. As mentioned in [24], automation is not a short-term solution because it requires full development efforts during a project involving definition of strategy and goals, definition of requirements evaluating different solutions and implementation. Hence, being a time and cost expensive activity itself, testing automation requires solid bases to be developed. The automation process examined starts from requirements specification, and can be reflected either in the generation of test cases or in the generation of test scripts. With respect to the generation of test cases, in the following two widespread approaches are illustrated, namely ModelBased and Scenario Based testing.

3.2.1 Model-Based Testing
Model-based testing (MBT) aims to demonstrate and figure out system’s defects through explicit behavior models. Test cases are generated according to these models describing the structure of input and expected output, while generated test cases are then transformed into test scripts that will be run. The process of model-based testing is reported in figure 10.
Requirements collected in the requirements specification are translated in models (see step (1) in the figure). The model, called also test model, represents an abstraction of the system (or of its subparts), built trying to be as more independent as possible from the development model in order to not propagate possible errors contained in the development model [26].

The second step (2 in the picture) is test selection criteria, and is related to a definition of a test generation strategy in order to achieve defined objectives for a system. In other words a selection criteria can be related to a given functionality of the system, to the structure of the test model, to coverage heuristics, or to a set of faults [25]. In general, choosing the best criteria is not possible [26]. The following are the most commonly used criteria:

**Structural Model Coverage Criteria:** it exploits the structure of the model, i.e. nodes and arcs of a graph model or conditions in a pre/post condition model. Depending on the notation used, there are specific kinds of coverage criteria. For instance, in a pre-post condition, some coverage criteria that are commonly used are: cause-effect coverage, and coverage of all disjoints in the post condition [25]. Some approach of structural model coverage criteria operates on nodes, transitions, pairs, cycles and so on. This criterion is particular useful for designing tests for the so called boolean decision models where statements are given in a flow that can be traversed only for certain vectors of particular inputs verifying branches conditions.

**Data Coverage Criteria:** these criteria try to minimize the number of inputs to select in order to test a large data range. The main idea is to categorize the data space into equivalence classes, where equivalence means significant to reveal faults. Pairwise and N-way coverage criteria are popular forms of data coverage criteria [26].
Requirements-Based Coverage Criteria: is the most interesting kind of testing for what concerns traceability of requirements in test cases, and is based on the idea of mapping requirements specification to elements in a model. In this way, textual specifications can be completely traced into corresponding models from which it is possible to generate test cases. For instance, let us consider a behavior requirement represented in an UML state machine model. If the derived test cases covered all the possible transitions in the state machine, then this would assure by-construction that all the requirements have been covered.

Ad-hoc Test Case Specifications: these are the criteria specified ad hoc for some particular cases, especially designed to cover boundary behaviors different from the common ones. The notation used for representing them is usually UML sequence diagrams or Markov chains. This category of coverage criteria is related to the scenario-based testing approach where test cases are generated from descriptions of abstract scenarios [26].

Random and Stochastic Criteria: these are the approaches in which the testing paths are modelled taking into account the probability of some action to be accessed in decisional branches. The generated test cases are then collected and used as profiles.

Fault-based Criteria: these criteria are mainly applicable to SUT models, since the goal of testing is to find faults in the SUT. One of the most common fault-based criteria is mutation coverage, an approach consisting in changing the model, and executing correct models and mutated models in order to find a correlation among faults common to them.

Step 3, consists in a transformation of the selected criteria into a test case specification model. The specification model represents the metamodel the test cases should conform to in order to be valid. In other words, a test case specification is a high level description of test cases specified in the selection criteria. Automating the transformation means to provide an automatic generator that given a model of requirements and a test specification, is able to generate the corresponding test cases.

Going ahead with the description of the steps, step 4 consist in running, given a model of a requirement and a test specification, the generation of corresponding test cases; hence, to transform a couple test case specification-model into a set of test cases specification. There are different strategies for the generation of test cases: Random generation: test is implemented running the system with different inputs selected from an analysis of the input space of the system. The results are then compared with the estimation made by the engineers. Model Checking: is based on the idea of verifying or falsifying the properties of a system. The test cases are generated according to the events defined as conditional statements on class of properties. Search-based algorithms: are the one in which tests are performed considering the flow of the test cases as a graph. Different strategies are developed for this testing techniques based on the desired coverage.

Finally, step 5 consists in running the test cases. The execution can be performed either manually or automatically by a testing framework providing support to run and record the operations in a test suite.
3.2.2 Scenario Based Testing

One of the most common used approaches in requirements analysis is the creation of use case scenarios as a specification of the system. UML use cases have become a standard way of specifying requirements as a collection of actions performed by actors of the system.

In [33] the authors propose UMGAR, a technique to assist engineers in the generation of UML models from a software specification written in NL. UMGAR is able to decompose complex structured sentences into simple sentences in a way that it is possible to preserve most of the information contained in the requirements specification. There are eight syntactic reconstructing rules that have been implemented in UMGAR. UMGAR parses each period to see if the specified requirement matches a statement sentence specified as “Subject: Predicate” or “Subject: Predicate: Object”. Subject and object represent a noun phrase, and a predicate as verb phrase. Requirements written in NL are parsed from tree different parsers: Standoford Parser: for context free grammars can recognize actors, use cases, classes, methods, associations and attributes from the specification, WordNet2.1 performs conversions from plurals to singulars, JavaRAP resolves pronouns in the specification. After parsed, the specification is given as input to a Use-case Model Developer that from normalized statements identifies actors, predicates, and associations, of the system. Thereafter, Analysis Class Model Developer parses requirements in order to do analysis class model and it classifies them according to a tree kinds of word: redundant and irrelevant words, attributes, and adjectives. After that a component called Design class model developer generates collaboration diagrams based on the classified words. UMGAR generates a collaboration diagram for each use case specification. Each rule should be structured specifying “who is doing what to whom”, according to the rules:

- Subject (NP) in the sentence is considered as sender object.
- Object (NP) is considered as receiver object. And Predicate (VP) can contain noun phrase, which might be considered as receiver object based on the VP structures.
- The verb phrase between subject and object is taken as message passed between objects.
- If a sentence has subject and predicate, without any object, then sequence stated in the use-case specification helps to find out the relation between both messages.
- Conditional statements represent two statements consecutively: it is managed by putting If clause at the beginning of the period and an end_If clause at the end of the sentence.
- Concurrent statements represent sequence of actions performed in parallel, and are managed by putting Start_Concurrent clause at the beginning and End_Concurrent clause at the end of concurrent statements.
- Iterative statements are managed putting Start_While statement at the beginning and End_While at the end of the period.
• Synchronization statements are managed inserting Start_Sync word after the first sentence to show the synchronous message started and End_Sync after the last sentence.

After generating the OO models, in order to maintain consistency in the models, and trace them from requirements specification, UMGAR implements an approach called KeyWord in Context (KWIC) that allows to trace requirements through the keywords, mapping them to the specific requirements showed into the system.

3.3 Related work

This thesis work deals with the automated interconnection of requirement specifications to corresponding test cases. In this respect, solutions available in the state of the art typically do not cover the problem as a whole: they cope either with the improvement of requirement definitions, or with the generation of test cases based on a well-defined model. The former issue and corresponding solutions have been illustrated in the previous sections. It is worth to remark here that CNLs rely on their own template, rules, vocabulary, which in turn are tailored to a specific domain taken into account, especially for generation of artifacts. This custom creation of CNLs hides general patterns for an appropriate definition of a certain CNL, in particular when having in mind a precise generation goal. This brought to the identification of best practices in both the requirement phase and the test cases definition in order to build a general concept applicable regardless of the domain.

The works reported below are examples of automation testing techniques while the one introduced in the subsection 3.1 refer to CNLs work. Indeed, the proposal illustrated in this thesis work is a general approach based on the introduction of a CNL as a solution to reduce the consistency and temporal gap between requirements and tests.

In [34] the authors propose Testing Object-oriented systems with the unified Modeling language (TOTEM) in order to automate generation of test cases (and scripts). TOTEM is a framework used on object-oriented software development and is based on UML standard models like use case, activity, sequence and class diagrams. The process starts with the formalization of use cases in textual and model-based notation. After that activity diagrams are generated to show the dependencies and interaction between the defined use cases. In addition, sequence diagrams are created defining more precisely what will be designed according to the object-oriented implementation. At this point, test cases are generated as regular expressions from requirements modeled as sequence diagrams [34].

The approach proposed in [35] is almost the same as this one. It has been proposed for Siemens medical project, and applied to several projects. It consists of using UML models to describe scenarios, the use case is supported from activity diagrams, sequence diagrams and data variation equivalence classes. Use cases are specified on a test point of view specifying expected input and output. Use cases are used in two phases: in the first phase they represent an abstract view of the system but later on in the process, after defined activity, class, sequence and package diagram, use cases can be populated with enough data to make them
testable. A big advantage is that use cases following the whole process, can assure consistency of requirements specifications, and if used as test cases it is possible to avoid loss of information [35].

Another important work is the one proposed in [36] consisting in a generation of tests from textual requirements. This approach is based on the idea that in the beginning of requirements definition, all the requirements are written in NL. NAT2TESTIMR is an approach to generate test cases from requirements described in CNL, based on the RTTester3 Internal Model Representation [36]. Initially, the approach parses the textual requirements to evaluate their conformance with the SysReq-CNL. After that, the approach provides a semantic interpretation for the requirements, using verb case frames as semantic representation. From the case frames, the requirements’ semantics are mapped into an internal model representation and used to generate test vectors with the support of the RTTester and its SMT solver. The whole process is automated by supporting frameworks. The tests generated by NAT2TESTIMR are an example of domain specific language that support simulation of models at design level [36].
4 Concept

This work aims at introducing a general solution to the problem of enhancing consistency between requirements specification and test cases through the definition of a CNL. The underlying goals are on the one hand to remove ambiguity in the requirements specification, and on the other hand to automatically generate test cases thus guaranteeing consistency between requirements and tests. This should be done in the most general way in order to provide a mechanism that can be reused in any kind of project, regardless of the application domain. This work can be seen as research glue between the different processes reported in figure 11, in order to give a scalable and sustainable approach to the automatic generation of test cases (which then can be transformed in test scripts) starting early in the development lifecycle process. As it is possible to see in the figure 11, the process starts (step 1) with the definition of requirements written in NL. Hence this approach is valid for both development from scratch and migration of legacy systems (for which requirements were only specified as NL).

![Figure 11: Concept approach from specification written in NL to test cases (or test scripts)](image)

Requirements are then automatically parsed and manually restructured in a more consistent specification (1.1). According to the criteria chosen by engineers (step 2.0), a CNL and a dictionary are generated starting from the structured requirements. Next step (3.0) consists in the definition of test cases criteria, allowing to establish a formal mapping between test cases model and CNL models (step 3). At this point, requirements written accordingly to the CNL can be translated into corresponding test cases.

4.1 Natural Language Requirement Specification

NL is the most widespread approach to write requirements specifications, mainly due to its expressiveness. In fact, using NL allows to express requirements in a way that all the stakeholders involved in requirements analysis are able to understand [5]. However, as stated so far one drawback of such expressiveness is that it is not easy to avoid ambiguities in writing requirements. If ambiguities are not detected they can be propagated to subsequent development stages, where discovering misunderstandings is by far more expensive. In order
to overcome this problem, many approaches have been proposed as trying to remove ambiguity in a NL specification through different frameworks and tools. However, as mentioned before, most of the tools are domain specific, and hence based on a level of formality proper of the domain itself. Moreover, due to the different domains of application, usually supporting tools and frameworks are developed for different purposes and with different schemas depending on the application domain.

4.2 Definition of a CNL and a Vocabulary

Regardless of the domain, a CNL needs a set of predefined rules that restrict natural language usage, embedded in a template, and a set of words collected in the so called dictionary. Hence the first step (1) with respect to the figure 11, is to parse the requirements written in NL in order to find relevant words to create the dictionary. This parser should consist at least of three components:

1. A component that collects all the words contained in the requirements specification, counting them, and giving as a result the number of different words with corresponding occurrences.

2. A component that searches for synonyms and records words correspondences so that test engineers can evaluate them and remove ambiguities in the terms and, based on the results obtained, propose improvements [28].

3. Ideally, the third component should examine and classify the dictionary words in a syntactic way to determine what a CNL may require as verbs, nous and so on. The results obtained parsing the initial specification should be manually examined in order to find common aspects and to gain some knowledge that can be useful later on in the definition of rules and templates needed from creating a CNL [28].

The vocabulary is the preliminary step (see step (2) in figure 11) for the definition of a CNL and can be obtained automatically through the use of the parser. The only design decision that requirements engineers need to take is on the eliminations of synonymous, and the use of some words in place of some others for the definition of the concepts. The engineers should come out with an analysis of the words, or in other terms with a classification of the words that are: most used, less used, forbidden, preferable, and conceptually meaningful [19]. It is worth noting that this analysis effort can be useful over the whole development process, since as explained later, a semantic parser will be created as relying on the meaningful words, forbidden and most used words will entail the creation of a template and rules for the CNL, while less used words will be eliminated and replaced (if possible) with more used ones.

For the definition of the CNL more design decisions are involved (step 3.0): the engineers should define first the purpose of the CNL (i.e. if it will be used for communicating with stakeholders or to automate the generation of artifacts). If, analogously to this work, the CNL is going to be defined for generation of artifacts, an important design decision involves the level of formality that the CNL should have (i.e. if the CNL has to be mathematicaloriented or NL-oriented). This decision will be then reflected on the kind of artifacts the CNL should
generate: in particular, as specified later in the text, the level of abstraction and precision of the artifacts generated from a CNL strictly depends on the way the CNL is defined, on its template, rules, and dictionary. Although between the CNL and test cases in the figure 11 there is a parser, it is clear that the more the CNL is permissive the less is possible to map it to precise test cases. In other words, the more a CNL is formal and restrictive, the more it is possible to generate concrete test cases. However this will be taken in account later on in this work.

4.2.1 Tradeoffs in CNLs

![Figure 12: Relation between PENS properties. White dots represent natural languages, black dots represent formal languages and blue dots represent CNLs (or CNLs classes). (20)](image)

Depending on the PENS classification in [20] there are four main characteristics, measured within a range from 1 to 5, that define a CNL: Precision, Expressiveness, Naturalness, and Simplicity. Figure 12 shows the relationships between these characteristics, where the black dots represent formal languages, white dots represent natural languages and blue dots represent different CNLs or CNLs classes. The relationships have been illustrated by Kuhn in a CNL classification survey that considered more than 100 CNLs. In particular, expressiveness and naturalness are directly proportional as well as precision and simplicity. Naturalness and expressiveness are instead inversely proportional to precision and simplicity. In figure 12 the black dots represent formal languages, white dots represent natural languages and blue dots represent different CNLs or CNLs classes.
Based on the relations among CNL properties it is possible to give a common schema to select (or implement) a CNL based on the problem under study. It is possible to see, according to Kuhn survey from table 4, that the CNLs available for translation, that is artifact generation in general, are 22. Among these 22 CNLs, 17 are comprehensible, with respect to the comprehensibility definition given in subsection 3, 17 are domain-specific, and 18 are from industrial domains. According to the definition given above, it I possible to see also that the all the implications done before about the PENS properties are in turn true. This can allow to make some implications on CNLs used for translation purposes: first of all taking into account the relation between comprehensibility, naturalness, and expressiveness it is easy to understand that the various proposed CNLs for translation are not so strict and formal for what concerns grammar restrictions and rules applied to the syntactic composition of the phrases. As a consequence, CNLs used for translation are much more similar to a NL than to formal languages: from this, it is implicitly derivable that these CNLs are used also for communicating with the stakeholders of the system. Another implication can be done noticing that the CNLs are mostly domain-specific: this means that it is possible to reduce the use of words more than in a general purpose language, so that the dictionary can be composed of words having a semantic interpretation that can be useful in the automatic generation of artifacts. A very important aspect to notice is that domain specific CNLs represent the 80% of all the CNLs developed for automatic generation of artifacts, implicitly testifying the difficulty in providing a general solution for the problem tackled in this work.

Going ahead in the analysis, and taking into account the relation precision-simplicity, it is possible to notice low values of them when dealing with automatic translation objectives. This means that generation of artifacts from CNL relies on the use of a well-defined template, since it has to be supported the translation from a language close to NL. In order to be well-defined, the template has to be based on a formal representation, typically a metamodel defining entities and their attributes. Since the template should also guarantee comprehensibility, it has to be based on natural constructs coming from the NL that is underlying the creation of the CNL. In other words, a template assuring comprehensibility should be based on all those natural constructs that can allow to write requirements with naturalness. For example, a CNL

<table>
<thead>
<tr>
<th>property</th>
<th>total</th>
<th>combined with property</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>C comprehensibility</td>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>T translation</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>F formal representation</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>W written</td>
<td>93</td>
<td>40</td>
</tr>
<tr>
<td>S spoken</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>D domain-specific</td>
<td>53</td>
<td>33</td>
</tr>
<tr>
<td>A academia</td>
<td>50</td>
<td>4</td>
</tr>
<tr>
<td>I industry</td>
<td>43</td>
<td>33</td>
</tr>
<tr>
<td>G government</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4: existing CNLs classified by Kuhn in [20] according with their properties
for composing and delimitating periods should rely on punctuation rather than on brackets or special characters [19].

4.3 Test Case Generation

Summarizing step 1, 2 and 3 with respect to figure 11, the following artifacts have to be defined:

- A parser for requirements written in NL that allows to avoid ambiguity comparing the words in the specification looking for synonyms and where possible removing those synonyms. As specified, the process of ambiguity removal might be automated, however the practical experience suggests to adopt the manual solution [29].

- A dictionary of all the words as output of the parser, that is then analyzed by requirements engineers to reduce the number of words, and to produce a set of most significant words useful for step 3 in figure 11 (that is the derivation of a semantic parser) as described below.

- A template, i.e. a formal specification of how to write phrases. This artifact identifies the metamodel on which the transformation to test cases is based. A CNL that is the synthesis of dictionary, template, and set of rules.

These artifacts collected together will be useful for the implementation of a parser able to generate test cases (step 4). Before being able of defining a transformation, a template for writing test cases should be provided. Providing a general template or guidelines for designing a template for test cases is not in the scope of this thesis work, because it strictly depends on the purpose of test cases: the template is totally different if test cases are generated to be then translated manually into test scripts (so in this case they depend on the testing engineers’ preferences), or if they will be automatically translated into test scripts (and in this case the template must depend also on the type of test script). Therefore, in this direction there is full freedom left to the testers and requirements engineers that can develop independently a template.

The next step after defining a template for test cases is to define the parser. A transformation (or parser) generating test cases from requirements shall take as input the CNL and test cases metamodels, together with the requirements written according to the defined CNL, and will generate as output a test case for each requirement conforming to the test case metamodel. The characteristics of the parser strictly depend on the level of formality that can be assured in writing requirements according to the defined CNL. Indeed, having a huge number of different entities defined in the requirement’s metamodel allows to map more concepts in the transformation. As a consequence, the test cases can be made more detailed with respect to the abstraction level at which the system is represented in the requirements specification. In the other way around, if the requirement’s metamodel does not contain a semantic interpretation of the words in the dictionary, the transformation deriving test cases will be limited to a re-elaboration of the contents in the requirements specification, that is without any deeper detail with respect to the initial specification.
Despite on the one hand having the same information in a different template can be perceived as a limit, on the other hand it carries by the advantage of keeping the consistency between requirements and test cases by construction. In fact, the amount of information remains constant, even if organized in a different template. However, it will only be possible to generate abstract test cases that will have to be manually analyzed and transformed into corresponding executable test scripts by engineers. It is worth noting that this is a direct consequence of the approach chosen for defining the CNL. As a matter of fact, such a decision hinders the definition of a semantic parser given that the successive step of automation (so from test cases to test scripts) is not realizable or too expensive in terms of money and time.

The other possibility is to produce a “smarter” transformation considering a possible map of the words contained in the dictionary (at least the most significant found in step 3) with corresponding artifacts, such that it would be possible to automatically produce more knowledge than what contained in the requirements specification. The development of such a mapping is trickier than the previous, since typically cannot be established in general terms. Notably, requirements should be categorized according to their purposes, being functional, non-functional, and so on, as specified in [32], and for each of the categories a test cases template should be provided. Moreover, the dictionary should be categorized based on the previous classification: for example, one word that in a functional specification view can mean something, in a system level perspective specification can mean something totally different. By defining class of different words, it is possible to give a mapping that is based on large sets, instead of single words, building so an approach that support upgrading an future modification. By following this approach, it is clear that the transformation can be made more and more specific depending on the amount of the entities defined in the metamodels, the different kind of requirements for which it is desirable to generate test cases, the amount of significant words, and so on.

Summarizing the discussion done so far, step 4 in figure 11 entails the creation of:

- A template (or a set depending on how many different kinds of requirements are needed for generation of test cases) representing the structure that the test cases should have. This can be done specifying, through a metamodel, the different entities involved in the mapping with corresponding CNL entities.
- A mapping between the words identified as meaningful in the definition of the CNL (step 3) and entities defined in the template itself (i.e. when a requirement contains a reference, specified using a meaningful verb of the dictionary, to another requirement)
- A parser (or transformation) that takes as input the requirements and generates as output appropriate test cases. Being more precise the transformation should take as input the template of CNL, the template of test case, and the set of meaningful words, and then execute the rules for the generation of test cases.

After this phase test cases should be formalized and at this point there are two possible ways: either to automatically generate test scripts from test cases, or to do it manually. What it has to be clear is that defining an automatic process involves a lot of effort, this means that the
process of automation should be done only if the company can make reuse of it (i.e. in more than one project) or use it in large scale projects. In any case, as stated so far, there is no general approach to develop such an automation support. This is mainly due to the nature of the produced artifacts, i.e. test scripts, that depending on the setting they are used in may require different frameworks, specific formats, and other domain-specific adaptations to be run over specific testing environments. As a very general guideline, it is expectable that the words of the dictionary should me mapped towards constructs of the specified output format of test scripts.
5 Case Study

This section aims to better clarify the application of the concept discussed in the previous section by giving a demonstrative example. In particular, I illustrate the requirements specification coming from a previous real-life work, and more precisely pertaining a project done for the Distributed Software Development (DSD) course at MDH. The requirements, listed in the following, are devoted to the realization of an online booking system that has three different users: a customer who wants to book a flight simulator, a coordinator that has to confirm, edit, or delete bookings, and an administrator that is responsible for accounting.

The remainder of the section has been divided into subsection according to the various phases introduced in the concept with respect to figure 11. For each of the phases, I discuss how the proposed ideas reflect to concrete solutions: analysis of requirements given in NL, derivation of a CNL, choice of test cases template, and eventually automated support for mapping requirements towards the CNL and from this latter to corresponding test cases. Therefore, taken as a whole, the case study demonstrates how the concept can be successfully used for creating the support of test cases generation based on the definition of requirements in NL. I eventually analyze in retrospective some of the test scripts developed in the original project as a form of validation of the idea behind the concept. This analysis demonstrates that generated test cases have concrete correspondence with the tests created by hand.

5.1 Requirements in NL

The online booking system represents a real-life project. Therefore, as expected the requirements specification coming from the first interview with the customer carries by problems related to ambiguity and lacks on consistency. Those requirements appeared as follows:

**Functional requirement #1:**

A customer should be able to book a simulator from the simulator page where it should be possible to check the availability of the simulator. If the simulator is not available for the selected timeslot, the system should prohibit to book. If the simulator is available, the user should provide data: email, name, Phone number, address, and additional information. Then the customer can send the applied data to the system that checks the data and if corrects notifies the customer and the coordinator through an email containing booking data, else ask the customer to reenter the data.

**Functional requirement 2:**

When the customer has booked a simulator, it should be possible to delete this booking from a page. This should be possible entering email and booking ID of the booking. The booking ID is in the email sent after the booking. The system notifies to the user that the booking has been canceled.
**Functional requirement #3:**
When a user has booked a slot, a coordinator can change the data of that booking. The system will validate the edited data. If valid the system will then notify the customer, otherwise asks to reenter the data.

**Functional requirement #4:**
When a user has booked a slot, a coordinator should assign an instructor to a booking selecting him/her from a list of provided instructor. Then the coordinator should confirm the booking. After the booking is confirmed the system should notify the customer and the instructor.

**Functional requirement #5:**
When a user has booked a slot, a coordinator can delete the booking. Afterwards the system will notify the user.

**Functional requirement #6:**
An administrator should be able to create an account specifying name email address and role. The system verifies the data and if valid send a notification to the email account, otherwise asks to reenter the data.

**Functional requirement #7:**
An administrator can to edit data of an existing account selecting a previous account. The account will be then notified by the system.

**Functional requirement #8:**
An administrator can delete an existing account. The account will be then notified by the system.

### 5.2 Parsing Requirements in NL
The first specification has been automatically parsed with a parser, reported in appendix 1, created on purpose as a typical practice, in order to collect the words. Figure 13 contains all the words found in the first specification with the corresponding occurrences. There are 326 total words as a combination of 94 different words in the first specification in NL. The second phase of this process consists in a disambiguation of requirements done manually. According to figure 11 in this phase the first specification is clarified and made more consistent selecting the most significant words, removing synonyms and collapsing those words expressing the same semantic concepts into a single common word. This analysis results in an intermediate representation preliminary to the specification of an appropriate CNL, and has been omitted for not compromising the readability of the text.

The next phase, according to figure 11 consists of a creation of a CNL and a vocabulary, based on the result of the analysis done before.
5.3 Definition of CNL and Vocabulary

As stated so far a CNL is based on a template and a set of rules, that together entail a grammar, and a dictionary of most important words. This phase starts from the intermediate requirements representation, where it is possible to see the most common aspects shared among requirements. It is possible to notice that there are different aspects in common between all the requirements, both in the usage of terms and in the usage of syntactic constructs. A suitable template for this specification could distinguish four different statements:

- **Simple Statement**: composed by “Actor + Verb + Object + Dot” This statement begins with an actor and ends with the dot and represents a declarative statement.
- **Composed Statement**: composed by a “Simple Statement + Behavioral + List of words + Dot”. This represents the declaration of an action and a behavior to follow in the
action. For readability, simple and composed statement will be called declarative statement.

- When Statement: specifies a precondition that has to be true in order to perform a simple or composed statement. In this case the period ends with a comma and it follows the template “When + Requirement + Comma + Declarative Statement” where requirement specifies a reference to another requirement.

- If-Else Statement: follows the template “If + Object State + Comma + Declarative Statement + Comma + Else + Declarative Statement + Dot” where object state is a particular property of an object. For readability, when and if-else statement will be called preconditioned statement.

This template represents the metamodel for requirements specification. This template has been chosen because of the nature of the sentences in the first requirements specification and according to what examined in the previous sections also through an analysis of the future test cases. Indeed as represented in figure 11, the definition of a CNL requires an analysis of the criteria that involve both restructured requirements and future test cases. In this case the definition of requirements has to facilitate the transformation into test cases, but at the same time has to keep readability of requirements respecting the assumptions made in subsection 4.2. Writing requirements according to this template allows expressiveness and readability for what concerns requirements specification and a sufficient level of formality to guarantee an automatic generation (see next subsection). According to this template the requirements have been manually restructured in the following specification:

Name: Book Simulator #1.

Description: If a simulator’s slot is available, a customer can book this simulator entering email, name, phone number, address, and additional information. The system validates this data. If this data are valid, the system emails the customer with booking ID slot and this data, else the customer can enter data again.

Name: Delete Booking #2.

Description: When a customer book a simulator, the customer can delete this booking, entering his email and booking ID. The system emails the customer that the booking has been deleted.

Name: Edit Booking #3.

Description: When a customer book a simulator, a coordinator can edit this booking. The system validates this data. If this data are valid, the system emails the customer that this booking has been edited, else the customer can enter data again.

Name: Assign Instructor #4.

Description: When a customer book a simulator, a coordinator can assign an instructor to this booking selecting him from a list of instructors. The coordinator confirm the booking. The system emails the customer and the instructor that an instructor has been assigned.
Name: Delete Booking #5.

Description: When a customer book a simulator, a coordinator can delete the booking. The system emails the customer that this booking has been deleted.

Name: Create Account #6.

Description: An administrator can create an account entering name, email, address and role. The system validates this data. If this data are valid, the system emails the account that an account has been created, else the administrator can enter data again.

Name: Edit Account #7.

Description: When an administrator create an account, an administrator can edit this account. Then the system emails the account that the account has been edited.

Name: Delete Account #8.

Description: When an administrator create an account, an administrator can delete this account. The system emails the account that the account has been deleted.

After the restructuration the requirements have been parsed again: 20% of the total words have been removed reducing the amount of total words from 326 to 262. Moreover, the amount of different words has been reduced of 38% from 94 words to 58. This process of restoring consistency has been performed manually. In figure 14 are reported the words in the specification.

![Figure 14: Second Specification parsed words](image-url)
From the set of words shown above, the following words have been selected in order to create a dictionary. The words have been categorized according to their syntactic value:

- **Syntactic constructs**: if, when, else.
- **Actors**: Administrator, Customer, Coordinator, System
- **Verbs**: book, edit, create, delete, validate, confirm, emails, assign, enter
- **Objects**: account, booking, instructor, data, simulator
- **Behavioral**: entering, selecting, that, with

This selection has been performed manually taking into account the results of the automatic parser but mostly considering the significant concept for the domain. This is one also the reason why the vocabulary itself should be build according to the specific domain, hence the difficulties in giving general guidelines in this field.

### 5.4 Mapping Requirements to Test Cases

The first thing that should be done after defining a CNL is to provide a template as metamodel for test cases so that it is formally possible to map the metamodel of the requirements to the metamodel of test cases. This metamodel is strictly affected by the CNL and the dictionary according to the class of words identified, and the syntactic rules defined. As it is possible to see from figure 11, the definition of test cases criteria should be discussed together within the definition of CNL criteria, facilitating both the process of test cases generation and traceability of information. In this case the decision made about the generation of test cases is that the test should be generated according to a sequence of actions, through which it is possible to simulate a whole requirement. A possible metamodel for test cases, following what found in the background and the state of the art, can be:

- **Precondition**: specifying a precondition that should be satisfied in order to perform an action contained in the requirement. A precondition represents a requirements rather than some precondition on the attributes
- **Action**: it is possible to have three kinds of different actions
  - **Base Action**: composed by “Actor + Verb + Object”
    - **Compound Action**: composed by “Simple Action + Behavior + Inputs”
    - **Conditioned Action**: composed by “Condition + Main Behavior + Alternative” where Condition is a condition over an object attribute, Main Behavior and Alternative can be a Compound or a Base action.

A possible example of test case (reported from the following specification) can be:

**Test case**: Delete Booking #5

**Precondition**: customer book simulator

**Action #1**: coordinator delete booking
**Action #2:** system emails the customer

**Behavior #2:** that

**Inputs #2:** this booking has been deleted

This comes from the specification of:

**Name:** Delete Booking #5

**Description:** When a customer book a simulator, a coordinator can delete the booking. The system emails the customer that this booking has been deleted.

The mapping between the two metamodels can be defined as follows:

- A Simple Statement is mapped to a Base Action.
- A Composed Statement is mapped to a Compound Action mapping Simple Statement with Action, the behavior with behavior and a set of words with the inputs.
- If-Else construct is mapped with a Conditioned Action, mapping Object State with condition, the then part with Main behavior (an Action either Simple or Compound), and the else period with Alternative.
- The when construct is mapped to Precondition: so the period after the keyword when until the first comma to the Precondition and the Declarative construct again with an Action.

Once the mapping between CNL and test case metamodels has been defined, it is possible to create a parser embedding the rules for deriving test cases from requirements in the form of CNL statements. Therefore, by applying the parser (transformation) to all the requirements defined according to the CNL (as shown above), it is possible to obtain corresponding test cases, as shown in the next subsection.

### 5.5 Test Cases Generation

The parser for the generation of test cases is reported in appendix 2. There are no general rules for implementing a parser like this one, depending it entirely from the defined vocabulary and rules in the CNL definition. For what concerns the one realized, the only two general rules that is possible to suggest are that the transformation should have for each template sentence provided in the CNL definition a method to find it in the specification, and at least one method that converts it in a construct of the target test case metamodel.

The result of the transformation is a set of test cases conforming to the metamodel illustrated in the previous subsection. With respect to the requirements pertaining to this case study, expressed through the CNL introduced in subsection 3, corresponding test cases have been generated as follows. For readability purposes, instead of having for each action a template: [Actor Verb Subject], the sole action will be specified:

**Test Case:** Book Simulator #1
**Condition #1:** simulator’s slot available

Main #1:

**Action #1:** customer book simulator

**Behavior #1:** entering

**Inputs #1:** email, name, phone number, address, and additional information.

**Action #2:** System validate data.

**Condition #2:** data are valid.

Main #2:

**Action #3:** system emails customer.

**Behavior #3:** with

**Inputs #3:** booking ID, slot, data.

**Alternative #2:**

**Action #3:** customer enter data.

**Alternative #1:** /

---

**Test Case:** Delete Booking #2

**Precondition:** customer book simulator

**Action #1:** customer delete booking

**Behavior #1:** entering

**Inputs #1:** email and booking ID.

**Action #2:** system emails customer

**Behavior #2:** that

**Inputs #2:** the booking has been deleted.

---

**Test Case:** Edit Booking #3

**Precondition:** customer book simulator

**Action #1:** coordinator edit booking

**Action #2:** system validates data

**Condition #1:** data are valid
Main #1:

Action #3: system emails the customer Behavior #3: that
Input #3: this booking has been edited

Alternative #1:

Action #3: customer enter data

Test Case: Assign Instructor #4
Precondition: customer book simulator
Action #1: coordinator assign instructor to this booking
Behavior #1: selecting
Input #1: list of instructors.
Action #2: coordinator confirm booking.
Action #3: system emails customer instructor
Behavior #3: that
Input #3: an instructor has been assigned.

Test Case: Delete Booking #5
Precondition: customer book simulator
Action #1: coordinator delete booking
Action #2: system emails the customer
Behavior #2: that
Inputs #2: this booking has been deleted

Test Case: Create Account #6
Action #1: administrator create account
Behavior #1: entering
Inputs #1: name, email, address and role.
Action #2: system validates data.
**Condition #3**: data are valid,

**Main #3:**

*Action #3*: system emails account 

*Behavior #3* that 

*Input #4*: an account has been created 

**Alternative #3:**

*Action #3*: administrator enter data 

---

**Test Case**: Edit Account #7

**Precondition**: administrator create account

*Action #1*: administrator edit account 

*Action #2*: system emails account 

*Behavior #2*: that 

*Input #2*: the account has been edited.

---

**Test Case**: Delete Account #8

**Precondition**: administrator create account

*Action #1*: administrator delete account. 

*Action #2*: system emails account 

*Behavior #2*: that 

*Input #2*: account has been deleted.

The generated test cases constitute an example of consistency and automatic generation of artifact since the usage of words has been reduced to 188 from 262 of the initial requirements specification. The information lost in the transformation represents 28% of the requirements specification, but it still allows to have a level of expressivity to be suitable for manual test script generation. At the same time, it could be possible to map the words in each class defined for the vocabulary to some test script target model defined in the definition of test cases criteria reported in figure 11, and hence obtain even test scripts.
5.6 Automatic Approach vs Manual in Practice

Table 5 reports the template for test cases used in the project taken into account in the case study. It is noticeable that this test case model is more complex than the one defined in the previous subsection, as result of a process of manual writing. As a consequence, a better specification is provided, but at the cost of a delay for the beginning of the process of test cases formulation, that has to wait for all the needed information to be available. Moreover, the information contained in the test cases is not always unambiguous, again remarking a manual process of transcription.

<table>
<thead>
<tr>
<th>Column Heading</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case ID</td>
<td>The ID or the number of the test case</td>
</tr>
<tr>
<td>Test Case Description</td>
<td>Description of the test case</td>
</tr>
<tr>
<td>User story code</td>
<td>The number of the user story, which is related to the test case, in the referenced</td>
</tr>
<tr>
<td></td>
<td>requirements document</td>
</tr>
<tr>
<td>Actor Involved</td>
<td>The actor of this test case</td>
</tr>
<tr>
<td>Precondition</td>
<td>The state of the system at the time the test starts</td>
</tr>
<tr>
<td>Main path</td>
<td>List of steps that needs to be applied for this test case</td>
</tr>
<tr>
<td>Alternative path (A#)</td>
<td>List of alternative steps that for this test case</td>
</tr>
<tr>
<td>Expected result</td>
<td>The result that should be observed from a succesful test</td>
</tr>
<tr>
<td>Actual result</td>
<td>The result which is observed after applying the test steps</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F: Failed,</td>
</tr>
<tr>
<td></td>
<td>N: Not tested,</td>
</tr>
<tr>
<td></td>
<td>S: Successful,</td>
</tr>
<tr>
<td></td>
<td>M: manually tested</td>
</tr>
</tbody>
</table>

In the following, I discuss a brief comparison between the real test cases for the application and the one generated through the implementation of the concept. In particular, the delete booking and book simulator test cases are reported together with some interesting results as a conceptual proof of the validity of the concept.

For delete booking, the automatically generated test case is:

**Test Case:** Delete Booking #2

**Precondition:** customer book simulator

**Action #1:** customer delete booking

**Behavior #1:** entering
Inputs #1: email and booking ID.

Action #2: system emails customer

Behavior #2: that

Inputs #2: booking has been deleted.

The real test case used in the project is the one reported in table 6. In this test case 70 words are used to express the same concepts expressed through an automatic generation with 22 words (both the counts are regardless of template structure). It is clear that in case of automation the test case in table 6 allows to see more details about the steps to follow in order to actually complete the procedure specified from the test case. But for a manual execution of the test cases, as for the scope of the project realized at MDH, the test cases created by hand do not provide critical additional information that may not be derived from automatically generated tests. Indeed, from the specification in table 6 the fields “Alternative path” do not provide a consistent and meaningful alternative from a software system point of view, and also “Expected results” and “Actual result” in this case do not provide any other significant detail about the test case. The only relevant human contribution in this test case, with respect to the auto-generated, is “Main path” points 1 and 3 adding more level of detail in the procedure of deleting a booking. This is, of course, not a big contribution, not at least for preferring a manual approach to an automatic one.

### Test Case 6: Delete Booking

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Description</td>
<td>Cancel booking</td>
</tr>
<tr>
<td>User story code</td>
<td>#13</td>
</tr>
<tr>
<td>Actor Involved</td>
<td>System, Customer</td>
</tr>
<tr>
<td>Precondition</td>
<td>Book simulator</td>
</tr>
</tbody>
</table>

#### Main path
2. Customer enters booking ID and her/his name
3. Customer clicks cancel
4. System sends notification to user

#### Alternative path (A1)
2. Customer forgot her/his booking ID
   a. Customer calls museum
   b. Coordinator cancels booking

#### Expected result
As a customer I can cancel a booking that I’ve already made

#### Actual result
The customer can delete a booking that he’ve already made

#### Conclusion
S

Table 6 Delete Booking: original manual test case of previous project for requirement #2

Following manual (table 7) and automatic (bold text) generated test cases for requirement #1:

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC#1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Description</td>
<td>Book simulator</td>
</tr>
<tr>
<td>User story code</td>
<td>#2, #3, #5, #7, #24, #25, #31, #50</td>
</tr>
<tr>
<td>Actor Involved</td>
<td>System, Customer, Coordinator</td>
</tr>
<tr>
<td>Precondition</td>
<td>/</td>
</tr>
<tr>
<td>-------------</td>
<td>--</td>
</tr>
</tbody>
</table>
| **Main path** | 1. Customer selects new booking on website.  
2. Customer selects a simulator.  
3. Customer sees the simulator’s availability.  
4. Customer selects a free time slot during the opening hours of the museum.  
5. Customer enters her/his name, email address and phone number.  
6. Customer provides additional information.  
7. Customer sees pricing information.  
8. Customer sends all booking information.  
9. System verifies provided information.  
10. System sends notification to customer. |
| **Alternative path (A1)** | 3. Customer selects time span outside the opening hours of the museum |
| **Alternative path (A2)** | 9. System verifies provided information.  
   a. System reminds customer that its name, email address, phone number or an optional postal address for the corresponding invoice is incorrect.  
   b. Customer corrects name, email address, phone number or an optional postal address for the corresponding invoice accordingly.  
   c. Customer sends all booking information again. |
| **Alternative path (A3)** | 10. System sends confirmation to coordinator.  
   a. Coordinator confirms booking.  
   System sends notification to customer. |
| **Alternative path (A4)** | 5. Customer enters her/his name, email address and phone number as well as an optional postal address for the corresponding invoice. |
| **Expected result** | As a customer i want to be able to book a simulator in the opening hour of the museum |
| **Actual result** | The customer can choose a simulator to see its availability. |
| **Conclusion** | 5 |

*Table 7: Book Simulator: original manual test case of previous project for requirement #1*

**Test Case:** Book Simulator #1

**Condition #1:** simulator’s slot available

**Main #1:**

*Action #1:* customer book simulator

*Behavior #1:* entering

*Inputs #1:* email, name, phone number, address, and additional information.

*Action #2:* System validate data.

*Condition #2:* data are valid.

**Main #2:**

*Action #3:* system emails customer.

*Behavior #3:* with

*Inputs #3:* booking ID, slot, data.

**Alternative #2:**

*Action #3:* customer enter data.
Alternative #1: /

For what concerns requirement #1, the situation is quite similar to requirement #2. Indeed the fields “Expected result” and “Actual result” still do not contain relevant information with respect to the test cases. Moreover, the alternatives do not specify a clear condition on which the branch has to be executed, although they provide additional useful information from the test case point of view. The test case book simulator in table 7 specifies the steps in 240 words, while the auto-generated one specifies it with 35. It is clear how much impact the auto-generation has in terms of consistency of the information in the process of translation from requirements to test cases. Moreover, the analysis is particularly significant if we consider that the 35 words are a subset of the ones contained in the requirements, while for the manual specification this is not true.

To summarize, by comparing the two manually defined test cases against the generated ones it is possible to conclude that in this case study automatic generation allows to preserve consistency on the requirement while saving time. Moreover, the loss of information is minimal if compared to the additional hints provided through human contributions. However, it should be always taken into consideration that for short term projects like this one, especially if including a short life span (i.e., no maintenance activities) the definition of an automatic transformation is time consuming and not worth the effort.

5.7 Conclusion

This small example constitutes only a trivial instance of the concept, given only for clarifying the concept introduced, and although it is a real case taken from a previous project, it cannot be considered as a formal proof of the concept because of its size. Nonetheless, it is possible to make some general consideration from the work done. First of all, analyzing the first and the second requirements specification it is possible to see: 1/5 of the total words have been removed reducing the amount of total words from 326 to 262. The quantity of total words has been reduced of 1/3, from 94 words to 59. There is no general index that can prove that ambiguity has been totally removed, and this is not the first purpose of the thesis. However, the relevant data for this thesis coming from the example, can be underlined through the analysis of the generated test cases: the set of words have been reduced still more from the requirements specification, and more important, no other word has been introduced, so providing an example of traceability and consistency between requirements and test cases. Although the test cases carry the same information as the requirements specification, this cannot be evaluated as a bad result since the first purpose of the thesis is to ensure consistency between requirements and test cases.
6 Discussion

Along the analysis and formulation of the topic a lot of knowledge and experience have been gathered. The hardest part in the design process was to try to formulate a general approach to the problem, that is the automated derivation of test cases from requirements. In fact, when I found partial solutions to it, they typically addressed the mapping differently depending on the application domain taken into account. Indeed, it has been noticed through the analysis of the state of the art and related work, that the gap between the industrial (or proprietary) projects and the academic projects is remarkable. As stated during the whole work, the main reason of such a difference is that the applicative domain of the projects is tightly coupled with the formulation of a CNL.

With respect to this work, most of the effort has been devoted to the identification of concepts common to all the domains in which CNLs are used. What clearly emerged from the analysis of the related state of the art is the difficulty in extracting and summarizing these shared aspects. In fact, proposing a general approach to enhance consistency and traceability between requirements and test cases through the generation of test cases from a CNL based specification involves different domains of knowledge. As clarified in the concept, there are steps involving language analysis, steps devoted to the creation of metamodels, and steps tackling automated mappings from one representation to another. As a matter of fact, there are no existing attempts (to the best knowledge of the author of this thesis) addressing the automated mapping of requirements to test cases as a whole. This forced the author to decompose the problem in sub-problems for which solutions are available, identify the solutions compatible with the overall vision proposed in this work, and synthesize them in the form of support knowledge for providing appropriate mechanisms to realize the mapping goal of the thesis.

This thesis work formulates a concept that is general in the approach adopted to reach the mapping between requirements and test cases. However, as a consequence of different domain specific practices, it was not possible to give a complete working solution for the general case. The proposed concept is portable and can be applied to different domains, but the characteristics of each domain have to be analyzed case-by-case in order to have the concrete mapping up and running.

The potential benefits of doing such an automation effort are manifold, since they come both from the solution as a whole and from the single intermediate sub-solutions. Notably, exploiting a CNL in a project allows reducing ambiguities in requirements specifications, which in turn enables an enhancement in the test cases definition phase. In addition, the adoption of a CNL in a project could be “propagated” to other phases of a development lifecycle. For instance, a CNL could also support the implementation phase: the words identified in the dictionary should be taken into account for keeping the consistency in the definition of classes, methods and objects of the (future) implementation. In this way, it could be possible to trace requirements in the code, and regardless whether test scripts were written manually or automatically generated from test cases, they could be designed in an easier way.
Taken as a whole, the automated generation of test cases on the one hand guarantees the consistency and traceability with requirements; on the other hand, it can be exploited as a way to assert quality attributes of the requirements specification itself. Moreover, this approach supports the maintainability of the system: if some of the requirements were added and/or changed (in conformance to the defined CNL), by running again the transformation it would possible to restore the consistency with the test cases, by adding new ones and/or changing existing ones, respectively.
7 Conclusions and Future Works

This work is based on the idea of keeping consistency between requirements written in a CNL form, and test cases generated automatically from the requirements specification. Generating test cases from a requirements specification is potentially beneficial but also intricate in practice. The interest of industries in these techniques is growing, and a certain corpus of work is already available in the state-of-the-art. Unfortunately, proposed approaches are typically domain-specific, while the few general ones tackle only a subportion of the problem.

This thesis work accomplishes the goals set at the beginning by giving an answer to all the research questions posed in subsection 1.2. The first step for answering the research question RQ1 was to investigate the state-of-the-art of CNLs. Then the research moved on trying to select CNLs designed for artifacts generation in order to gain the necessary knowledge for proposing a general concept. The formulated concept consists of a workflow starting with the specification of requirements in NL. Requirements are scanned by a parser that can count the occurrences of the words and find synonyms. The underlying goal is trying to remove some of the words from the specification and at the same time to identify an easy to use template for formulating periods in the specification. The restoring of requirements should be performed manually by requirements engineering, while the analysis of synonyms relevance can be performed either manually or automatically. The knowledge gained in this step allows to create a CNL, hence to select meaningful words from the specification in order to create the dictionary, and a set of rules to create a template. Then, also a template for test cases should be defined. Through another parser that maps the two specified metamodels (the one for the CNL and the other for test cases), test cases can be generated. In general these test cases will have the same expressive power of the CNL. In order to perform an enrichment of the content and generate test scripts from test cases, it is necessary to introduce a semantic parser for the words identified in the dictionary of the CNL, as well as another metamodel describing test scripts.

As a proof-of-concept, the proposal has been demonstrated in the context of a concrete project developed for a course. The demonstration experience shows promising results about the applicability of the techniques proposed in the concept. However, the concept would require a more thorough validation against industrial sized problems. Such a validation goes far beyond the scope of this work and is left as future work. Further investigation directions could include the exploitation of generated test cases to validate the requirements, especially when having a semantic parser available. In this respect, the concept and its implementation would benefit a “ stricter” adoption of MDE mechanisms. In particular, parsers could be substituted by proper model transformations that on one hand would provide trace mechanisms for free, and on the other hand would disclose the opportunity of having bidirectional mappings (from requirements to test cases and back). Eventually, the effort devoted to the definition of the CNL could be made more effective by exploiting the CNL throughout the whole development lifecycle, notably in the implementation, in the documentation of the system, and so on.
8 References


[38] http://tech-talk.org/2015/01/21/system-development-life-cycle-sdlc-approaches/
Appendix 1:
Java parser that returns the words and their occurrences in a requirements NL specification.

```java
public static void count(String s){
    String k= s.toLowerCase();
    String[] a= k.split(" ");
    System.out.println("number of words: "+ a.length);
    Hashtable<String, Integer> hs= new Hashtable<String, Integer>();
    for(int i=0; i<a.length; i++){
        if(hs.containsKey(a[i])){
            hs.replace(a[i], hs.get(a[i])+1);
        } else{
            hs.put(a[i], 1);
        }
    }
    Enumeration<String> e= hs.keys();
    int c=0;
    while(e.hasMoreElements()){c++;
        String as= e.nextElement();
        System.out.println(as+ " : "+ hs.get(as));
    }
    System.out.println("Different words: "+ c); }
```
Appendix 2:
In this appendix is specified the second parser implemented to transform requirements written according to the defined CNL in test cases. This parser has different methods (specified as `isSentence`) to find all the possible sentences that can occur in a CNL based specification (according to the one defined in section 5), and for each sentence it has a method (specified as `sentence2testcostruct`) to convert the sentences in CNL in test cases constructs.

```java
public static boolean when2precondition(String s, String[] actor, String[] behavior, String[] verb, String[] object, String follows){
    String[] src= s.split( " ");
    String actions="";
    ArrayList<String> result= new ArrayList<String>();
    boolean guard = false;
    for(int i=0; i<src.length; i++){
        String current= src[i].toLowerCase();
        for(int j=0; j<actor.length; j++){
            if(current.contains(actor[j].toLowerCase())){
                result.add(actor[j]);
            }
        }
        for(int j=0; j<verb.length; j++){
            if(current.contains(verb[j].toLowerCase())){
                result.add(verb[j]);
            }
        }
        for(int j=0; j<object.length; j++){
            if(current.contains(object[j].toLowerCase())){
                result.add(object[j]);
                action = s.substring(s.indexOf(object[j])+object[j].length());
                guard = true;
                guard = true;
                System.out.println("Precondition: "+result.toString());
            }
        }
        if(guard==true){
            break;
        }
    }
    if(isIfElse(action)){
        ifElse2Conditioned(action, actor, behavior, verb, object, follows,1);
        return true;
    } else {
        if(isBehavioral(action, behavior))
            behavioral2compound(action, actor, behavior, verb, object,1);
        else simple2base(action, actor, verb, object,1);
        return false;
    }
}
```

```java
public static void simple2base(String s, String[] actor, String[] verb, String[] object, int count){
    if(s.length() < 5){return;}
    String[] src= s.split(" ");
    ArrayList<String> result= new ArrayList<String>();
    boolean guard = false;
    for(int i=0; i<src.length; i++){
        String current= src[i].toLowerCase();
    }
}
```
```java
for(int j=0; j<actor.length; j++) {
    if (current.contains(actor[j].toLowerCase())) result.add(actor[j]);
}

for(int j=0; j<verb.length; j++) {
    if (current.equalsIgnoreCase(verb[j].toLowerCase()))
        result.add(verb[j]);
}

for(int j=0; j<object.length; j++) {
    if (current.contains(object[j].toLowerCase()))
        result.add(object[j]);
    guard = true;
}

if (guard==true){
    break;
}
}
System.out.println("Action "+count+": "+ result.toString());

public static void behavioral2compound(String s, String[] actor, String[] behavior, String[] verb, String[] object, int count){
    String[] src = s.split(" ");
    String Behavior=null, input= null;
    ArrayList<String> result = new ArrayList<String>();
    boolean guard = false;
    for(int i=0; i<src.length; i++) {
        String current= src[i].toLowerCase();
        for(int j=0; j<actor.length; j++) {
            if (current.contains(actor[j].toLowerCase()))
                result.add(actor[j]);
        }
        for(int j=0; j<verb.length; j++) {
            if (current.contains(verb[j].toLowerCase()))
                result.add(verb[j]);
        }
        for(int j=0; j<object.length; j++) {
            if (current.contains(object[j].toLowerCase()))
                result.add(object[j]);
        }

        for(int j=0; j<behavior.length; j++) {
            if (current.contains(behavior[j].toLowerCase())){
                Behavior=behavior[j];
                guard = true;
            }
        }
        input=(s.substring(s.indexOf(Behavior)+Behavior.length()));
    }
```
if (guard == true){
    break;
}

System.out.println("Action "+count": "+ result.toString());
System.out.println("Behavior "+count": "+ Behavior);
System.out.println("Input "+count": "+ input);

public static void ifElse2Conditioned(String s, String[] actor, String[] behavior, String[] verb, String[] object, String follows, int count){
    String condition = s.substring(0, s.indexOf(","));
    int c = count;
    String[] c1 = condition.split(" ");
    boolean guard = false;
    for (int i=0; i<c1.length; i++){
        for (int j=0; j<object.length; j++){
            if (c1[i].contains(object[j])){
                System.out.println("Condition "+c": "+ condition.substring(condition.indexOf(object[j])+1));
                guard = true;
                break;
            }
        }
    }
    if (guard == true) break;
}

String then;
String Else;
if (s.contains("else")){
    String[] remaining = s.split("else");
    then = remaining[0].substring(s.indexOf(",")+1);
    Else = remaining[1];
} else{
    then = s.substring(s.indexOf(",",)+1);
    Else = null;
}

System.out.println("Main "+c": ");
if (isBehavioral(then, behavior)){
    behavioral2compound(then, actor, behavior, verb, object, c);
} else {
    simple2base(then, actor, verb, object, c);
}

requirements2testcases(follows, actor, behavior, verb, object, c+1);

if(Else!=null){
    System.out.println("Alternative "+c": ");
    if (isBehavioral(Else, behavior)){
        behavioral2compound(Else, actor, behavior, verb, object, c);
    } else {
        simple2base(Else, actor, verb, object, c);
    }
} else System.out.println("Alternative "+c":/");

public static boolean isBehavioral(String s, String[] behavior){
```java
String[] src = s.split(" ");
for(int i=0; i<src.length; i++) {
    for(int j=0; j<behavior.length; j++) {
        if(src[i].contains(behavior[j])) {
            return true;
        }
    }
}
return false;

public static boolean isIfElse(String s) {
    String[] src = s.split(" ");
    for(int i=0; i<src.length; i++) {
        if(src[i].equalsIgnoreCase("if")) {
            return true;
        }
    }
    return false;
}

public static boolean isWhen(String s) {
    String src = s.toLowerCase();
    if(src.contains("when")) {
        return true;
    } else {
        return false;
    }
}

public static void requirements2testcases(String req, String[] actor, String[] behavior, String[] verb, String[] object, int count) {
    int c = count;
    String[] statements = req.split("\.");
    for(int i=0; i<statements.length; i++) {
        String current = statements[i];
        String follows = req.substring(req.indexOf(current) + current.length());
        if(isWhen(current)) {
            if(when2precondition(current, actor, behavior, verb, object, follows)) break;
        } else {
            c++;
        }
        if(isIfElse(current)) {
            ifElse2Conditioned(current, actor, behavior, verb, object, follows, c);
            break;
        }
        if(isBehavioral(current, behavior)) {
            behavioral2compound(current, actor, behavior, verb, object, c);
            c++;
            continue;
        } else {
            simple2base(current, actor, verb, object, c);
        }
    }
}
```
```
String[] statements = req.split("\"\.");
System.out.println("Test Case:", statements[0].substring(4));
requirements2testcases(req.substring(req.indexOf("Description: "+13), actor, behavior, verb, object, 1));