isec Engenharia

DEPARTAMENTO DE INFORMÁTICA E SISTEMAS

Automatic Test Definition for High-Integrity Systems

Relatório de Trabalho de Projeto para a obtenção do grau de Mestre em Engenharia Informática

Especialização em Desenvolvimento de Software

Autor

Luís Miguel Coelho Jordão

Orientadores

Professor Doutor João Cunha, DEIS-ISEC

Professor Doutor João Gabriel Silva, Critical Software

Coimbra, Maio 2023

INSTITUTO POLITÉCNICO DE COIMBRA

INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

Mestrado em Engenharia Informática

Automatic Test Definition for High-Integrity Systems

Relatório de Projeto apresentado para a obtenção do grau de Mestre em Engenharia Informática

Autor

Luís Miguel Coelho Jordão

Orientadores

Professor Doutor João Cunha

Departamento de Engenharia Informática e Sistemas

Instituto Superior de Engenharia de Coimbra

Professor Doutor João Gabriel Silva

Critical Software, SA

Coimbra, Maio 2023

Acknowledgments

Este projecto de mestrado é o resultado de um longo caminho percorrido.

O resultado deste caminho surge com um enorme sentimento de gratidão para com as pessoas que directa e indirectamente me ajudaram a percorrê-lo.

Agradeço ao professor Doutor João Cunha pela sua inexcedível orientação, pela visão crítica, pelo rigor científico e pelas palavras de motivação para o longo caminho a percorrer.

Agradeço ao professor Doutor João Gabriel Silva por me ter proposto este tema de trabalho e pela sua preciosa orientação na *Critical Software, S.A.*, pelo estímulo intelectual nas conversas técnicas que fomos tendo, e por ter zelado sempre pelo sucesso deste projecto.

Quero também agradecer ao Instituto Superior de Engenharia de Coimbra, nomeadamente aos professores, por me terem providenciado os recursos, ensinamentos e a oportunidade de conduzir este trabalho e pesquisa, assim como aos meus colegas de mestrado que me acompanharam neste percurso, pelo suporte e pelo ambiente colaborativo que providenciaram.

Aos colaboradores da *Critical Software, S.A.* pelo contributo, pelas discussões técnicas e pelo interesse que demonstraram no *Sesnando*.

À minha familia, à minha companheira e aos amigos, pelo companheirismo, pelas conversas, pela compreensão e por torceram pela minha chegada à meta. A vocês irei dedicar mais tempo.

Estou-vos bastante grato por ter aqui chegado.

Resumo

A atividade de testes é uma das tarefas mais dispendiosas no ciclo de vida de desenvolvimento de software.

No sentido de otimizar o esforço gasto nestas tarefas, foi desenvolvida uma ferramenta, *Sesnando*, cujo objectivo é interpretar e compilar requisitos de sistema escritos numa linguagem natural controlada e a partir destes gerar automaticamente um conjunto de testes que permitam verificar a implementação destes mesmos requisitos.

Durante a fase de interpretação do requisito, o *Sesnando* age como um validador da sua escrita e fornece mensagens ao utilizador sobre a sua construção. Posteriormente, gera um conjunto de testes para a sua verificação.

Neste trabalho, é também feita uma avaliação sobre as capacidades do *Sesnando* assim como uma análise relativamente aos métodos tradicionais. Os resultados obtidos mostram que é possível reduzir o esforço na atividade de especificação de testes de sistema em até 90%.

Palavras Chave: Sesnando, Testes, Automaticamente, Software, Comboios, Ferrovia, Locomotiva, Requisitos, Testes Automáticos

Abstract

The Software testing activity is one of the most expensive tasks in the lifecycle of software development.

In order to optimize the effort spent on these tasks, *Sesnando* tool has been developed whose objective is to interpret and compile system requirements written in a controlled natural language and from these, automatically generate a set of tests that verify the implementation of these requirements.

During the requirements interpretation phase, the *Sesnando* acts as a validator of the requirement writing and provides messages to the user about its construction. Afterwards, it generates a test specification for its verification.

In this work, an assessment is made of the capabilities of *Sesnando* as well as an analysis of traditional methods. The results obtained show that it is possible to reduce the effort in the system testing specification activity up to 90%.

Keywords: Sesnando, Automatic Test Generation, Testing, Tests, SDLC, Railway, Requirements

"I have no special talent,

I am only passionately curious"

Albert Einstein

Contents

List of Figures

List of Tables

List of Requirements

Acronyms

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ASDT	Aerospace, Defense and Transportation
AST	Abstract Syntax Tree
BDD	Behaviour Driven Development
CCTV	Closed-Circuit Television
CNL	Controlled Natural Language
CSW	Critical Software
FAA	Federal Aviation Administration
GUI	Graphical User Interface
GWT	Given-When-Then
ICD	Interface Control Document
IREB	International Requirements Engineering Board
ISEC	Instituto Superior de Engenharia de Coimbra
ISL	Interface Signal List
JRU	Juridic Recording Unit
MCDC	Modified Condition / Decision Coverage
MEI	Mestrado em Engenharia Informática
NLP	Natural Language Processing
PIBS	Passenger Information and Beacon System
RSSB	Rail Safety and Standards Board
SIL	Safety Integrity Level
SRS	Software Requirements Specification
TCMS	Train Control and Management System
VnV	Verification and Validation

Definitions

- Logical Signal - A requirement signal that is contained within a Requirement Clause.
- Manufacturer - Critical Software (CSW) client that assembles and maintains the hardware parts of the railway systems under testing. The manufacturers' name will not be disclosed due to a non-disclosing agreement.
- Railway Project - Safety-critical project in the area of the Railway, developed at CSW, that is intended to run on the Manufacturers' systems and has been used to evaluate *Sesnando*.
- Requirement Clause - A single condition within a Requirement Predicate.
- Requirement Predicate - GIVEN, WHEN, or THEN statements within a requirement.
- Requirement Statement - A structure that follows the Given-When-Then blueprint to define a particular behavior of the system.
- *Sesnando* Inputs - A text document containing requirement statements.
- Technical Signal - A software signal or variable that holds a value on the system and maps to a Logical Signal.
- Test Case - A set of read and write actions to verify part of the requirement.
- Test Specification - A set of test cases to verify the requirement statement.

Chapter 1

Introduction

By mid-1968 the term Software Engineering was coined at NASA by Margaret Hamilton [2]. During that time, more than 400 people were working on Apollo's software, because software was how the US was going to win the race to the moon. As it turned out, of course, software was going to help the world do so much more [3].

At that time, programming meant punching holes in stacks of punch cards, which would be processed overnight in batches on a giant computer that simulated the Apollo lander's work. Everything needed to be tested precisely [3].

One day, Margaret's daughter was playing with the command module simulator's display-and-keyboard unit and as she toyed with the keyboard, an error message popped up. Lauren had crashed the simulator by somehow launching a pre-launch program called P01 while the simulator was in mid-flight. There was no reason an astronaut would ever do this, but nonetheless, Margaret wanted to add code to prevent the crash.

NASA denied this idea by stating that the astronauts were trained to be perfect and they would not make any mistakes. So Margaret added certain procedures to the documentation stating what astronauts should and should not do. However, they did mistakes. In the late 1968, an astronaut inadvertently selected a functionality that had wiped out all the navigation data and, without that data, the Apollo computer would not be able to figure out how to get the astronauts home. Fortunately, Houston uploaded new navigational data and the Apollo astronauts came home [2] [3].

Testing activities are within the most important tasks of any business, from the world of software to the toothpaste we use, which should be meticulously tested to guarantee quality before being placed in the hands of the user. The objective of this phase is to guarantee quality standards, avoiding failures in the product that might lead to catastrophic losses.

A comprehensive verification effort ensures that all requirements in the specification are adequately tested. However, one criticism of verification is that it increases software development costs considerably [4].

The objective of this project is to create a compiler capable of understanding system Requirements and automatically generate their test cases in order to reduce the cost and effort involved in the testing activities.

1.1 Problem and Motivation

Common requirements engineering activities involve elicitation, interpretation and structuring (analysis and documentation), negotiation, verification and validation, change management and requirements tracing. There are several process models available to describe the requirements engineering process. The process itself is often depicted in different forms, including linear, incremental, non-linear and spiral models [5].

Despite the effort to produce good requirements, requirement testing can also be a costly and challenging activity. Functional tests are usually done by teams independent of the development organization on a technical basis, but have well-established, working relationships with the development organization [6].

Considering the environment that motivated this work, Software testers are required to gather solid domain-knowledge about software systems and to learn technical features and operational concepts on complex systems such as Trains, Satellites or in the area of Avionics that take time to master. A misunderstood requirement or lack of technical domain can result in tests that fail or in countless hours of back-and-forth communication.

To address this problem, a software program has been conceptualized to store common technical knowledge and with it, to be capable of automatically generate test specifications given any software requirement as an input.

1.2 Objectives

Precise requirements are critical for the creation of successful systems, therefore heavy and vague natural language is highly discouraged when it comes to critical systems [7]. Thus, having the written requirements following a stipulated grammar makes those requirements much easier to understand by developers and test engineers.

One of the goals of this project is to guarantee that the requirements of the system under testing are compliant with a predefined grammar (See Section 3.4.4) by checking their syntax and whether they are parseable, otherwise a syntax-error message is generated.

Also, traditional methods carry significant effort regarding the verification activities (Section 2.5). *Sesnando* aims to reduce this effort by automatically generating test cases from requirements. A requirement can be seen as a mini-model that takes a set of inputs and generates outputs. Its test cases will be generated by obtaining and combining the input values from the requirement according to the coverage criteria in place, whereas its outputs will be used as expected results.

1.3 Schedule

This project was planned for five main activities. Those are illustrated on Figure 1.1.

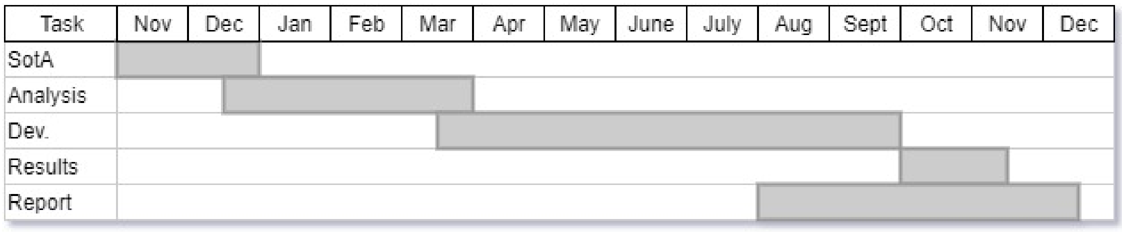


Figure 1.1: Scheduled Activities

- SotA - State of the art - Study of the existing technologies and scientific advances and how they can contribute to this project.
- Analysis - Analysis and problem dismantling - This task consisted of the analysis of the available resources, results from state of the art and *Datasets* (e.g. sets of requirements).
- Dev. - Development of the solution - Development of the application, capable of understanding requirements, generating test cases, and test artifacts.
- Results - Results and further testing - Analysis of the produced results from the developed tool as well as possible adjustments.
- Report - Final project Report - Writing a final project report containing the results from all previous phases.

1.4 Starting point

Sesnando kicked-off at Critical Software (CSW) under the management of Professor João Gabriel Silva conceived to automatically generate tests for railway projects. When the current masters' project works have started to be carried out, there were already some advances made in the research field by two software engineers at Critical Software.

The existing team carried a deep research on which programming language to use and which tools. Initially, *Sesnando* was projected to be implemented using the C++ programming language with *Flex and Bison* as its language recognition tools (Section 2.2). However, because *Flex and Bison* is no longer maintained, and due to other existing approaches providing fast development and scalability, the final decision (by the team) was the use of the C# programming language with Antlr [8] library, which is actively maintained.

When I've joined the team, the existing members had already started to design a grammar to support GIVEN-WHEN-THEN predicates (Section 2.1.1). Most of the word elements under each predicate were then tagged and processed individually. The signal mapping (Section 3.4.5) was done by editing an XML file, as well as the rules to combine test case inputs.

I have done several additions and improvements to the grammar in order to support quantifier elements (Section 3.4.4) as well as the whole grammar back-end engine for the transformation rules to generate the intermediate model, in this case, an Abstract Syntax Tree (AST) (Section 3.4.4.2). The main difference is that, instead of word tagging and individual processing, the engine now generates concrete objects (e.g a condition, a signal, etc) that extends the Abstract model (an Expression) from the requirement. These grammar additions were also supported by a tool that I have made using the python language as presented on Section 3.3, that was used to analyse the existing requirements, guaranteeing that the current designed grammar supports all possible logic paths present on such requirements.

It will be explained further that some requirement elements (signals) need to map to software elements, acting as the *glue* between requirements and software models (Section 3.4.5). This was previously achieved through the use of an XML file. Back and forth discussions within the team made evident the benefit of having an internal server containing this information (Requirement Signals mapped to Technical signals, In Section 3.3), where every tester could collaboratively populate information to support multiple instances of *Sesnando*, hence, the need to create Signal Manager. The Signal Manager is a Web Application that *Sesnando* connects to, in order to obtain information that aids the process of translation from Requirement

Signals into Technical signals that will be used to generate test cases. I have actively participated into the design of the server database (Section 3.4.7.2) due to my experience participating in Railway projects. However, the front-end design as well as the API endpoints were done by another engineer. The interface of this WebApplication is presented in Section 3.4.7 where a mapping between a Requirement signal and a Technical Signal is demonstrated. A bit of the signal translation functionality is also presented at Section 3.4.6.1.

The Test Generator (Section 3.4.6) and the Test Designer (Section 3.4.8) modules were done entirely by me. Previously the combination of test inputs, i.e., the combination of input values to meet the coverage criteria was done using the rules defined on an XML file. This proved to be inefficient, as we had to predict how many conditions a requirement would had beforehand, hence, the need to create the Test Generator where the Modified Condition / Decision Coverage (MCDC) criteria is automatically calculated for every input requirement. The functionality of the Test Generator is presented at Section 3.4.6.

Previously, in order to identify which test cases were generated for a given requirement, it was needed to read and identify such test cases directly from the test script that was generated from *Sesnando*. I have designed and implemented a Graphical User Interface (GUI) Test Designer (Section 3.4.8), that is launched from the Test Generator that presents the test specification in a tabular way, so any user can easily identify the list of test cases generated. The user is able to export test scripts from there.

The overall solution evolved to a scalable architecture respecting design principles and best practices, so that it can be easily adapted to future projects.

1.5 Scope of this document

Chapter 1 introduces the current project, its main motivations and objectives in general.

Chapter 2 presents the state of the art, what are the current scientific advances in this subject, the available tools and technology.

Chapter 3 is divided in four main sections. It discusses the work done for this project by defining the architecture of *Sesnando* 3.1, an analysis on its requirements 3.2, analysis and pre-processing of the existing resources such as existing Railway requirements 3.3 and details the design of the solution adopted 3.4.

Chapter 4 presents the results and experimental analysis of this tool in-the-field.

Chapter 5 presents a conclusion of this project and the future work.

Chapter 2

State of the art

This section presents the study of the technology and the concepts contained within this project, as well as some research done on the topic of automatic test generation. *Sesnando* interprets software requirements written in a controlled natural language using computational linguistics and language recognition techniques. Tests for these requirements are then generated by applying software testing algorithms.

This chapter is organized in 6 sections:

- Software Requirements (Section 2.1) - Presents the requirements engineering activity as well as the requirements specification techniques in the context of this project.

- Computational Linguistics and Language Recognition (Section 2.2) - Presents the language recognition techniques and the tools that were used for this project.

- Software testing and validation (Section 2.3) - Presents the V-Model and Testing activities as well as the coverage criteria in the scope of the involved railway projects.

- Vehicle Concept and Operability (Section 2.4) - Introduces concepts and railway terminology from the vehicle perspective that is presented throughout this document.

- Overview of the current industry procedures (Section 2.5) - Presents the current industry testing procedures and traditional methods.

- Related Work (Section 2.6) - Presents some of the research and works done in the area of automatic test generation.

2.1 Software Requirements

Software Requirements are a set of statements that describe the behaviour or functionality of a system. Usually, software requirements are a low-level design of the business requirements.

Software requirements play an essential role when designing a system. Requirements are descriptions of how a software product should perform and therefore, should include not only user needs but also those arising from general organizational, government and industry standards [5].

Extremely accurate requirements are hard to define, as such, some principles have been designed to ease this process, e.g. the *INCOSE Guide for Writing Requirements* [7] or the *S.M.A.R.T* criteria [9] in order to improve how they are written, so they are strongly understandable by a human or a machine.

Requirements are written prior to the development phase and are usually written by the Developers and Requirement Managers. Errors introduced during the Requirements Engineering phase have exponential costs on later phases, therefore, the importance of writing precise requirements.

Error-free requirements are the foundations on which the code should be built and tested. If the requirements are faulty, the code is likely to be faulty and tests of the code will be impossible, will fail, or give meaningless results. The effort to produce error-free requirements is considerable, but is nevertheless smaller that the effort to answer technical questions and to correct the requirements, the code and its tests because the original requirements were faulty [1].

2.1.1 Requirements Specification Techniques

A requirement is a statement which translates or expresses a need and its associated constraints and conditions. Requirements should state 'what' is needed, not 'how'. Requirements should state what is needed for the system-of-interest and not include design decisions for it [10].

The quality of software requirements is a major factor that determines the quality of a system. Most requirements are written in a natural language, which is good for understandability, but lacks precision. To make them more precise, researchers have started using formal methods and notations. There are different styles, scopes and applicability [11].

The survey "The role of formalism in system requirements" [11], has analyzed a wide range of techniques for expressing software requirements, with a degree of formalism ranging over a broad scale: from completely informal (natural language), through partially formal (semi-formal, programming-language-based), to completely formal (automata theory, other mathematical bases).

Inevitably, proponents of formal methods will point to the imprecision of natural language. Just as inevitably, opponents will argue that formal texts are incomprehensible to many stakeholders. There is truth is such statements on both sides, but they cannot end the discussion [11].

2.1.2 The Given-When-Then Blueprint

The GIVEN-WHEN-THEN (GWT) is a requirement blueprint defined by the Railway Manufacturer (a *Critical Software* client) as a syntax rule to write system requirements.

The GIVEN-WHEN-THEN-template defined as part of the well known BDD (Behaviour Driven Development) methodology in the engineering literacy is applied when writing logical requirements, as it will be presented later in this section.

This template helps to create uniform logical requirements to easily distinguish the requirement from its pre-condition (GIVEN), the trigger (WHEN) and its action (THEN) [10]. The results of a requirement (THEN) are obtained when both pre-conditions and triggers are fulfilled as per Figure 2.1.

Figure 2.1 demonstrated that only when A and B conditions are fulfilled, the C action is set.

It is strongly recommended that when comparing values on requirements (equal, greater than, less than, etc.), the requirement text should avoid using mathematical symbols. The intent is to prevent risk of deviate into writing equations instead of text. For instance:

- Instead of writing
  GIVEN the train tread brake applied status = TRUE,
  it should be written as:
  GIVEN the train tread brake applied status is equal to TRUE

The main difference between *GIVEN* and *WHEN* statement is that the *GIVEN* conditions must be present in order for the *THEN* actions to be performed whereas the *WHEN* can be defined as a trigger, in other words, a button press or a signal pulse. See Figure 2.1.

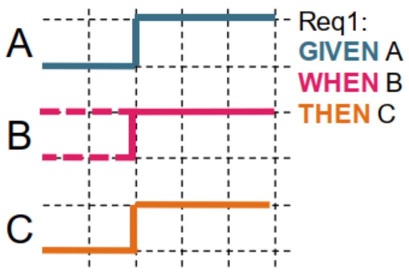


Figure 2.1: GIVEN-WHEN-THEN Requirement Histogram [1]

The need for this formality is also presented by the International Requirements and Engineering Board (IREB) on the Requirements Engineering Fundamentals book, chapter 5 [12], where a Conditional condition and a temporal condition must be stated.

Example of a requirement following IREB conventions:

if train speed is greater than 3km/h **as soon as** emergency brake pushbutton is pressed the train shall apply emergency brakes.
(Note that the word *shall* has the same effect as the word *must*).

a) Conditional condition: if

b) Temporal condition: as soon as

As per the Manufacturer, in the context of this project, for the conditions the following keywords are defined and shall be used:

a) Conditional condition: GIVEN

b) Temporal condition: WHEN

If several conditional conditions (preconditions) have to be fulfilled, then each precondition shall begin in a new line. Begin the new line starting with the second precondition with the keyword AND. Example on Requirement REQ211-1.

```
REQUIREMENT REQ211-1
        GIVEN <precondition 1>
                AND <precondition 2>
                AND <precondition 3>
        WHEN <trigger1>
        THEN <action1>.
```

As such, defining the previous requirement into a Given-When-Then stricture results in the following expression:

```
REQUIREMENT REQ211-2
        GIVEN train speed is greater than 3km/h
        WHEN emergency brake pushbutton is pressed
        THEN the train shall apply emergency brakes.
```

2.2 Computational Linguistics and Language Recognition

The human brain subconsciously groups character sequences into words and looks them up in a dictionary before recognizing a grammatical structure. The first translation phase is called lexical analysis and operates on the incoming character stream. The second phase is called parsing and operates on a stream of vocabulary symbols, called tokens, emanating from the lexical analyzer [8].

Flex and Bison (formerly Yacc) were the first widely used open source versions for lexing and parsing. Each of these software has a history of more than 30 years, which is an accomplishment in itself. For some people, they are still the first software they think of when it comes to parsing [13][14].

Like any other software, there are benefits from implementing a compiler in a high-level language. In particular, a compiler can be self-hosted, that is, written in the programming language it compiles. Building a self-hosting compiler is a bootstrapping problem, i.e. the first such compiler for a language must be either hand written machine code or compiled by a compiler written in another language, or compiled by running the compiler in an interpreter [15]. Besides lexing and parsing, tree walking involves traversing the parse tree and applying rules or other operations at each node in the tree to extract meaning or perform other language-processing tasks.

The translation requires multiple steps (lexing, parsing and tree walking) as presented on Figure 2.2.

This intermediate form is a tree data structure, called an abstract syntax tree (AST)(Figure 2.5), and is a highly processed, condensed version of the input [8].

A more modern language translation solution is AnTLR, which is inspired on Flex and Bison, and is a sophisticated parser generator that can be used to implement language interpreters, compilers, and other translators [8]. An AnTLR grammar is an artifact written in *Backus-Naur Form* (BNF) that contains lexer tokens and parsing rules.

Backus-Naur Form is a formal mathematical way to specify context-free grammars. John Backus, who was also the inventor of *Fortran*, won the 1977 Turing Award

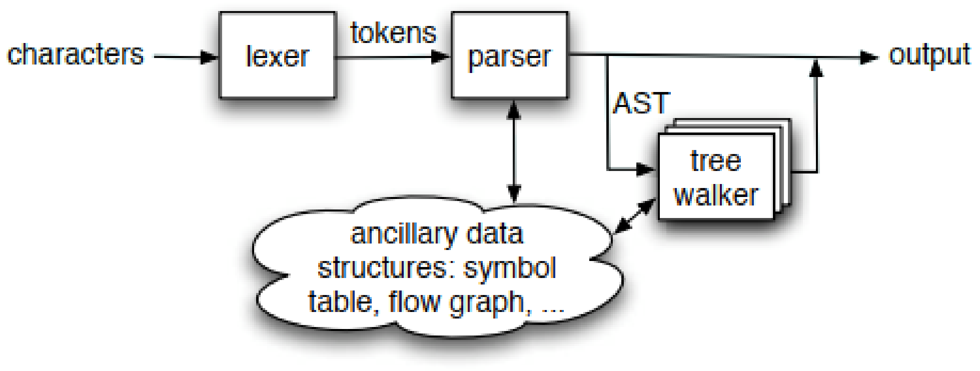


Figure 2.2: Translation Process

for BNF and *Fortran* [16]. An example of the BNF form for decimal numbers is presented in Figure 2.3.

```
<expr> ::= '-' <num> | <num>
<num> ::= <digits>
        | <digits> '.' <digits>
<digits> ::= <digit>
           | <digit> <digits>
<digit> ::= '0' | '1' | '2' | '3' |
  '4' | '5' | '6' | '7' | '8' | '9'
```

Figure 2.3: Decimal Numbers in BNF notation

Niklaus Wirth started the early developments of Extended Backus-Naur Form (EBNF) which is an extension of BNF and make expressing grammars more convenient [16]. One of the greatest improvements is the definition of the symbol occurrences (from 0 or more occurences). An example of the EBNF form for decimal numbers is presented on Figure 2.4.

```
<expr> := '-'? <digit>+ ('.' <digit>+)?
<digit> := '0' | '1' | '2' | '3' | '4' |
  '5' | '6' | '7' | '8' | '9'
```

Figure 2.4: EBNF Notation Decimal Numbers

EBNF is now widely used as the de-facto standard to define grammars and programming languages.

AnTLR executes an action according to its position within the grammar. In this way, it executes different code for different phrases (sentence fragments). For example, an action within, say, an expression rule is executed only when the parser is recognizing an expression [8]. This process is demonstrated on Figure 2.5.

ANTLR is mainly targeted for Object Oriented Programming (OOP) languages

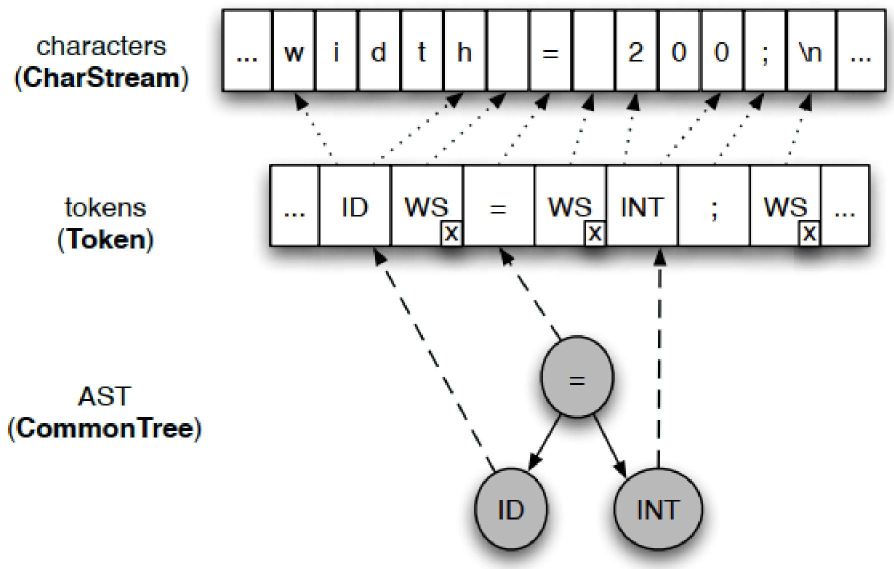


Figure 2.5: Characters, tokens and AST

like *Java* or C# and allows for an easily scalable solution and a fast-paced development, hence, the use of ANTLR in *Sesnando*.

2.3 Software testing and validation

The purpose of software testing activities is to verify that a system is operating according to requirements and meets the customer needs. Without tests, systems would be prone to errors. Testing can occur at different levels and this is presented in the commonly known in the engineering literacy as V-Model, as presented on Figure 2.6. This cycle is named V-Model after its "V" design and is used to improve the overall quality of the software, where for every design and implementation phase there is a verification activity. V-Model can be presented with slight changes as it is usually tailored according to the type of project under development.

The V-model was derived from the waterfall model so there is similarity in the activities and their sequence. There is progression through requirements, design, coding/implementation, testing, and release. The single lengthiest activity in this software development flow is code development. Code development encompasses fixing previously identified and prioritized defects and adding features. Testing activities on the same phase can be done concurrently [17]. However, as per 2.6, a new upper testing phase only starts when the previous one is verified.

The activities present on Figure 2.6 can be described as follows.

- User / Business Requirements - Conceptualisation phase and customer require-

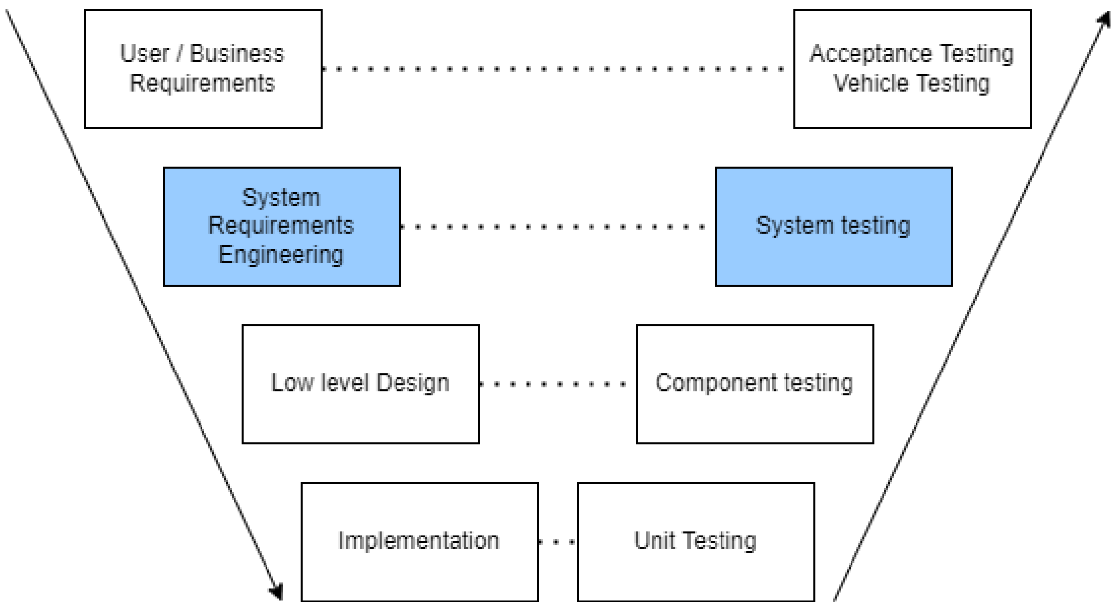

User / Business Requirements
Acceptance Testing Vehicle Testing
System Requirements Engineering
System testing
Low level Design
Component testing
Implementation
Unit Testing

Figure 2.6: V-Model

ments analysis and engineering.

- System Requirements Engineering - Involves the apportionment of requirements on a technical level.
- Low level design - System architecture and interfacing of low level components.
- Implementation - Coding
- Unit testing - Unit testing involves testing the smallest units of code, typically individual functions or methods, to ensure that they are working properly. This is typically done by the developer who wrote the code, and it is usually done early in the development process to catch any bugs or problems before they become more difficult to fix.
- Component testing - Component testing, involves testing groups of related units of code, called components, to ensure that they work together properly. This is usually done by a quality assurance (QA) team, and it is typically done later in the development process, once the individual units have been tested.
- System testing - Tests are performed on a simulated environment where only the domain model and specification of requirements are known. These tests should not be performed by developers.
- Acceptance Testing - Also known as Functional testing, these tests are performed on the final product using a specification generated from the Business

Requirements. Those can be done with the presence of the client, hence, the name Acceptance tests.

The scope of this project is within the System testing activities, as per Figure 2.6. System testing is a testing technique where only the Requirement Specification and the Domain model is known. There are several coverage criteria that serves the purpose of requirements specification testing. A coverage criteria is a rule of collection of rules that impose test requirements on a test set. The coverage criteria under the scope of this Railway project is Modified Condition / Decision Coverage (MCDC) as defined by the Manufacturer. It is a de-facto standard for certifying software in the safety-critical markets [18], hence, the use of MCDC.

MCDC applies to logic testing and has a high probability of error detection for the cost incurred (number of tests), especially when compared with other coverage criteria specified in DO- 178B [19] (Statement Coverage, Decision Coverage). The benefit of MCDC is that it requires a much smaller number of test cases while sustaining a quite high error-detection probability. As published by Federal Aviation Administration (FAA), the following is stated regarding MCDC:

- Modified Condition / Decision Coverage - Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome [20].

2.4 Vehicle Concept and Operability

Figure 2.7, illustrates the most common train topology on the projects where Critical Software is involved. This figure also presents a common terminology that is present throughout this document.

As per Figure 2.7, a train unit might contain 3 to 10 cars, in which the standard is the use of 5 cars. A train unit is commonly defined as "Unit" or "Consist" and can be coupled to form a longer operating train.

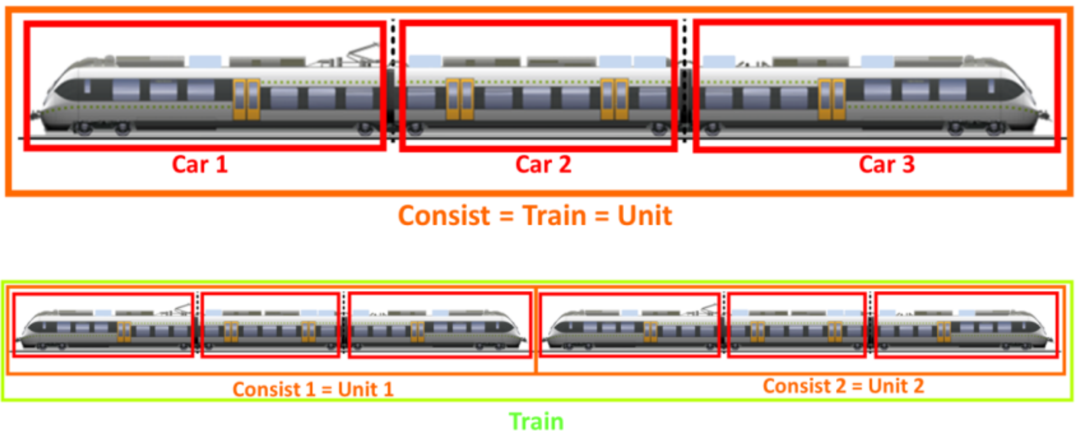


Figure 2.7: Train coupling and configurations

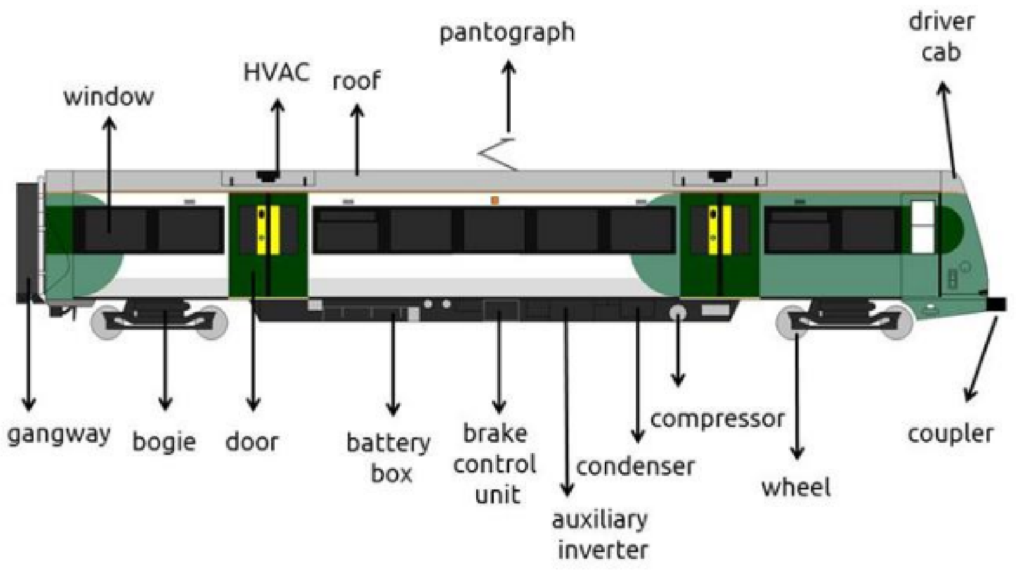


Figure 2.8: Train main components

Figure 2.8 presents the most common components on the train. Train cars are very similar throughout the train. However, train cabs are only present on the train unit extremes, also, a train unit contains only one pantograph. These values are doubled when a train is formed of two units.

2.5 Overview of the current industry procedures

This section presents the system testing activities in the railway industry, which Critical Software (CSW) is involved. The process begins when the rolling stock manufacturer receives a set of customer requirements commonly known as *Business*

Requirements.

Aside from these requirements, there are *Rule Book* requirements created by Rail Safety and Standards Board, which are documents that contain direct instructions to Railway staff, as well as the requirements from the *standards* catalogues related to a Safety Integrity Level (SIL).

The concept of safety integrity was then taken up and adapted in the various offshoot standards. For the railway domain, the concept of safety integrity was taken up in the CENELEC EN 50126, EN 50128 and EN 50129 standards [21].

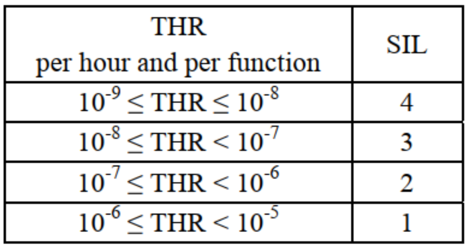
SIL levels are based on risk of failure and go from level 1 to level 4 according to the criticality of a system, being SIL-4 systems with a lower probability to fail.

The acceptable (or tolerable) risk is a value of a risk level arrived at by an objective, deliberate decision. This threshold of acceptability is known as the Tolerable Hazard Rate (THR). The THR is expressed as the probability of occurrence of a failure, expressed in the form

$$10^x$$

per hour. When identifying hazardous situations, it is crucial to assign them a THR and consequently a SIL Level [21].

For the railway domain, Figure 2.9 shows the link which exists between the SIL and the THR. In fact, this table was introduced for railway signaling and can be extended to all or part of the system [21].



THR per hour and per function	SIL
$10^{-9} \leq THR \leq 10^{-8}$	4
$10^{-8} \leq THR < 10^{-7}$	3
$10^{-7} \leq THR < 10^{-6}$	2
$10^{-6} \leq THR < 10^{-5}$	1

Figure 2.9: SIL Table from CENELEC EN 50129

For instance, software for the Passenger Information System and for Passenger Comfort is non-SIL (SIL-0) as the failure of this system does not cause a disastrous situation. However, for instance, software that deals with Traction, Braking and Doors is identified as SIL2, as if this software fails it might incur in damages. To mitigate this identified risk, SIL2 software runs on a redundant device that becomes the primary device if the other fails.

There are several *standards* catalogues applicable to the railway industry. The practices involved in the development of the software are mostly presented on BS EN 50126 [22] and CENELEC 50128 [21]. BS EN 50126 is a standard that introduces the RAMS process, which is a process that characterises a system in terms of Reliability, Availability, Maintainability and Safety. The objective of the RAMS process described in this standard (BS EN 50126) is to ensure that all aspects of RAMS are covered in order to make provision for the safety of railway applications and for the avoidance of loss of their value [22].

CENELEC 50128 specifies the process and technical requirements for the development of software for programmable electronic systems for use in railway control and protection applications. It is aimed at use in any area where there are safety implications. These systems can be implemented using dedicated microprocessors, programmable logic controllers, multiprocessor distributed systems, larger scale central processor systems or other architectures [23].

The Functional Architecture team is responsible for the apportionment of the existing requirements, that is part of System Requirements Engineering as illustrated on Figure 2.6 . These existing requirements are usually derived into new architectural, performance, functional and sub-system requirements as seen on Figure 2.10.

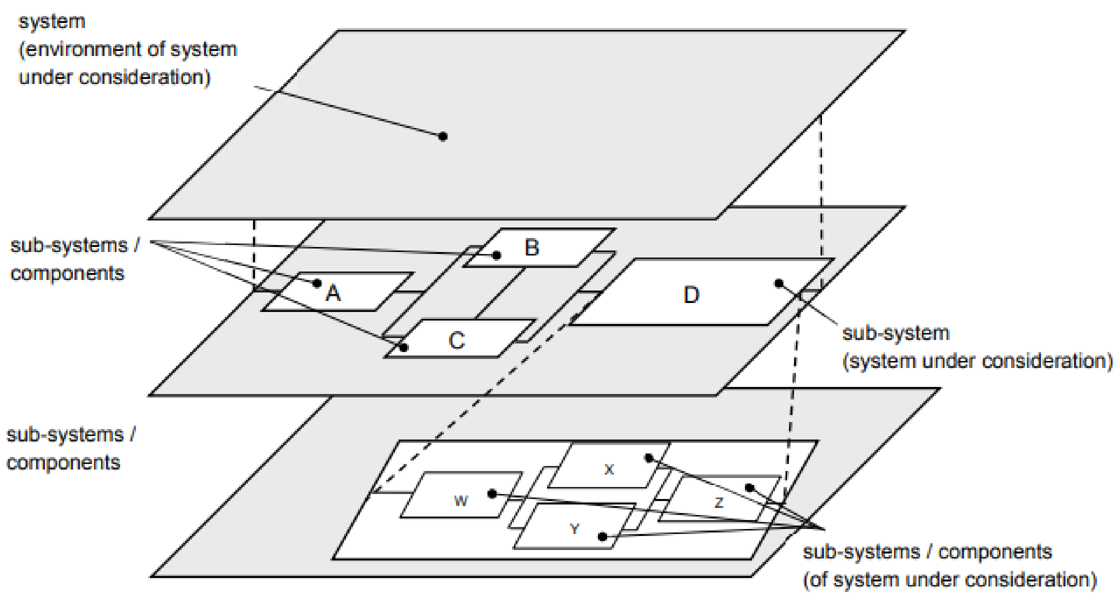


Figure 2.10: System Hierarchy

The functional team, responsible of Functional requirements, needs to assure that these derived requirements satisfy the original customer requirements, are testable, coherent and non-contradictory. The development team writes software requirements derived from functional level and implements the software following the norm EN 61131 [24]. To facilitate code safety, Ada language [25] and Misra-C guidelines [26] are also used at critical software in the area of Aerospace, Defense and Transportation (ASDT).

The software is only uploaded to vehicle level once all previous tests pass under a simulated environment in the form of a *Test Rack* (Figure 2.11), containing all the train equipment that compose the TCMS (Train Control and Management System) like the Central Control Unit of the train.

Figure 2.11: Test Rack - Train Test Equipment

The activity of testing relies on the interpretation of the software requirements stored on an IBM DOORS database by the manufacturer, and the verification if the system is behaving accordingly. To support the testing activities at system testing level, an Interface Control Document (ICD) is required.

The realisation of testing a requirement relies on the identification of software signals that represent the logic present on a requirement e.g. the command to open a train door, passenger emergency button, etc. and on the definition of the combination between the inputs and expected results in order to meet the requirement coverage criteria. This combination of test steps results in a test specification that can be converted into a test script using a test tool provided by the manufacturer. The test script is then loaded on the test tool which will then produce a test report file. Tests might by automatic or semi-automatic depending on the nature of the system.

2.6 Related Work

A controlled natural language (CNL) is a language that is designed to be easy for humans to read and write, while still being precise and well-defined enough to be processed by a computer. CNLs are often used in specific domains where it is important to have clear and unambiguous communication, such as legal documents, medical records, or technical specifications [27]. Like a Controlled Natural Language

(CNL), a Requirement Specification Language (RSL) has a formal syntax. In 1992, NASA described a general purpose RSL.

RSL is a hybrid of features found in several popular requirement specification languages. The purpose of RSL is to describe precisely the external structure of a system comprised of hardware, software, and human processing elements. To overcome the deficiencies of informal specification languages, RSL includes facilities for mathematical specification [28]. Purposes of RSL include the production of executable code and test cases for a system.

Daniel Maciel et al., proposed a Model Based Testing approach using an RSL and the Robot Framework. This approach included model-to-model transformation techniques, semi-automatically generating test cases from the RSL as well as test cases into test scripts [29].

The benefits of automatic test generation are widely acknowledged today and there are many proposed approaches in the literature [30]. Chunhui Wang et al. proposes in the paper (Automatic Generation of Acceptance Test Cases from Use Case Specifications: an NLP-based Approach) an approach that supports the generation of executable, system-level, acceptance test cases from requirements specifications in natural language, with the goal of reducing the manual effort required to generate test cases and ensuring requirements coverage [30].

System test cases might be derived from the requirements of the system under test [31]. Javier J. Gutiérrez et al. presented a Survey (Generation of test cases from functional requirements) containing the results of among 13 approaches to drive the generation of test cases from functional requirements [31].

Pre-made tools and solutions cater for the average use cases and sometimes simplify things to make it easy to use. Due to specific needs, such as the development of a grammar for a specific language (Manufacturers' Given-When-Then blueprint), requirement coverage criteria, and type of System under test and centralized testing information, *Sesnando* has been created.

Chapter 3

Sesnando

This chapter presents the core functionalities of *Sesnando* project and is divided into four sections.

- Architecture of *Sesnando* (Section 3.1) - Describes the project architecture by presenting the main modules, communication and data flow.
- Requirements of *Sesnando* (Section 3.2) - Presents an analysis of the requirements of this Project.
- Data Analysis and Pre-Processing (Section 3.3) - Presents an analysis of the existing Railway requirements and how they have been used to define the Grammar of the requirement interpreter.
- Software Detailed Design (Section 3.4) - Presents the functionality of *Sesnando*, its operability concepts and building blocks.

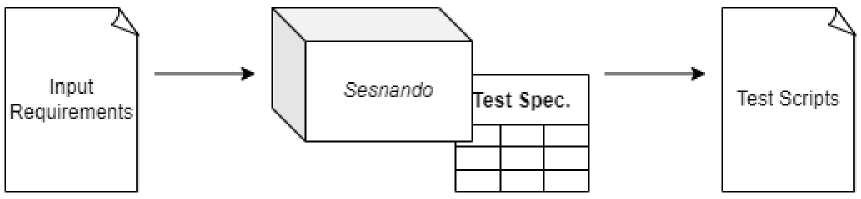


Figure 3.1: Basic execution flow

In general, this chapter describes how *Sesnando* can be used, what the intermediate steps are, how they have been built and what will be generated as a result.

The entry point of this software is a *Windows Console application* that can take a set of arguments (as defined on Section 3.4.1). From this application *Sesnando* takes the requirements (compliant with format defined on Section 3.4.2) contained on an input file and starts generating test cases (Section 3.4.6).

Once the set of test cases have been generated for each requirement, *Sesnando* calls a Graphical User Interface (GUI) application (Test Designer - Section 3.4.8), to display such test cases.

Most Software Development Life Cycles (SDLC) require test specifications to be reviewed and approved, as such, the user is able to edit the generated test specifications (Fig. 3.16) if an improvement is required before exporting its corresponding test script. The final test script can then be imported into the Manufacturers' tool to execute such tests.

It is important to mention that in order to properly generate test specifications, *Sesnando* relies on an internet connection to access a remote repository (Signal Manager Service) containing information about the software under testing. This is detailed on Sections 3.4.6.3, 3.4.5 and 3.4.7.

The displayed test specification can be saved on a persistent file (which can be opened at any time using the *Test Designer*). The user is also able to export this test specification as a test script to be used on an external test execution tool defined by the Railway manufacturer (Section 4). The saving functionality has been included for cases when an internet connection is not available and the user wants to recreate the test script.

While I actively participated in the design of the Signal Manager Service, the works presented on this thesis do not include its development and programming activities.

Figure 3.1 presents the basic execution flow of *Sesnando* whereas Figure 3.3 on Section 3.1 presents the main execution flow in a detailed manner.

3.1 Architecture of Sesnando

Sesnando is divided into multiple modules that communicate with each other. This section presents these modules and their main roles.

- Sesnando Setup – This module is responsible for verifying and installing all the Application dependencies on the operating system.
- Sesnando Compiler – This module is the entry point of *Sesnando* Application and will call all the relevant modules as needed.
- Requirements Module – This module will be called at the beginning of the program execution. It will read the Input Requirements and will then parse its contents using an Abstract Syntax Tree (Section 2.2), i.e. this module contains

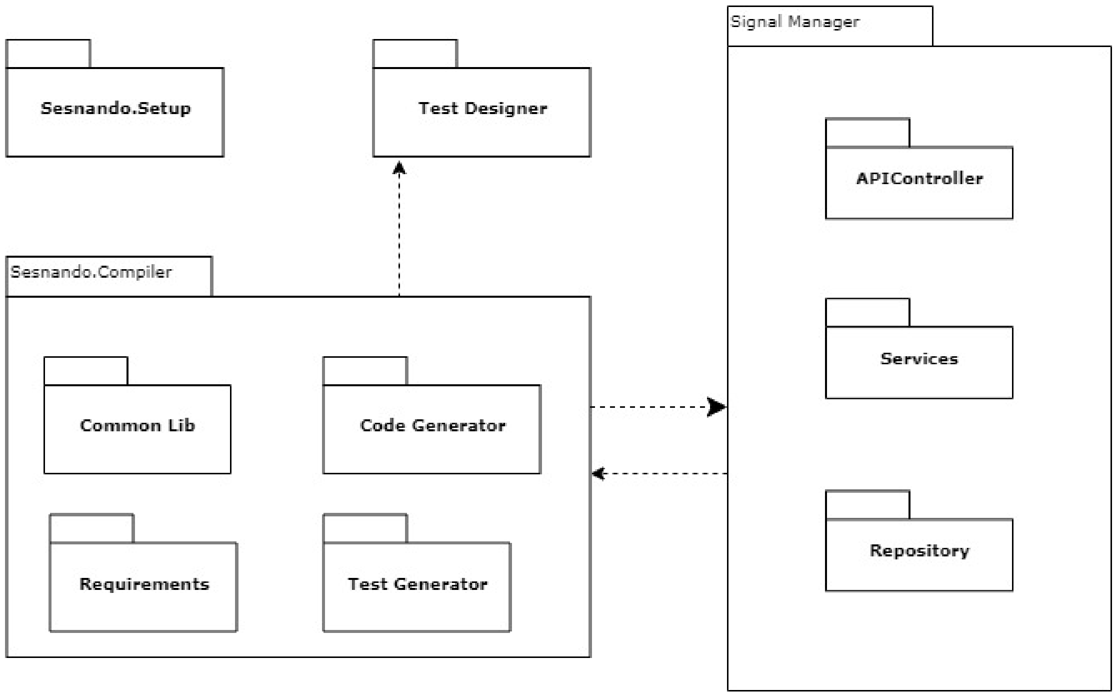


Figure 3.2: Main application modules

a predefined grammar that will be used to compile the requirements into a set of Objects. This will be detailed on Section 3.4.3).

- Test Generator – Once the object tree is successfully constructed by the Requirements Module, this module will be notified to extract the required data from it in order to start to define the first tests. See Section 3.4.6.

- Common Lib – A Library where the actual Abstract Syntax Tree objects will be stored, accessed by Requirements module and Test Generator module.

- Code Generator – This module receives the standard output from the test generator and will then generate proper test scripts compatible with the external test environment.

- Signal Manager – Signal Manager is a remote service that receives requests from the Test Generator. Test Generator connects to Signal Manager to request additional data of a requirement signal, e.g., in which state should the system be set in order to verify a requirement. This is fully described on Section 3.4.7. Signal manager implements a Controller-Service-Repository Pattern – This is a layered design pattern presented as a good practice when developing a back-end service as it provides a separation of concerns.

  - Controller – This layer is responsible to expose the Rest API endpoints, so the functionality can be consumed by external entities.

  - Service – This layer contains all the Business Logic and receives requests from the Controller and dispatches read and write requests to the repository.

  - Repository – This layer receives requests and dispatches data to the Service layer. Its only responsibility is to read and to persist data on the Signal Manager Database.

  Tom Collings published an article in 2021 explaining this pattern on `https://tom-collings.medium.com/controller-service-repository-16e29a4684e5` [32].

- Test Designer – Test designer is a detachable Graphical User Interface (GUI) that displays the generated test cases to the User. This tool also allows the user to review, edit and export the generated test specifications. This is fully described on Section 3.4.8.

Regarding Figure 3.2, *Sesnando* is provided as a native windows application that contains Setup, Compiler and Test Designer. *Sesnando* communicates with a Signal Manager service to obtain additional railway requirements data and calls the Test Designer module to present test data to the user.

Figure 3.3, details this approach, the *Requirements Compiler* takes the requirements from the input file and parses them into a parse tree, which is managed by ASTController on Common Lib project. The structure of this tree is presented on Class Diagram - Fig. 3.8.

Test Generator (3.4.6) is then called to retrieve the parse tree by asking the *AST Controller* and requests the necessary information from the *Signal Manager* (Section 3.4.7) trough a *Web API*.

Once the tests are generated, the *Test Designer* (Section 3.4.8) is called to display the test specification to the user where a test script can be created to be imported to the customer test tool.

This is a superficial overview of the main execution flow of *Sesnando*. The whole process is detailed on Section 3.4.

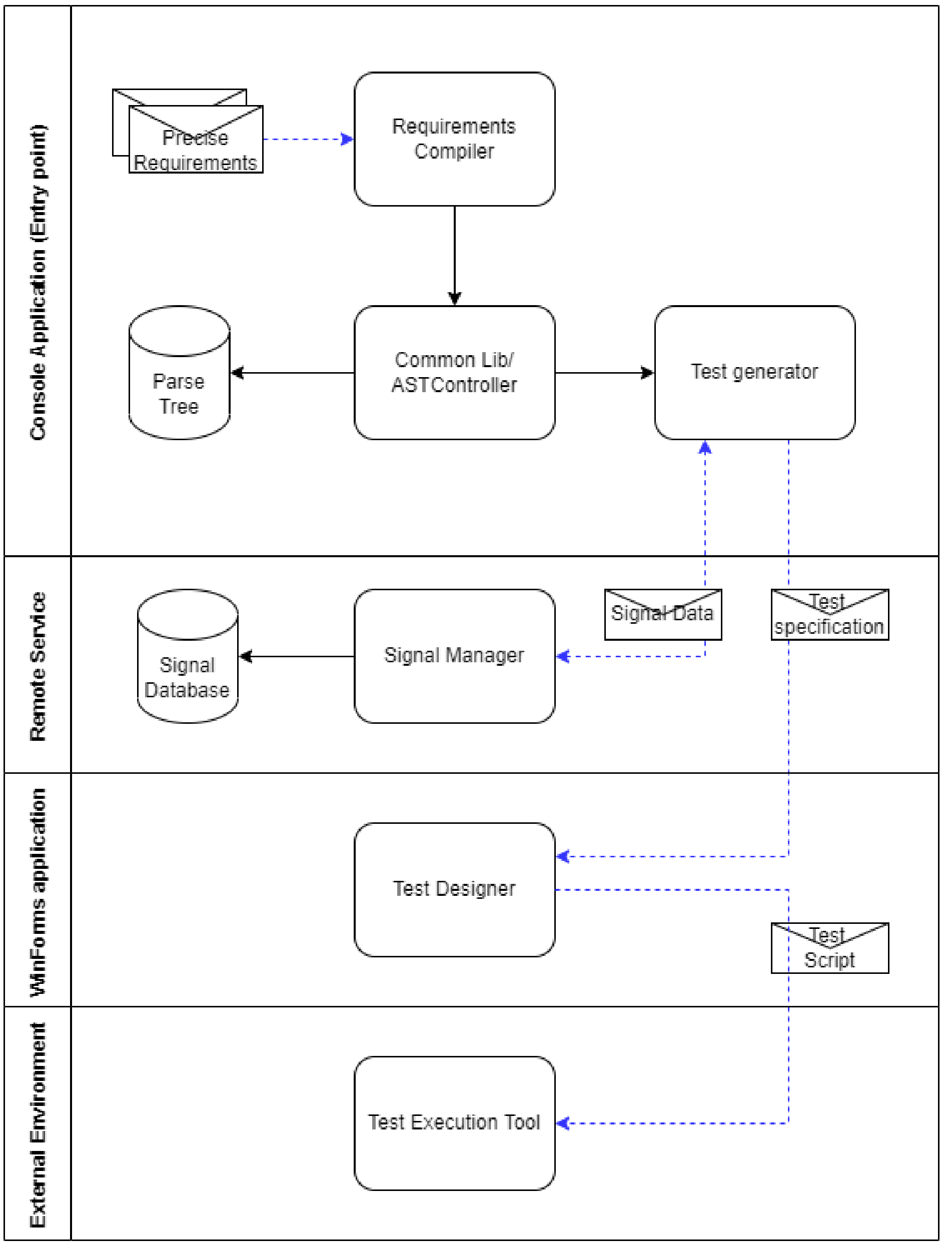


Figure 3.3: Main flow diagram

3.2 Requirements of Sesnando

The activity of identifying software functionalities and behavior is described as *Requirements Engineering.* The requirements of the *Sesnando* tool (denominated as 'The application') will be identified in a simple manner.

- The application must be able to recognize a requirement statement (Section 3.4.2) identified by its keyword REQUIREMENT.
- The application must be able to identify *Given*, *When* and *Then* predicates (Section 3.4.2).
- The application must extract Logical Expressions between *AND* and *OR* boolean operators (Section 3.4.4).
- The application must identify Logical Expressions using the operators: *Equal to*, *Lower Than*, *Lower Than or Equal*, *Greater Than* and *Greater Than or Equal* (Section 3.4.4).
- The application must be able to identify Requirement Signals (Logical Signals, Section 3.3) within a Logical Expression.
- The application must be able to identify an Operand (bool, int) (Section 3.4.4) within a Logical Expression.
- The application must be able to identify a Logical Expression Quantifier (Section 3.4.4.1), i.e., the system components to which the expression shall be evaluated, e.g., a train axle.
- The application must be able to compile the requirement into a Parse tree (Section 3.4.3).
- The application must be able to access the Parse Tree (Section 2.2) and generate a set of test cases from the compiled requirements (Section 3.4.6).
- The application must be able to generate a test script (Section 3.4.8) to be used on the external testing environment (provided by the customer, Section 4).
- The application must allow saving the generated set of test cases into a persistence storage in CSV file format (Section 3.4.8).

3.3 Data Analysis and Pre-processing

This section presents the analysis procedures towards the available Railway software requirements. The requirements of the Railway project used for this study are stored on a Requirement Management Tool. Those requirements (presented as examples on Section 3.4) have been exported and analysed in order to extract relevant knowledge from them towards building a solid grammar that could satisfy the logic of these requirements. This process is presented on Figure 3.4.

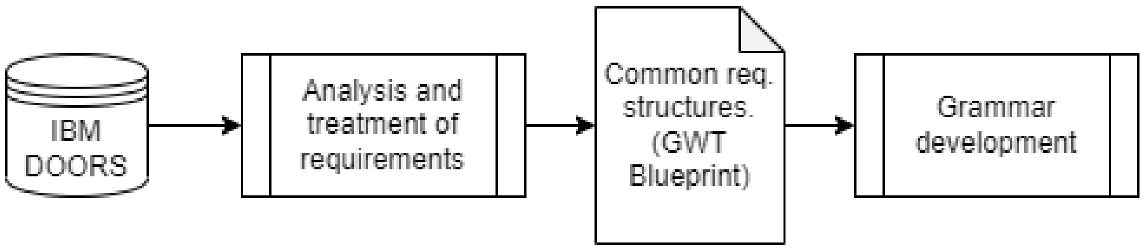

IBM DOORS
Analysis and treatment of requirements
Common req. structures. (GWT Blueprint)
Grammar development

Figure 3.4: Process of requirement analysis

Regarding Figure 3.4, IBM DOORS is the repository containing the Railway software requirements. Initially, sets of requirements have been exported from this repository in chunks of Excel files as supported by the IBM DOORS interface, but later, a software tool has been developed (by myself) with the ability to directly access the requirement repository via Open Services for Lifecycle Collaboration (OSLC) REST API and to support the Analysis and treatment process of requirements.

The objective of this analysis is to identify the most common requirement structures towards building a grammar that is able to support them. The principal analysis taken was in the fields of the contiguous sequences and main requirement structures. For this, several models on N-Grams and Skip-grams have been generated. N-grams model analysis are widely used in computational linguistics and communication theory such as Natural Language Processing (NLP) [33].

N-gram analysis is a technique that involves identifying sequences of N consecutive words in a text. For instance, N-grams of value 5 (5-Grams), statistically presents the most repeating sequences of 5 words, and so on. This is used to identify the most common requirement structures. Some parts of the requirements have even been anonymized, such as Requirement Signals, to avoid biased data, i.e, repeating structures containing the same signal.

Skip-gram analysis is similar, but allows words to be skipped in the sequence. Stop-word analysis is used to identify and remove commonly used words with little meaning from a text, such as "the", "a", etc.

Stop-Words analysis was a tentative to remove words that could have less meaning on requirements

The techniques to export the best N-gram models relied on arbitrary values and *trial and error*. N-grams models above the value of 10 started to produce the same sequences, so the model analysis sat between the values of 2 and 10. As it was intended to obtain a variety of the most repeating sequences. Figure 3.5 presents a N-gram Model analysis where N = 10.

The results of this analysis expressed the grammar of *Sesnando* shall contain:

- Logical boolean operators such as "AND" and "OR";
- Operators such as: Equal, greater than, less than, greater than or equal to, less than or equal to;
- Signal attributes such as "is active";
- Quantifiers containing their attribute and scope, as in "for at least one side of the train" (four quantifiers are supported on *Sesnando*, See Section 3.4.4.1);
- *THEN* condition in the form of "THEN <Outcome Requirement signal> is set to *VALUE*".

These have been used to define the requirements grammar, as presented on Section 3.4.4 (Grammar Elements) and Section 3.4.4.2 (Defining requirement grammar). In other words, the existing requirement structure and common natural instructions have been analysed to support the writing of a grammar that supports the same logic that is already presented on the existing requirements.

This analysis and the obtained results involved the writing of a python program that took several sets of requirements as input. The main activities that have been performed using this tool are presented as bullet points.

- Export existing requirements written in a natural language from the manufacturer requirement repository (IBM Doors).
- Treat existing requirements by anonymizing Requirement Signals, so that, these do not influence the extraction of the most common used expressions. Anonymizing the data can also help prevent any bias or unfair treatment of the requirements being analyzed, as it ensures that the analysis is based solely on the information that defines the requirement structure and not on any signal information.
- N-gram, Skip-gram and Stop-word analysis [34], as previously defined.
- Analysis and plotting of the obtained results that will be used to support the requirement logic already presented in natural form.

The following is a list of libraries that have been used to develop this Requirement analysis tool:

- pandas - for importing requirement files to the python tool as dataframes, for data storage and data analysis. "pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language." - For further details see: `https://pandas.pydata.org/`.
- nltk - Natural Language Toolkit for word sequence and requirement structure analysis. "NLTK is a leading platform for building Python programs to work with human language data." For further details see: `https://www.nltk.org/`.
- matplot - Data plotting and visualisation, for instance, to build the diagram of Figure 3.5. "Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python" For further details see: `https://matplotlib.org/`.
- xlswriter - To locally persist processed data. "XlsxWriter is a Python module that can be used to write text, numbers, formulas and hyperlinks to multiple worksheets" For further details see: `https://xlsxwriter.readthedocs.io/`.

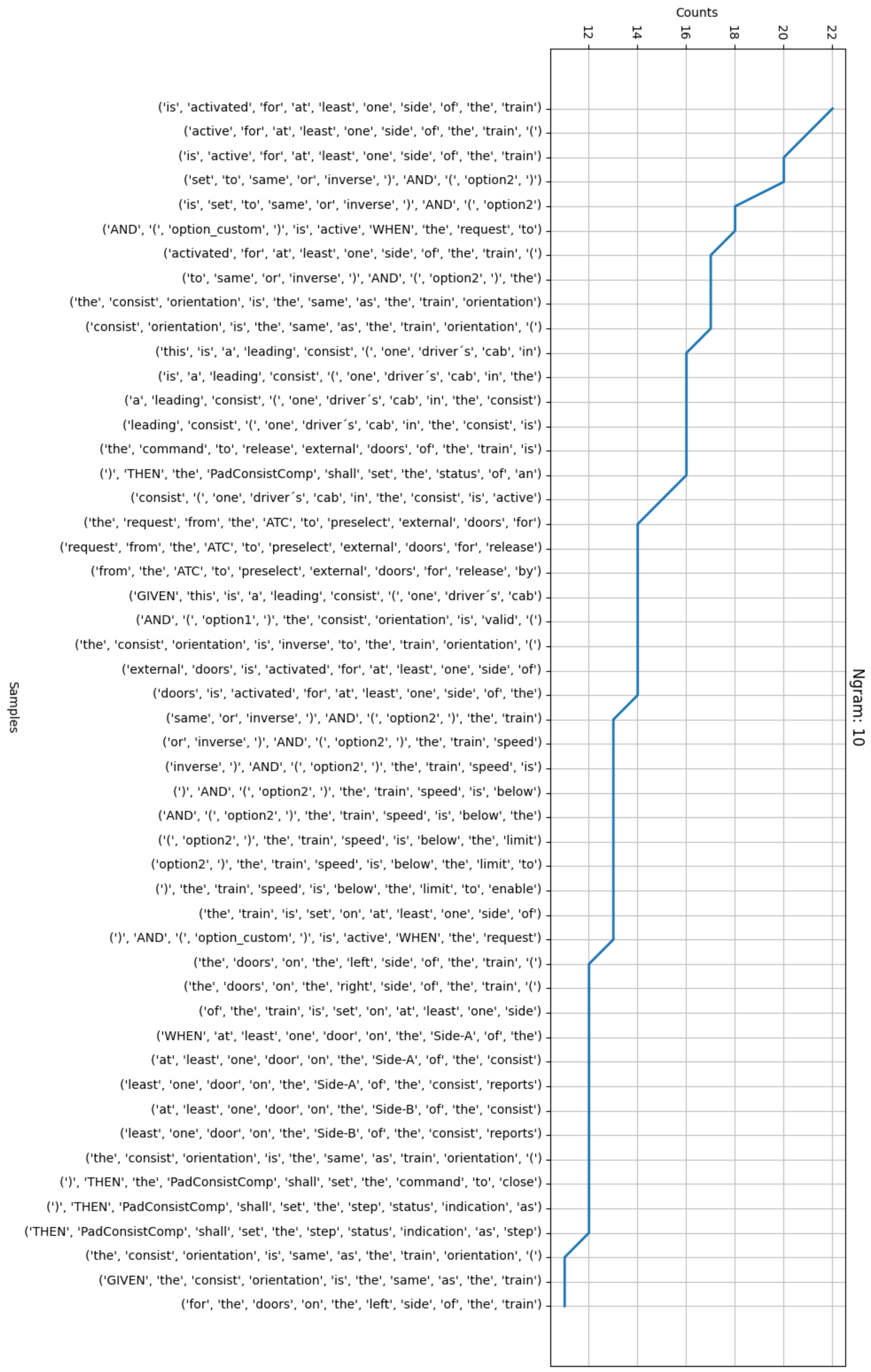


Figure 3.5: N-gram Model, N = 10

3.4 Software Detailed Design

This section provides a detailed description of the internal building blocks of *Sesnando* as well as its inputs, functionalities and outputs. These have been introduced in Section 3.1.

- Command Line Arguments (Section 3.4.1) - Describes the arguments that *Sesnando* supports from the command line.
- Input requirements (Section 3.4.2) - Presents examples of the requirements' structure that *Sesnando* supports.
- Requirements processing (Section 3.4.3) - Describes the compilation of the requirements into a Parse tree and how an XML sample can be obtained by using Debug mode.
- Grammar Elements (Section 3.4.4) - Presents the installed grammar on *Sesnando* and how requirements should be defined.
- Signal Mapping (Section 3.4.5) - Describes how Requirement signals are translated into Software signals.
- Test Generation (Section 3.4.6) - Describes how *Sesnando* compiles input requirements and how tests are generated on a design level.
- Signal Manager (Section 3.4.7) - Describes the main role of Signal manager on test generation and presents an overview of its user interface.
- Test Designer (Section 3.4.8) - Presents the test designer user interface, and how test specifications are visualised with the aid of this module.

3.4.1 Command Line Arguments

The entry point of *Sesnando* is a command line application. The command line arguments can be passed at the application launch, but a configuration file can be set, so that the user can define a set of default parameters avoiding the need to pass the arguments at all times. Default values from the configuration file will be used when omitting these parameters from the command line. The available command line arguments are defined on Table 3.1.

As *Sesnando* can be executed following a set of parameters from the command line or from a *JSON* configuration, a detailed description of these commands are as follows.

- signal_manager - The address of the remote Signal Manager server, containing the data required to successfully generate test cases.
- input - The location of the input file containing the requirements to be parsed by *Sesnando*.
- output_folder - The default location for the artifacts generated by *Sesnando* (Can be overriden using "Save as..." from the test designer).
- test_designer - The location of the Test Designer GUI module (usually on the same folder as the compiler binaries).
- debug - Debug mode mode allows a verbose execution of *Sesnando*.

Table 3.1: Sesnando Config parameters

Command	***Default Value***	***Description***
-sm –signal_ manager	127.0.0.1	Signal Manager IP Addr.
-i –input	./input_files/requirements.txt	Input Requirements location
-o –output_folder	./output_files/	Output Folder
-td –test_ designer	./SESNANDO.TestDesigner.exe	Test Designer Location
-d –debug	N/A	Debug mode

3.4.2 Input Requirements

At this stage of the development, *Sesnando* takes a set of input requirements written in a text file (.txt) and parses them from there. These requirements must be compliant with a predefined grammar. The grammar specification is presented on Section 3.4.4. The grammar guidelines [35] were documented by me and were then reviewed and approved by the Requirement Managers at CSW. Requirement 3.1 is a simple example of a generic requirement structure that can be parsed by *Sesnando*.

Requirement 3.1: Given-When-Then Blueprint

```
REQUIREMENT(REQ342-1, function, module)
{
      GIVEN <RequirementSignal1> is equal to true
         and <RequirementSignal2> is equal to true;
      WHEN;
      THEN <RequirementSignalX> is set to true;
}
```

GIVEN and *WHEN* predicates combined, dictate the outcome of the requirement through the fulfilment of the THEN clauses.

While Requirement 3.1 presents the requirement Given-When-Then Blueprint, Requirement 3.2 presents a concrete example that follows the structure presented in Requirement 3.1 by providing examples of possible Requirement Signals.

Requirement 3.2: Door Obstacle Alarm and CCTV

```
REQUIREMENT(REQ342-2, function, module)
{
      GIVEN <Status Door Closing> is equal to true
         and <Status of Obstacle Detection> is equal to true;
      WHEN;
      THEN <Door Obstacle Alarm and CCTV> is set to true;
}
```

As stated on the above requirement, when the GIVEN predicate evaluates to true, i.e, a Door is closing and an obstacle has been detected, the Train Control and Management System must set a CCTV signal to true (and generates a driver alarm), so the incident can be visualised by the train driver.

3.4.3 Requirement processing

From the requirement example at Section 3.4.2, *Sesnando* uses a lexer and a parser (ANTLR Library, Section 2.2) to translate it into a Parse tree. A parse tree is a concrete representation of the input and contains all the requirement information. Figure 3.6 is a representation of the Parsed Requirement 3.2.

In order to use *Sesnando*, such software requirements shall be written on a controlled natural language (CNL) (following the Given-When-Then blueprint presented on Section 3.4.2) that define the functionalities of a software system. This compiler aims to automatically generate system tests (Section 2.3) by parsing and interpreting requirement contents in this so called Parse tree.

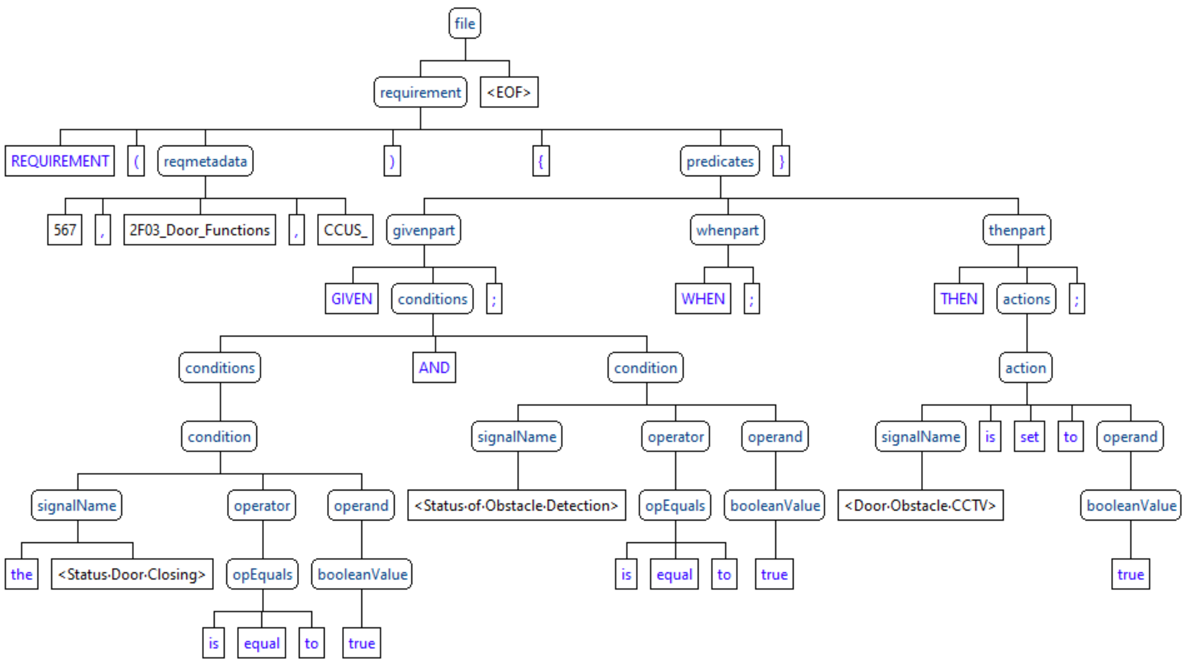


Figure 3.6: Parse tree from the Requirement 3.2

Before executing *Sesnando*, Debug mode can be enabled by setting the debug flag trough the command line. This turns *Sesnando* into a more verbose execution by displaying the intermediate steps to the user. Due to this, *Sesnando* provides a representation of the Parse tree in an XML format:

```
<?xml version="1.0" encoding="utf-8"?>
<AST>
  <RequirementSet>
    <Requirement>
      <Id>123456</Id>
      <TestCaseId>R151_2F03_DoorStat_TC_001</TestCaseId>
      <Namespace>MWT_</Namespace>
      <GivenConditions p4:type="ComparisonExpression">
        <ExpressionType>EXP_COMPARISON</ExpressionType>
        <ExpressionOperator>
          <ExpressionType>EXP_COMP_OPERATOR</ExpressionType>
          <OperatorType>OP_EQ</OperatorType>
        </ExpressionOperator>
        <LogicalSignal>
          <ExpressionType>EXP_SIGNAL</ExpressionType>
          <SignalName>CTC_OPDoorsFromMIO</SignalName>
          <SignalFormat>SIGNAL_ALIAS</SignalFormat>
        </LogicalSignal>
        <SignalValue>
        <ExpressionType>EXP_SIGNAL_VALUE</ExpressionType>
        <Type>SV_BOOL</Type>
        <Value>TRUE</Value>
      </SignalValue>
    </GivenConditions>
    <WhenConditions p4:type="EmptyLogicalExpression">
      <ExpressionType>EXP_EMPTY</ExpressionType>
    </WhenConditions>
    <ThenActions p4:type="ActionExpression">
      <ExpressionType>EXP_ACTION</ExpressionType>
      <LogicalSignal>
        <ExpressionType>EXP_SIGNAL</ExpressionType>
        <SignalName>CTC_STrcnSafeFromMIO</SignalName>
        <SignalFormat>SIGNAL_ALIAS</SignalFormat>
      </LogicalSignal>
      <SignalValue>
        <ExpressionType>EXP_SIGNAL_VALUE</ExpressionType>
        <Type>SV_BOOL</Type>
        <Value>TRUE</Value>
      </SignalValue>
      <Actor />
    </ThenActions>
  </Requirement>
 </RequirementSet>
</AST>
```

The above XML representation is a direct export of the Parse tree stored in Common Library (Common Lib) that can be generated by the ASTController.

The process of serializing an object's public properties and fields into a serial format (in this case, XML) for storage or transmission is known as XML serialization. In this case, this is achieved using the XmlSerializer class of .NET4.5.

The benchmarking results of this requirement interpretation is present in Annex A.

3.4.4 Grammar Elements

One of the core values of *Sesnando* is the ability to validate the writing of requirements given the installed grammar. This tool is able to display detailed error messages when it fails to interpret a given requirement as well as error messages when certain keywords or expressions are inadvisable.

Besides, the installed grammar tends to approach a level of natural language given that there is a tendency on these markets to describe high-level requirements using natural language as they are more readable by the stakeholders.

A requirement contains sets of conditions defined by GIVEN and WHEN elements and a set of actions defined by THEN element, these are defined as requirement predicates. GIVEN and WHEN define the outcome of the requirement (i.e. whether the THEN actions are applied or not) but are not mandatory, when omitted the outcome of the requirement is evaluated on the system with no restrictions.

A requirement clause is a single logical condition contained on each requirement GIVEN and WHEN predicate and may be separated from other clauses by boolean operators e.g. *AND*s. The most simple predicate (GIVEN or WHEN) clause contains two operands and one operator. The first operand is described as requirement signal and the second the signal value, which might be a boolean or an integer value. The most common operators are defined using natural language and are presented on Table 3.2

Table 3.2: Requirement grammar - Operators

Grammar Operator	***Logical Operator***
is equal to	=
is greater than	>
is greater than or equal to	<=
is lower than	<
is lower than or equal to	<=

As per Table 3.2, an example of a clause containing a natural operator, could be defined as: *GIVEN <Traction Safe Status>* ***is equal to*** *true in at least one DTCar in the train.*
This expression would check whether the requirement signal *Traction Safe Status* evaluates to true on a given car of the train. A train usually contains 5 coupled cars.

3.4.4.1 Quantifiers

Each requirement predicate condition can be enhanced with a quantifier. A quantifier is useful when a requirement signal represents more than one element on the train of the same type, e.g. a train door, a train axle, brake mechanisms, etc. A quantifier defines how many or which elements of the same type need to fulfil a condition in order for the full clause (i.e., the base condition plus the quantifier) to evaluate to true. Thus, a requirement signal can map to one or more technical signals. This will be presented on next Section 3.4.5.

The following, is a description of the Requirement 3.3. Right after, the actual requirement will be presented using the *Given-When-Then* (GWT) blueprint.

```
Given that a dragging brake is detected for at least one brake
in the Train Unit, Dragging Brake Detected shall be set on the
Juridical Recording Unit.
```

Requirement 3.3: JRU Dragging Brake Detected

```
1 REQUIREMENT(REQ344-1, 2F02_Traction_Braking, CCUS)
2 {
3       GIVEN <Dragging Brake Detected> is equal to true
4          for at least one brake in the Unit;
5       WHEN;
6       // Juridical Recording Unit
7       THEN <JRU Dragging Brake Detected> is set to true;
8 }
```

The above requirement written in the GWT blueprint defines that at least only one dragging brake is necessary for this event to be registered on the JRU. Line 6 of the second verbatim represents a comment that might be presented on the requirement to clarify any definition. This is supported by the installed grammar and will be ignored during the compilation of the requirement.

The complete list of the supported quantifiers is presented on the following bullet point list.

- **FOR ONE** - Predicate clause evaluates to true if one and only one component within the quantifier evaluates to true, e.g. one of the two driver cabinets of the train.
  - Example: <Status Train cab> is active **for one** cab of the train;

- **FOR AT LEAST** - Predicate clause evaluates to true if at least one component within the quantifier evaluates to true, e.g. a door in a set of train doors.
  - Example: <Status door> is open **for at least** one door of the train;

- **FOR ALL** - Predicate clause evaluates to true if all attributes/components within the quantifier evaluates to true, e.g. all the emergency brakes need to be applied in order for the clause to evaluate to true.
  - Example: <Status Emergency brake> is applied **for all** emergency brakes of the train;

- **FOR THE SAME** - When the value of an attribute of a technical signal, needs to be checked on the set of technical signals of another condition.
  - Example:
    WHEN <Driver Desk Door Open> is equal to true **for one** *side* of the train;
    THEN the <TCMS command door open> is set to true **for the same** *side* as in <Driver Desk Door Open>;

  <component> as in <Requirement Signal in FOR ONE quantifier>" - This quantifier must be used in conjunction with "for one" quantifier in the same requirement when both signals share the same attributes, i.e., the requirement signal within the clause containing the "for one" quantifier and the requirement signal within the "for the same..." quantifier. Predicate clause evaluates to true if the same attribute as in the "for one" clause contains the same attribute value. The following example describes a use case of this quantifier.

A requirement predicate supports multiple clauses and each clause supports only one quantifier. Each different quantifier is parsed into a different class object. Figure 3.7 presents the Requirement 3.4 containing a quantifier parsed into a Parse tree.

Requirement 3.4: JRU Dragging Brake Detected

```
REQUIREMENT(REQ344-2, 2F03_Door_Functions, CCUS)
{
      GIVEN <Status of Obstacle Detection> is equal to
         true for at least one door of the train;
      WHEN;
      THEN <Door Obstacle alarm and CCTV> is set to true;
}
```

On Figure 3.7, for the quantifier node, "for at least one" defines the type of the quantifier. *Door* is the component of the signal and the train is the scope of the signal (i.e., local car, unit, or the whole train), meaning that each Door needs to be

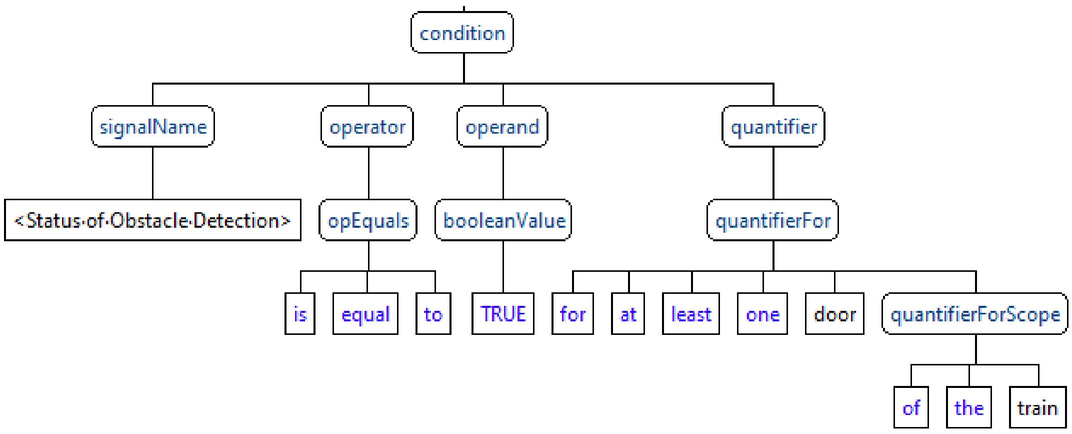


Figure 3.7: Requirement Quantifier Parse tree

checked whether the present conditions evaluates to true for at least one door of the train (i.e., an obstacle has been detected during door close sequence).

3.4.4.2 Technical approach for grammar definition

As the grammar lexer set has been identified it was then necessary to define an Abstract Syntax tree (AST). It is known that *GIVEN*, *WHEN* and *THEN* are the main predicates that define the skeleton of a requirement, thus, each predicate should implement its own Condition tree.

There are widely known libraries to support the building of a grammar, such as Antlr, Flex and Bison [13], however, flex and bison only work with the C++ language and Antlr works with a number of different languages [8], hence the choice of *Antlr* library to support the development of this grammar. *GIVEN*, *WHEN* and *THEN* are represented in the form of a Logical Expression that can be derived into child classes, given the lexer type. The representation of the syntax tree is presented on Figure 3.8.

The AST on Figure 3.8 implements a list of requirements that *Sesnando* has compiled from the input file. The type of each expression is given by the ELogicalExpressionType enumerator. *AndExpression* and *OrExpression* are the defined boolean operators, however, *Sesnando* will report a warning message when an "OR" is used, as it usage is not advised according to Railway original requirement guidelines. A *ComparisonExpression* is used for *GIVEN* and *WHEN* predicates and the *ActionExpression* on *THEN* predicates, as they define the output actions of a requirement, i.e., the expected results.

The defined grammar also supports the use of comments in the input file in the format of line comment or block-comment, "//" and "/**/" respectively.

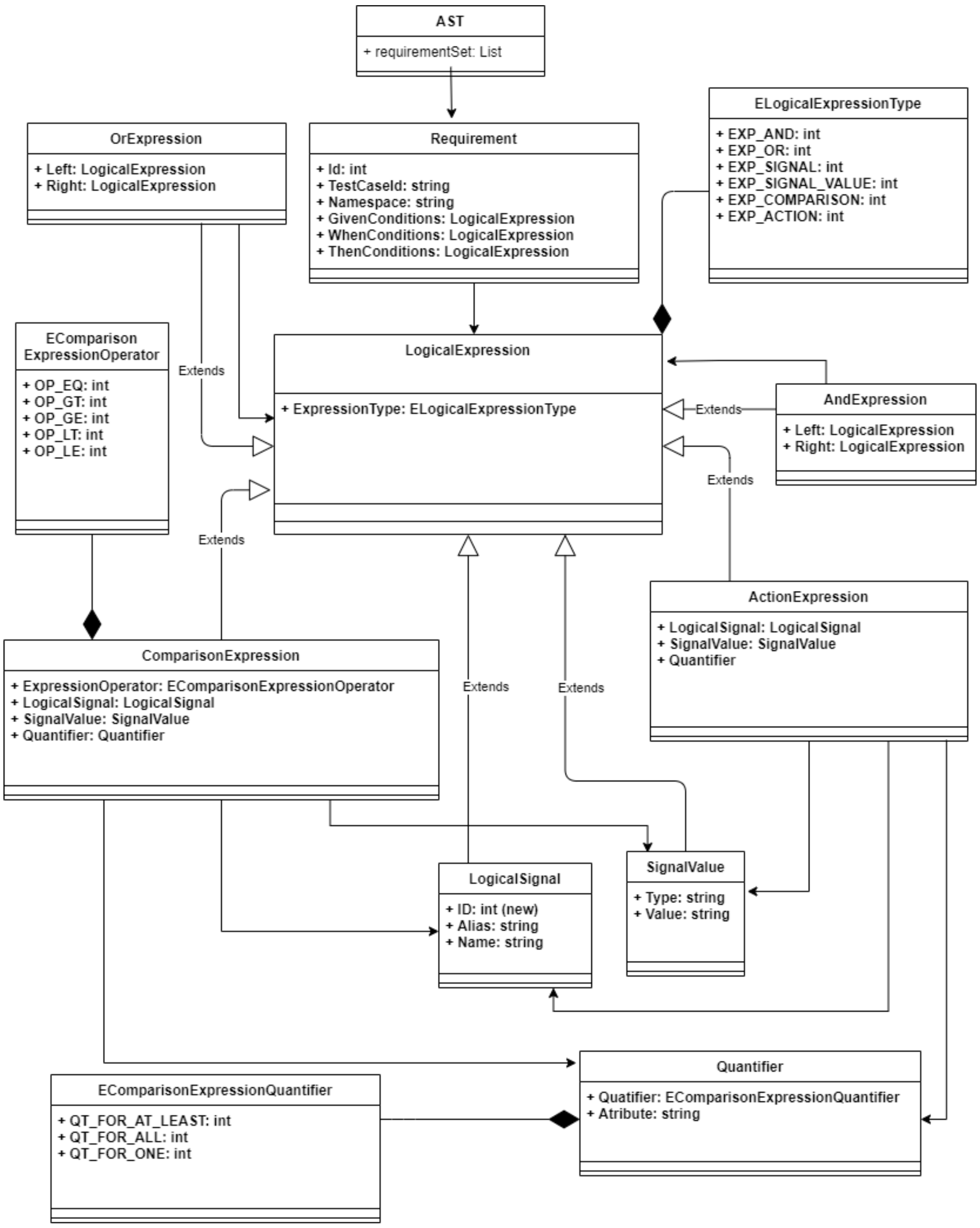


Figure 3.8: AST Class Diagram

3.4.5 Signal Mapping

Signal Mapping is the activity of identifying the required software signals used for testing from the signal names present on the Requirements. This functionality is integrated on the Test Generator module as presented on Figure 3.3.

Requirement signals are from now on defined as Logical signals, as per naming conventions by the Railway project stakeholders.

A Logical signal maps to one or more Software signals (defined as Technical signals as per naming conventions) Figure 3.9.

Technical signals can be seen as software variables that are being used in traditional programming. The most common variable types are boolean and integer values that define the status of a system or sub-system of the train and these variable values change while the train is in operation.

The Train control and Management system (TCMS) acts as the brain of the train and keeps track of all the remaining device status, e.g. Brake status, Door status, whether the train is at a station or not, whether it is being energised by the external network, the status of propulsion systems, converters and transformers, whether there is a fire on the train, etc.

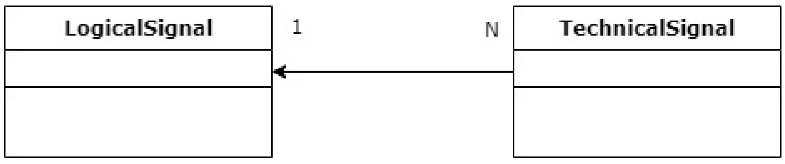


Figure 3.9: LogicalSignal and TechnicalSignal class relationship

Expanding on this subject, Requirement 3.5 is given as example.

Requirement 3.5: Traction Safe Status

```
REQUIREMENT(REQ345-1, 2F03_Door_Functions, CCUS)
{
      GIVEN <Doors enabled> is equal to false
            and <Train speed> is equal to 0;
      WHEN;
      THEN <Traction Safe> is set to true;
}
```

Regarding Requirement 3.5, a signal within a clause evaluating the train speed (Logical signal: <Train Speed>), would map to one technical signal (BUS.ETH_-1.trainStatus.TrainSpeed) of the system, as the train speed is an atomic and singleton value throughout the train (i.e., it is not possible to have multiple train speeds).

However, when checking whether the doors are enabled (Logical signal <Doors enabled> is equal to false), checkings need to be performed for each door on the train. In other words, if a train contains 10 doors, this requirement signal would actually map to as many signals as needed to represent all the door status.

In order to determinate the status of the doors of the train, a collection of requirements are necessary containing the conditions for each state. If any door is open, the train door status is set as *Open*, thus, when all doors are closed, the train door status is set as *Closed*. This is the role of quantifiers, as presented on Section 3.4.4.1.

The signal mapping of these two requirement signals is presented on Table 3.3.

Table 3.3: Signal mapping of train speed and door status

Logical Signal	***Technical Signal***
<Train speed>	BUS.ETH_1.trainStatus.TrainSpeed
<Doors Enabled>	BUS.ETH_1.DCUStatus.C1ISDOEnCD BUS.ETH_1.DCUStatus.C1ISDOEnAF BUS.ETH_1.DCUStatus.C2ISDOEnCD BUS.ETH_1.DCUStatus.C2ISDOEnAF BUS.ETH_1.DCUStatus.C3ISDOEnCD BUS.ETH_1.DCUStatus.C3ISDOEnAF BUS.ETH_1.DCUStatus.C4ISDOEnCD BUS.ETH_1.DCUStatus.C4ISDOEnAF BUS.ETH_1.DCUStatus.C5ISDOEnCD BUS.ETH_1.DCUStatus.C5ISDOEnAF

Requirement 3.5 expresses the use of the signals previously mentioned. As per Table 3.3, each technical signal mapping to <Doors Enabled> provides the status of all doors throughout the train. It is also important to mention that when omitting any type of quantifier, the default behavior is the same as "for all" quantifier, hence, for the verification of each door that maps to the requirement signal, the quantifier can act as a filter of the signals that go into the verification.

The corresponding signals are obtained by sending a request to the Signal Manager API (3.4.7), which is a centralised knowledge base that supports the information present on a requirement with additional data. This will be discussed in detail on next Section.

Once the technical signals that are used for testing are identified, *Sesnando* starts to generate tests to verify the requirement.

3.4.6 Test Generation

Test Generator has two main roles: Translate each requirement signal into one or more software signals, and generate the combinatory explosion from the input signal values into test cases. This will be detailed throughout this section.

Once a requirement is compiled into a Parse tree, Test Generator module firstly picks up the *GIVEN* and *WHEN* clauses. The evaluation of these combined predicates (evaluation of GIVEN conditions AND WHEN conditions) dictates the outcome of the requirement (decision), in other words, dictates how the set of the *THEN* actions, or the effects on the train system should be observed (Figure 3.10). These effects are defined as expected results. When both predicates evaluate to True, the *THEN* actions defined by the requirement should be observed on the system, otherwise, they should be observed in its negative form or unrealised.

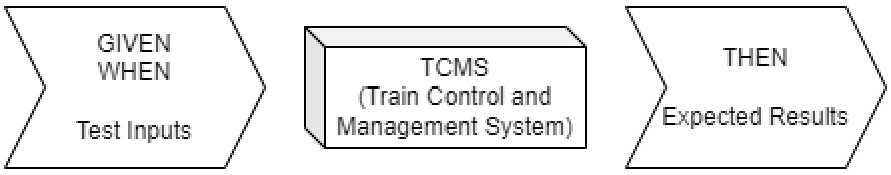


Figure 3.10: Test Inputs and Expected Results

3.4.6.1 Requirement coverage and applications of MCDC

Generated tests shall follow the the MCDC (Modified Condition/Decision Coverage) coverage criteria. MCDC is used in critical software development to ensure adequate testing.

Essentially, it states that every condition in a decision should independently affect the outcome of a decision and every decision should take every possible outcome. In the context of the railway requirements of this project, it means that every clause/condition within a GIVEN and WHEN that affects the outcome of the requirement should be exercised with its possible inputs.

Considering the presented constrains, the outcome of a requirement decision is given by the Equation 3.1.

$$\text{THEN Actions} = \text{GIVEN Predicate} \circ \text{WHEN Predicate} \tag{3.1}$$

The *THEN* actions (expected results) shall be observed on the system under testing if *GIVEN* and *WHEN* predicates combined evaluate to True. Equation 3.2 presents how GIVEN and WHEN Predicates are determined.

$$\text{Predicate Outcome} = \text{Clause1} \wedge \text{Clause2} \wedge \text{ClauseN} \tag{3.2}$$

Previous Equations will be demonstrated by using the Requirement 3.6 blueprint as example, containing one *GIVEN* **AND** operator and a single *WHEN* clause, which results in a set of tests presented on Table 3.4.

Requirement 3.6: Requirement Signal Structure

```
REQUIREMENT(REQ346-1, function, tcmsdevice)
{
      GIVEN <SignalIn1> is equal to true
            and <SignalIn2> is equal to true;
      WHEN <SignalIn3> is equal to true;
      THEN <SignalOut1> is set to true;
}
```

Regarding Requirement 3.6, SignalIn(1,2,3) will be set on the system as True (as dictated by the requirement signal values SignalInX), whereas SignalOut1 will be read from the system and compared against expected results provided on Table 3.4.

Requirement Logical expression:
signalOut1 = SignalIn1 AND SignalIn2 AND SignalIn3

Table 3.4: Test Spec for single AND

Cell values: 0 = False, 1 = True

Test case	***SignalIn1***	***SignalIn2***	***SignalIn3***	***SignalOut1***
1	0	1	1	0
2	1	0	1	0
3	1	1	0	0
4	1	1	1	1

It can be seen on Table 3.4 that when the GIVEN and WHEN conditions evaluate to true, the expected result (signalOut1) is expected to to be observed on the system as True.

It is important to mention that this requirement compiler supports predicate clauses separated by **AND** boolean operators only. The reason is that the manufacturer prohibits the use of **OR** operators on system requirements by stating the following on document [1]: *"Multiple conditions shall be linked only by AND, i.e. OR conditions are forbidden."*.

The use of *Sesnando* on a different manufacturer project where Requirements with multiple boolean logic operator are used, require further developments on

Sesnando tool. This approach might include the reduction and simplification of the input logical expression and the application of a newly developed algorithm. These techniques have already been explored by this Github user (armin-montigny) [36]. However, **OR** operators are implicitly present on element quantifiers as described on next Section 3.4.6.2.

3.4.6.2 Handling quantifiers on requirements

Previous Section 3.4.6.1 introduces the handling of Logical Signals from a compiled requirement. A Logical signal maps to 1 to N technical signals and this approach is detailed on Section 3.4.5.

Quantifiers define how to handle the set of technical signals that a Logical Signal maps to. For instance, by observing the following clause:

```
<Brake Applied> is equal to true
```

Being <Brake Applied> a Logical Signal mapping to 4 individual brake technical signals, one might ask: "How many brakes should be applied to fulfill the condition? Should all train brakes be applied? Is only 1 brake applied enough?" The answer to these questions are given by the quantifiers.

To better understand this concept, a real world scenario will be presented. Train Control and Management System (TCMS) has a functionality to detect whether a brake failed to release whilst at speed. When it happens, this event is recorded on the Juridical Recording Unit, commonly known as black-box. This functionality is described by Requirement 3.7.

Requirement 3.7: For at least one Brake quantifier

```
REQUIREMENT(REQ346-2, 2F02_Traction_Braking, CCUS)
{
    GIVEN <Train speed> is greater than 3 kph
       and <Brake Applied> is equal to true
       for at least one brake in the Unit;
    WHEN;
    // Comment: JRU - Juridical Recording Unit
    THEN <JRU Dragging Brake Detected> is set to true;
}
```

Requirement 3.7 explicitly indicates by the *for at least one* quantifier that if **ANY** brake is applied whilst at speed a *Dragging Brake Detected* event is recorded on the train. In order to register this event, it is necessary to actually verify that the train is at Speed (greater than 3 kph) and there is at least one brake applied. This is illustrated by the cause-effect-graph [37] on Figure 3.11.

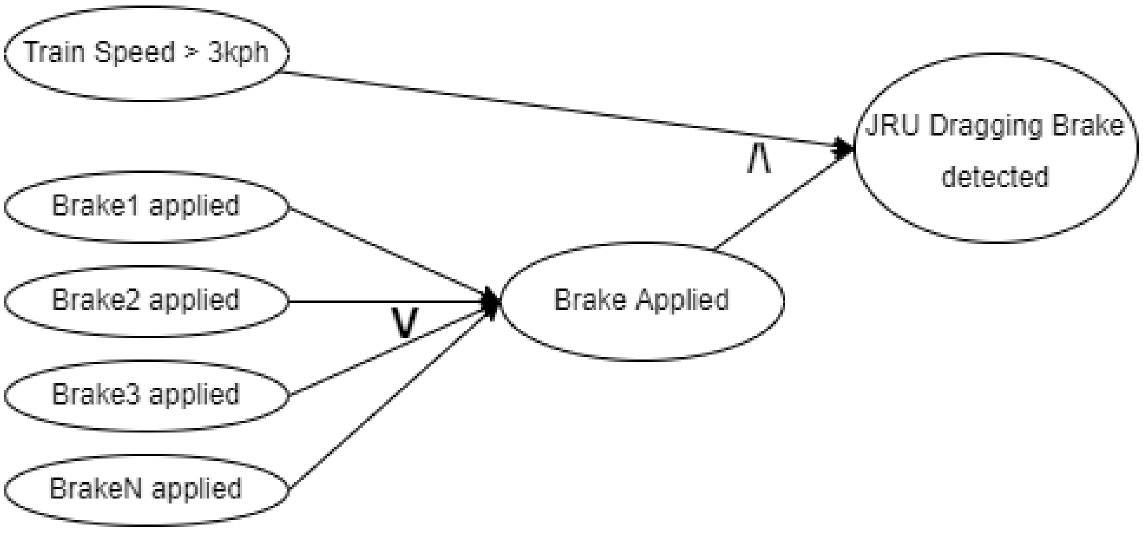


Figure 3.11: Cause-Effect-Graph of Dragging Brake detection

The requirement Requirement 3.7 contains two signals whereas the <Train speed> maps to only one technical signal and the <Brake Applied> Logical signal maps to 4 Technical signals. The clause containing the <Train speed> evaluates to true if the speed is above 3 kph, so value 4 is used for the cases where the clause evaluates to true, and the value 3 is used when the clause should evaluate to false.

Test Generator proceeds to ask the Signal Manager remote service (Section 3.4.7) for the corresponding technical signals (Section 3.3) for every Logical Signal in Requirement 3.7.

On a detailed technical point-of-view, the Test Generator extracts all the Requirement signals by traversing the Parsed Object Tree of the requirement (See Figure 3.6) and for each Logical Signal found, requests the corresponding Technical Signals from the Signal Manager calling an API endpoint (API/signal/LOGICAL_SIGNAL_NAME) where LOGICAL_SIGNAL_NAME is the actual Requirement Signal name from the input requirement.

For instance, on the <Brake Applied> request, Signal Manager returns 4 corresponding technical signals. Train Speed and the Event of dragging brake detected are unique signals, thus, they map to only 1 signal each.

The mapping of these Logical signals and their respective Technical Signals is presented on Table 3.5.

Table 3.5: Signal Mapping for Dragging Brake detected

Logical Signal	***Technical Signals***	***Alias***
Train Speed	BUS.ETH_1.trainStatus.TrainSpeed	TSpeed
Brake Applied	BUS.ETH_1.BkStatus1.Applied BUS.ETH_1.BkStatus2.Applied BUS.ETH_1.BkStatus3.Applied BUS.ETH_1.BkStatus4.Applied	Bk1 Bk2 Bk3 Bk4
JRU Dragging Brake Detected	BUS.ETH_1.JRU.BkDrgDetected	JRU DBkDet

Obs.: BUS.ETH terminology means that the software Technical Signal can be read and set by connecting to the system via an ethernet port, but this feature is not relevant for the scope of this project.

The Logical expression resulting from the Requirement 3.7 is (Eq. 3.3):

$$JRUDraggingBrakeDetected = TrainSpeed > 3kph \wedge BrakeApplied \tag{3.3}$$

Whereas, once the technical signals are identified, it turns into (Eq. 3.4):

$$JRUDBkDet = TSpeed > 3kph \wedge (Bk1 \vee Bk2 \vee Bk3 \vee Bk4) \tag{3.4}$$

What initially was a simple AND expression, turned into a more complex expression due to quantifiers. Before generating the Test Specification for Logical Expression in Eq. 3.4, the Test Generator evaluates all the possible outcomes for each requirement clause. This is presented by Table 3.6.

Table 3.6: Dragging Brake detection - Evaluation Table

Logical Signal	***JRU DBkDet = True***	***JRU DBkDet = False***
Train Speed	>3 kph	<= 3kph
Brake Applied	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} = \text{True}$ 4 Signals Brake Applied is **true** if: BrakeApplied$[i] = True \quad \forall i \in \{1,2,3,4\}$	$\begin{pmatrix}0&0&0&0\end{pmatrix} = False$

As per MCDC, every input condition in a decision shall generate every possible outcome. As such, given that one requirement signal might map into multiple technical signals, every technical signal must be tested (as per the previous example, every brake on the train). Meaning that the logical expression on Eq. 3.4 results in the test specification of Table 3.7.

Table 3.7: Test Spec Dragging Brake Detected

Test Case	***TS***	***Bk1***	***Bk2***	***Bk3***	***Bk4***	***JRU DBkDet***
1	4	0	0	0	0	0
2	3	1	0	0	0	0
3	4	1	0	0	0	1
4	4	0	1	0	0	1
5	4	0	0	1	0	1
6	4	0	0	0	1	1

On Table 3.7, and regarding the Requirement 3.7 it is verified that:

- Test Case 1 - Requirement decision outcome is False when Brakes are not applied.
- Test Case 2 - Requirement decision outcome is False when the train is not above the speed threshold (>3kph).
- Test Case 3-6 - Requirement decision outcome is True when both conditions are True.

When the *for-at-least-one* quantifier is used on a Logical expression, the inner Technical signals (ts) that map to the Logical signal is given by 3.5:

$$\text{Logical Signal on for-at-least-one (OR)} = (ts_1 \vee ts_2 \vee ts_3 \vee ts_4) \tag{3.5}$$

The applicability of this equation can be seen on Eq. 3.4.

However, if for instance, the Brake Applied condition quantifier mandates that all brakes must be Applied using the *for-all* quantifier as in:

```
<Brake Applied> is equal to true for all brakes in the Unit;
```

Applying the same principle, the Brake Applied Technical signals inner condition expression would be given by Eq. 3.6.

$$\text{Logical Signal on for-all (AND)} = (ts_1 \wedge ts_2 \wedge ts_3 \wedge ts_4) \tag{3.6}$$

As such, with regard to the variables used in Eq. 3.4, a *for-all* quantifier would result in the extended Logical Expression on Eq. 3.7.

$$JRUDBkDet = TSpeed > 3kph \wedge (Bk1 \wedge Bk2 \wedge Bk3 \wedge Bk4) \tag{3.7}$$

This results in the Test Specification of Table 3.8.

Table 3.8: Test Spec Dragging Brake Detected

All brakes applied whilst at speed

Test Case	*TS*	*Bk1*	*Bk2*	*Bk3*	*Bk4*	*JRU DBkDet*
1	3	1	1	1	1	0
2	4	0	1	1	1	0
3	4	1	0	1	1	0
4	4	1	1	0	1	0
5	4	1	1	1	0	0
6	4	1	1	1	1	1

When a Requirement Signal maps to multiple technical signals, and in order to exercise every possible outcome of a requirement decision, *Sesnando* expands every requirement Logical Signal into (No. Technical Signals + 1) test cases. This scenario is detailed on Table 3.6 and can also be observed on the Test Specifications on Table 3.7 and Table 3.8. The process to create these Test Specifications is automated within the Test Generator module of *Sesnando*.

3.4.6.3 Quantifiable attributes on requirements

Sesnando extracts every Logical Signal from the input requirements and asks the Signal manager (Section 3.4.7) for their corresponding Technical signals (Section Section 3.3).

A Logical Signal quantifier defines how these technical signals are tested (from the Test Generator perspective), and when applied to a train element (Doors, Brakes, Axles). Then *Sesnando* tests these components individually as seen on previous Section 3.4.6.2 on the Brake Applied example.

For instance, a requirement that states that all doors shall be closed, would state the following:

```
<Door Closed> is equal to true for all doors in the Train;
```

However, there might be some cases where a subset of technical signals fulfill a requirement condition, for instance, the doors on one side of the train. This can be put as:

```
<Door Closed> is equal to true for one SIDE of the train;
```

Requirement 3.8 illustrates this scenario. It states that, given the train doors are allowed to be open for one side, when the driver presses the Door Open push-button on Cabin Desk for the same side, the Train Control and Management System (TCMS) sends a command throughout the train for the doors on the same side.

Requirement 3.8: TCMS command door open

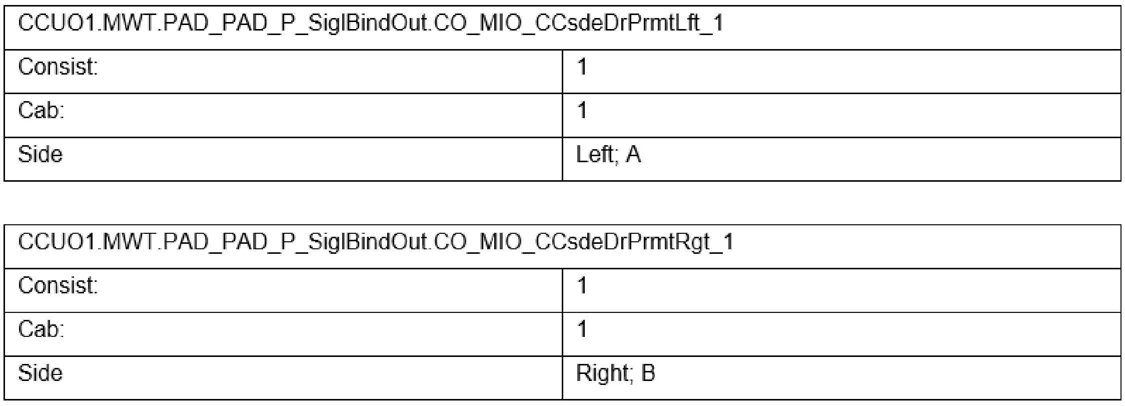
```
REQUIREMENT(REQ346-3, function, module)
{
      GIVEN <Door Open Permit> is equal to true for one side;
      WHEN <Driver Desk Door Open> is equal to true
         for the same side;
      THEN <TCMS command door open> is set to true
         for the same side;
}
```

Naturally, *Sesnando* will set all the Technical signals, such as Door Open Permit, in order to exercise every possible requirement outcome. However, on the field, that signal is set according to the train and location circumstances (i.e., is set from other requirements).

Expanding on this subject, *Door Open Permit* technical signals must have the attribute **side** assigned on the Signal Manager database. This is how Test Generator handles the sorted signals by the signals attribute (side). The technical signals mapping to *Door Open Permit* logical signal are the following:

```
MWT.PAD_PAD_P_SiglBindout.CO_MIO_CCsdeDrPrmtLft_1
MWT.PAD_PAD_P_SiglBindout.CO_MIO_CCsdeDrPrmtRgt_1
```

The mapping from Logical Signals to Technical Signals as well as their attributes can be found on the railway project documentation (Interface Control Documents (ICDs) and Interface Signal List (ISL)) and are then populated on Signal Manager Server to support the generation of tests by *Sesnando*. Figure 3.12 presents 3 attributes that have been assigned for each technical signal of *Door Open Permit* Logical signal.



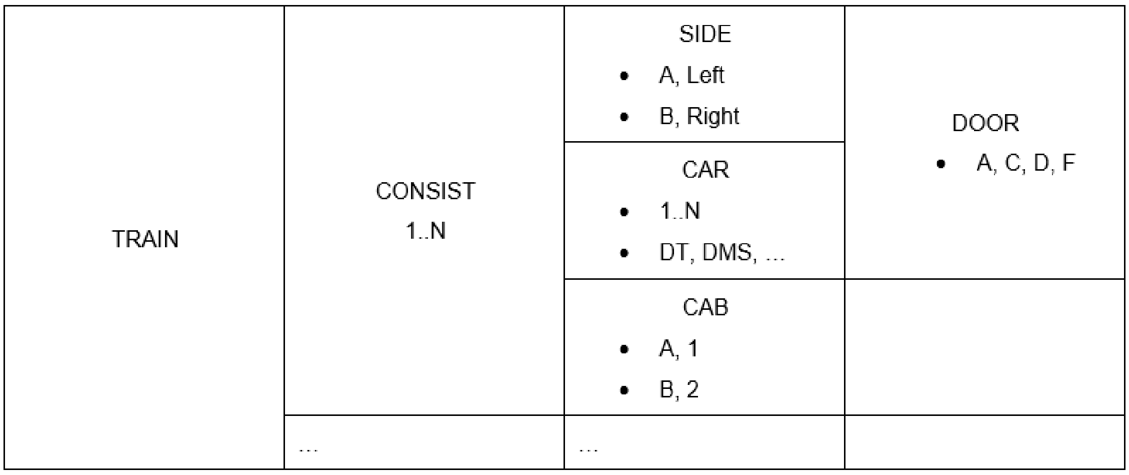
CCUO1.MWT.PAD_PAD_P_SiglBindOut.CO_MIO_CCsdeDrPrmtLft_1	
Consist:	1
Cab:	1
Side	Left; A

CCUO1.MWT.PAD_PAD_P_SiglBindOut.CO_MIO_CCsdeDrPrmtRgt_1	
Consist:	1
Cab:	1
Side	Right; B

Figure 3.12: Door open permit signals attributes

Meaning that the first Technical Signal defines the permission to open the doors on the left side and the second signal on the right side.

From a business perspective these attributes can be arranged on an hierarchical way. An excerpt of this hierarchy is presented on Figure 3.13. The illustrated designations are detailed on Chapter 2.



TRAIN	CONSIST 1..N	SIDE • A, Left • B, Right	DOOR • A, C, D, F
		CAR • 1..N • DT, DMS, ...	
		CAB • A, 1 • B, 2	
	...	...	

Figure 3.13: Table train elements and attributes (excerpt)

Regarding Figure 3.13, one Train contains at least 1 Consist (also called Unit) and a Consist contains Cars and Cabins (Cabs). DT, DMS, etc., are cars identifiers and not relevant for this example. Thus, if a requirement states the following quantifier:

```
<Door closed and Locked> for all Cabs on the train;
```

only the doors within train cabins are considered on the requirement.

As such, and returning to the example Requirement 3.8 in the beginning of this section, Test Generator generates tests for Left and Right side of the train, resulting in the logical expressions in Eq. 3.9 and Eq. 3.10 respectively.

Given that the "same side" refers to the side where the Door Open Permit is equal to true, the MCDC test cases for the given requirement would be as follows:

- GIVEN: Door Open Permit = true, WHEN: Driver Desk Door Open = true THEN: TCMS command door open = true
- GIVEN: Door Open Permit = true, WHEN: Driver Desk Door Open = false THEN: TCMS command door open = false
- GIVEN: Door Open Permit = false, WHEN: Driver Desk Door Open = true THEN: TCMS command door open = false

These test cases ensure that the TCMS command door open is set to true only when the Door Open Permit is true and the Driver Desk Door Open is also true for the same side. The Boolean expression of this statement is given as:

$$\text{TCMS command door open} = DoorOpenPermit \wedge DriverDeskDoorOpen \quad (3.8)$$

This can be illustrated on the Table 3.9:
Being L and R the left and the right side respectively

Table 3.9: Test Spec - for the same side quantifier

Test case	***DrPm[L,R]***	***DrDsk[L,R]***	***DrCmd[L,R]***
1	0	1	0
2	1	0	0
3	1	1	1

When mapping previous Logical signal signals to their Technical signals, Table 3.9 is expanded to cover the following expressions applied to Technical Signals:

$$DrCmdLft = DrPmLft \wedge DrDskLft \quad (3.9)$$

$$DrCmdRgt = DrPmRgt \wedge DrDskRgt \quad (3.10)$$

Where,

- DrPmLft and DrPmRgt are Technical Signal aliases representing <Door Open Permit> Logical signal for Door Open Permission;
- DrDskLft and DrDskRgt are Technical Signal aliases representing the <Driver Desk Door Open> the Logical signal for the Driver Door open command;

- DrCmdLft and DrCmdRgt are Technical Signal aliases representing the <TCMS command door open> Logical Signal representing the Command for Door opening which are the Expected results of the Requirement 3.8.

Requirement 3.8 results in the Test Specification of Table 3.10.

Table 3.10: Test Spec - for the same side quantifier

Test case	***DrPmLft***	***DrDskLft***	***DrPmRgt***	***DrDskRgt***	***DrCmdLft***	***DrCmdRgt***
1	0	1	0	1	0	0
2	1	0	0	1	0	0
3	1	1	0	1	1	0
4	0	1	1	0	0	0
5	0	1	1	1	0	1

Referring to Table 3.10, DrCmdLft is expected to be set as 1 (True) when the left doors are allowed to be open and the Driver presses the Door Open button for the Left side on the Cab desk, otherwise, the door open command is not expected to be set on the TCMS. This approach also applies to the doors on the Right side of the train.

This test table covers all possible combinations of the input variables, DrPmLft, DrPmRgt, DrDskLft, and DrDskRgt, and ensures that the output variables, DrCmdLft and DrCmdRgt, are set to true only when the corresponding Door Open Permit and Driver Desk Door Open variables are true for the same side.

3.4.7 Signal Manager

Signal Manager runs on a remote server where additional signal data is stored. The main benefit of this centralized data approach is that every tester, developer or project stakeholder can contribute to develop a solid knowledge-base to sustain the test generation process of *Sesnando*.

As mentioned on the previous sections, the Test Generator asks the Signal Manager service for the technical signals that map to a given requirements' Logical Signal. This is achieved by calling a Signal Manager API endpoint, such as:
https://<IP_Address>:5001/api/signal/<LogicalSignal>

3.4.7.1 Signal Manager Interface

Signal Manager can be accessed trough a Web App Interface running on port 5001. From there, the user is able to edit Technical signals (Software signals) which can be mapped to the corresponding Logical Signals (Requirement Signals) as well as add relevant data, like attributes and states.

To introduce the Signal Manager interface, the following use case scenario can be given as an example: When a *Logical signal* is not found on the Signal Manager Database, the Test Generator reports an error message to the user regarding the Logical Signal, as it must be added to the Database. The execution halts and no Test Specification is generated.

This process will be demonstrated using Requirement 3.9 as example:

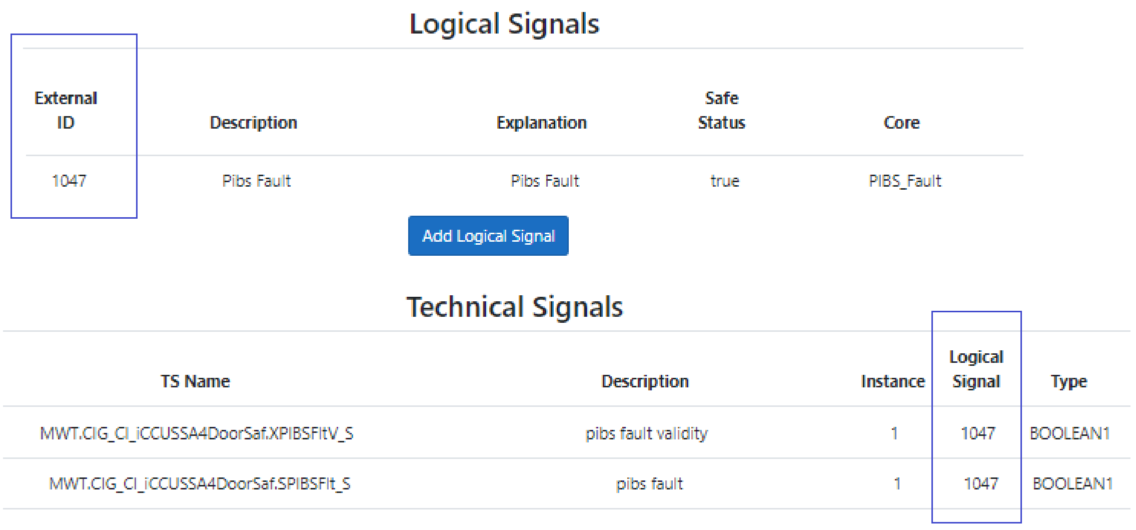
Requirement 3.9: Evt Loss of CSDE Protection

```
REQUIREMENT(REQ347, function, doors_module)
{
   // Passenger Information and Beacon System
      GIVEN <PIBS Fault> is equal to true;
         AND <PIBS Isolated> is equal to false;
      THEN <Evt Loss of CSDE Protection> is set to true;
}
```

Assuming *PIBS Fault* Logical Signal is not found on the Signal Manager database, Test Generator would return the following error message:

```
Error while fetching Logical Signal from Signal Manager: PIBS Fault
```

To overcome this problem, this signal needs to be mapped on the Signal Manager. See (Fig. 3.14)



Logical Signals

External ID	Description	Explanation	Safe Status	Core
1047	Pibs Fault	Pibs Fault	true	PIBS_Fault

Technical Signals

TS Name	Description	Instance	Logical Signal	Type
MWT.CIG_CI_iCCUSSA4DoorSaf.XPIBSFltV_S	pibs fault validity	1	1047	BOOLEAN1
MWT.CIG_CI_iCCUSSA4DoorSaf.SPIBSFlt_S	pibs fault	1	1047	BOOLEAN1

Figure 3.14: Signal Manager PIBS Signal Mapping example

Figure 3.14 is an excerpt from the *Signal Manager* user interface. Logical Signals can be added by pressing **Add Logical Signal**. Once *PIBS Fault* is added, it is identified by the id **1047** which is a foreign key on the Technical Signals database, meaning that it maps to two technical signals as highlighted on Figure 3.14.

Every database change will be recorded on a Log table that can be audited if necessary.

Signal Manager also supports the upload of XLSX spreadsheets containing the mapping data of Logical Signals and Technical Signals.

3.4.7.2 Signal Manager Architecture

As previously stated, *Signal Manager* runs as a detached service from the local compiler solution.

Since the information is centralized on a remote server, this would allow that every tester, developer or involved user could contribute to the growth of the information in it. This benefits all the users of *Sesnando* as all the local client instances would make use of this information to generate tests. Meaning that every piece of testing information such as signal mapping, attributes, etc., would be added only once.

The *Signal Manager* acts as a Web Service and serves the Test Generator through a REST API. There are several endpoints to consult, edit and create additional signals on the Signal Manager Database.

The purpose of these API Endpoints is not only to support the Test Generator, but also the management of the signal data without the need of accessing the web user interface, e.g, automating the import of signal data by an external client or tool.

The *Signal Manager* contains 14 database tables to support the logic described throughout this document. An excerpt of this database is presented on Fig. 3.15.

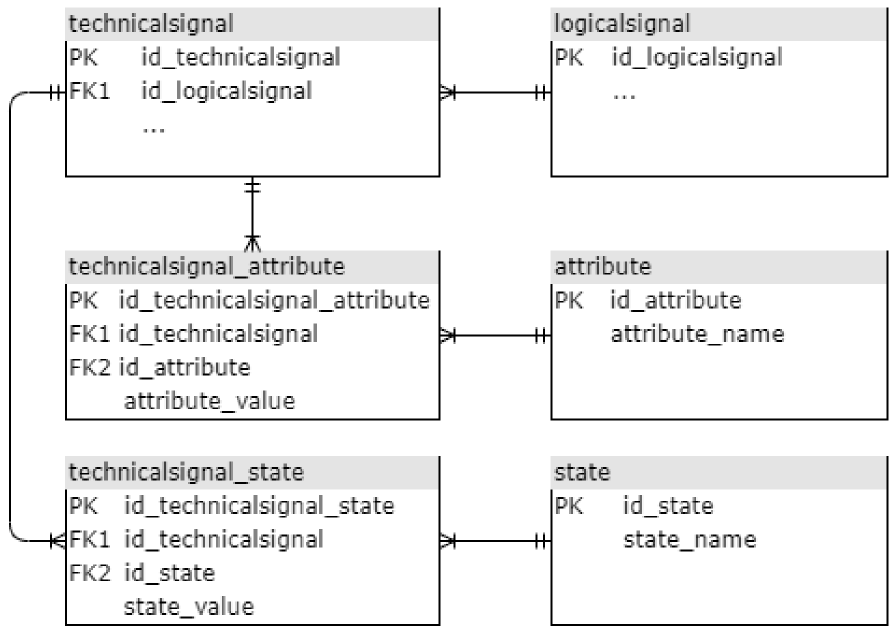


Figure 3.15: Signal Manager DB Architecture (Excerpt)

A Logical Signal maps to one or more technical signals and a technical signal supports multiple attributes and states.

A train Brake can be used as an example, as a possible attribute would be the location of this Brake within the train (Car=1,2..N) and its Train Axle (Axle=1..2), given that a train Car contains two axles and a possible state would be whether this brakes are released or not (state="released", "not released").

Technical Signal states map to an integer value to represent the state. This information is cross-checked on the Test Generator to process the correct technical signals values according to the requirement information.

For instance, the Technical Signal states are used to mask Logical Signal values such as:

```
<Brake Status> is equal to Released
```

It can be seen that Technical Signals map to a Logical Signal trough a database id. A Technical signal might map (not mandatory) to several attributes, e.g., a door *Side* or Car location within the train or a state, e.g., Open, Released, Closed. The same way, a given state or attribute can be present on multiple Technical Signals (N-to-N relationship). Technical Signal states is a feature under development. It is fully supported by the Signal Manager, but not yet by Test Generator.

3.4.8 Test Designer

The *Test Designer* is a Graphical User Interface (GUI) module of *Sesnando* that displays the generated Test Specifications to the user. This section presents an example of a generated specification of a requirement.

3.4.8.1 Test Designer Interface

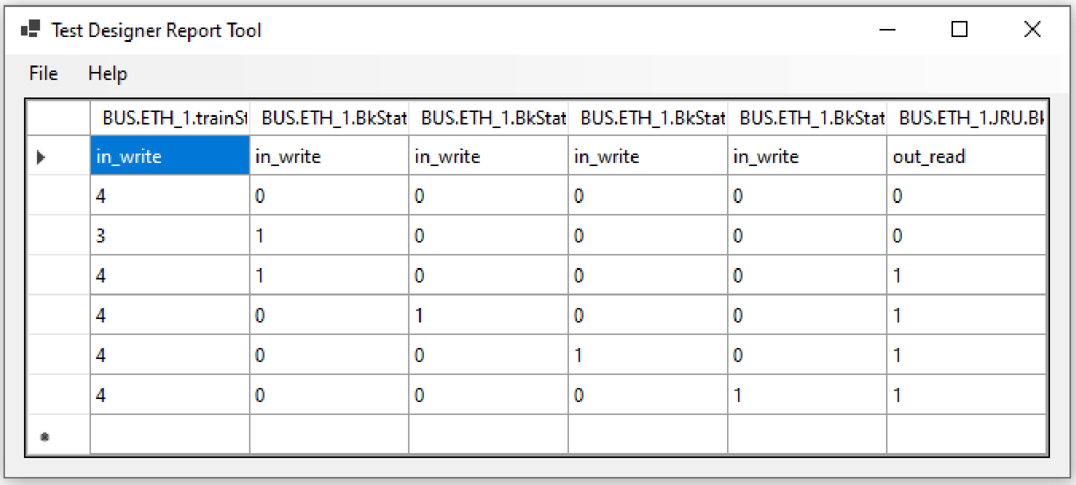
Once the Test Specification is generated by the Test Generator, it notifies the *Test Designer* and all the requirement test cases are displayed.

Requirement 3.10 from Section 3.4.6.2 will be used to present the *Test Designer* interface.

Requirement 3.10: JRU Dragging Brake Detected

```
REQUIREMENT(REQ348, 2F02_Traction_Braking, CCUS)
{
      GIVEN <Train speed> is greater than 3 kph
         and <Brake Applied> is equal to true
         for at least one brake in the Unit;
      WHEN;
      // Comment: JRU - Juridical Recording Unit
      THEN <JRU Dragging Brake Detected> is set to true;
}
```

The interface of the *Test Designer* is presented on Figure 3.16.



	BUS.ETH_1.trainS	BUS.ETH_1.BkStat	BUS.ETH_1.BkStat	BUS.ETH_1.BkStat	BUS.ETH_1.BkStat	BUS.ETH_1.JRU.B
▶	in_write	in_write	in_write	in_write	in_write	out_read
	4	0	0	0	0	0
	3	1	0	0	0	0
	4	1	0	0	0	1
	4	0	1	0	0	1
	4	0	0	1	0	1
	4	0	0	0	1	1
*						

Figure 3.16: Test Designer Interface

Similar to Table 3.7, boolean values True and False are converted to 1 and 0 respectively. The first row contains all the technical signals, and the second row the direction of the signal, i.e., if its a signal to be read or to be set on the system under testing.

- in_write - The value to be set on the corresponding technical signal.

- out_read - The expected results after setting the signals under in_write.

Third to last row are the list of test cases to execute on the system. Tests are executed from top to bottom. First, all the signals from the third row are exercised, then the forth one, and so on.

Sesnando is a functional tool under continuous development. It is not a certified tool, however, it is generating tests for certified software. Due to this circumstances the user is responsible to review and accept the outputs given by *Sesnando*, thus, he or she is able to modify the Test Specification as intended, and from there, generate a test script. The main usage of test scripts is presented on next Section 3.4.8.2.

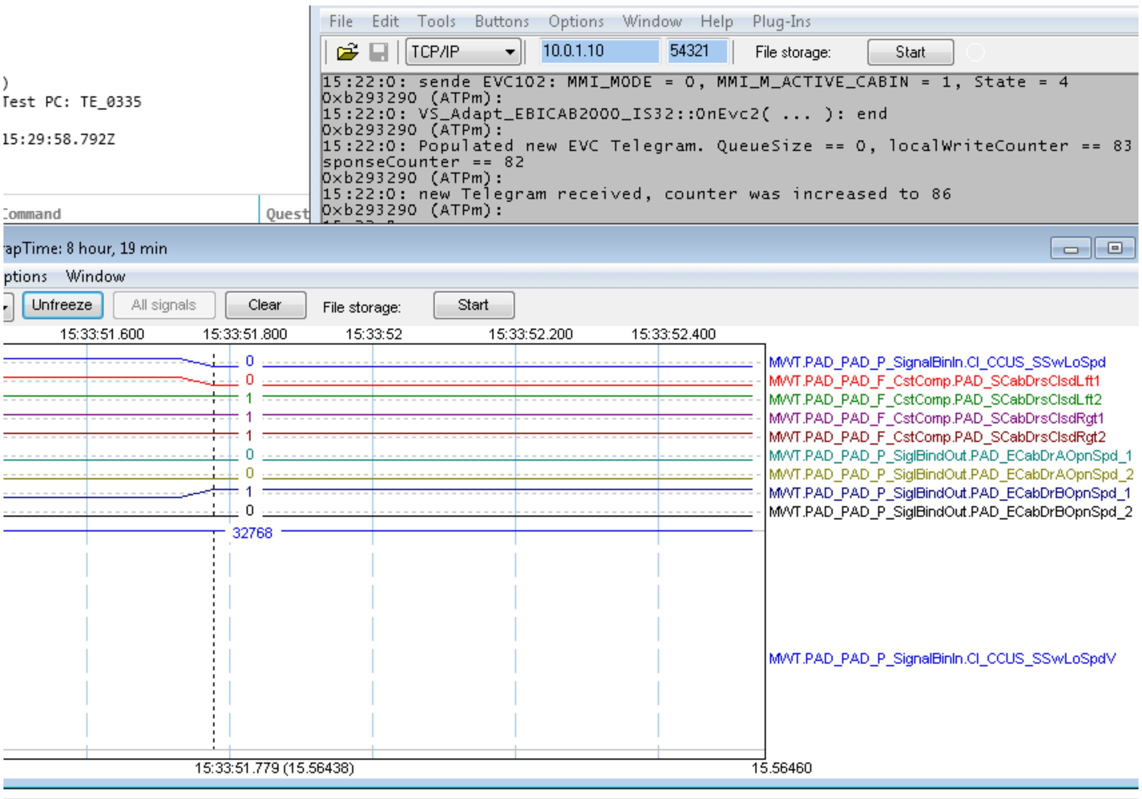
The test tables from the Test Designer is a representation of the test specifications presented on Section 3.4.6.

3.4.8.2 Test Scripts and Test Execution

After visualising and reviewing the test specification on the *Test Designer*, the tester is able to export it into a test script compatible with the Manufacturer test execution tool.

This feature can be located by navigating to "File -> Export to test script" in the interface of Figure 3.16. At this moment *Sesnando* only supports one script type (only to one specific test tool) but it is intended that the user interface could be extended to support other test tools as needed. For instance, in the future, one could create a set of transformation rules in a C# script that could be compiled from *Sesnando* using the CodeDOM Compiler.

Figure 3.17 presents the Manufacturer test tool that allows the setting and observation of the internal signals of a simulated train in operation.



```
)
Test PC: TE_0335

15:29:58.792Z
```

```
15:22:0: sende EVC102: MMI_MODE = 0, MMI_M_ACTIVE_CABIN = 1, State = 4
0xb293290 (ATPm):
15:22:0: VS_Adapt_EBICAB2000_IS32::OnEvc2( ... ): end
0xb293290 (ATPm):
15:22:0: Populated new EVC Telegram. QueueSize == 0, localWriteCounter == 83
sponseCounter == 82
0xb293290 (ATPm):
15:22:0: new Telegram received, counter was increased to 86
0xb293290 (ATPm):
```

```
0 MWT.PAD_PAD_P_SignalBinIn.CI_CCUS_SSwLoSpd
0 MWT.PAD_PAD_F_CstComp.PAD_SCabDrsClsdLft1
1 MWT.PAD_PAD_F_CstComp.PAD_SCabDrsClsdLft2
1 MWT.PAD_PAD_F_CstComp.PAD_SCabDrsClsdRgt1
1 MWT.PAD_PAD_F_CstComp.PAD_SCabDrsClsdRgt2
0 MWT.PAD_PAD_P_SiglBindOut.PAD_ECabDrAOpnSpd_1
0 MWT.PAD_PAD_P_SiglBindOut.PAD_ECabDrAOpnSpd_2
1 MWT.PAD_PAD_P_SiglBindOut.PAD_ECabDrBOpnSpd_1
0 MWT.PAD_PAD_P_SiglBindOut.PAD_ECabDrBOpnSpd_2
32768 MWT.PAD_PAD_P_SignalBinIn.CI_CCUS_SSwLoSpdV
```

Figure 3.17: Test Environment

On Figure 3.17 the background window presents a terminal that connects to a simulated train environment running on a validation facility.

The validation facility (which might be in a form of Test Rack or a local testers' computer (Section 2.5)) loads the script generated from *Sesnando*. On a simple operation, the signal values start to be set on the software to produce the desire outcomes. The real-time value of every technical signal through the test time-span can be observed on the multi-chart displayed on the focused window of the test tool on Figure 3.17. (Window names omitted due to copyrighted material)

This demonstrates that a complete end-to-end pipeline from requirements to test execution is possible with little user intervention.

Chapter 4

Experimental Analysis and Results

This chapter analyses *Sesnando* advantages and limitations comparing to the traditional approach regarding testing of Railway software projects, namely regarding the testers' effort, MCDC test coverage, and requirements complexity.

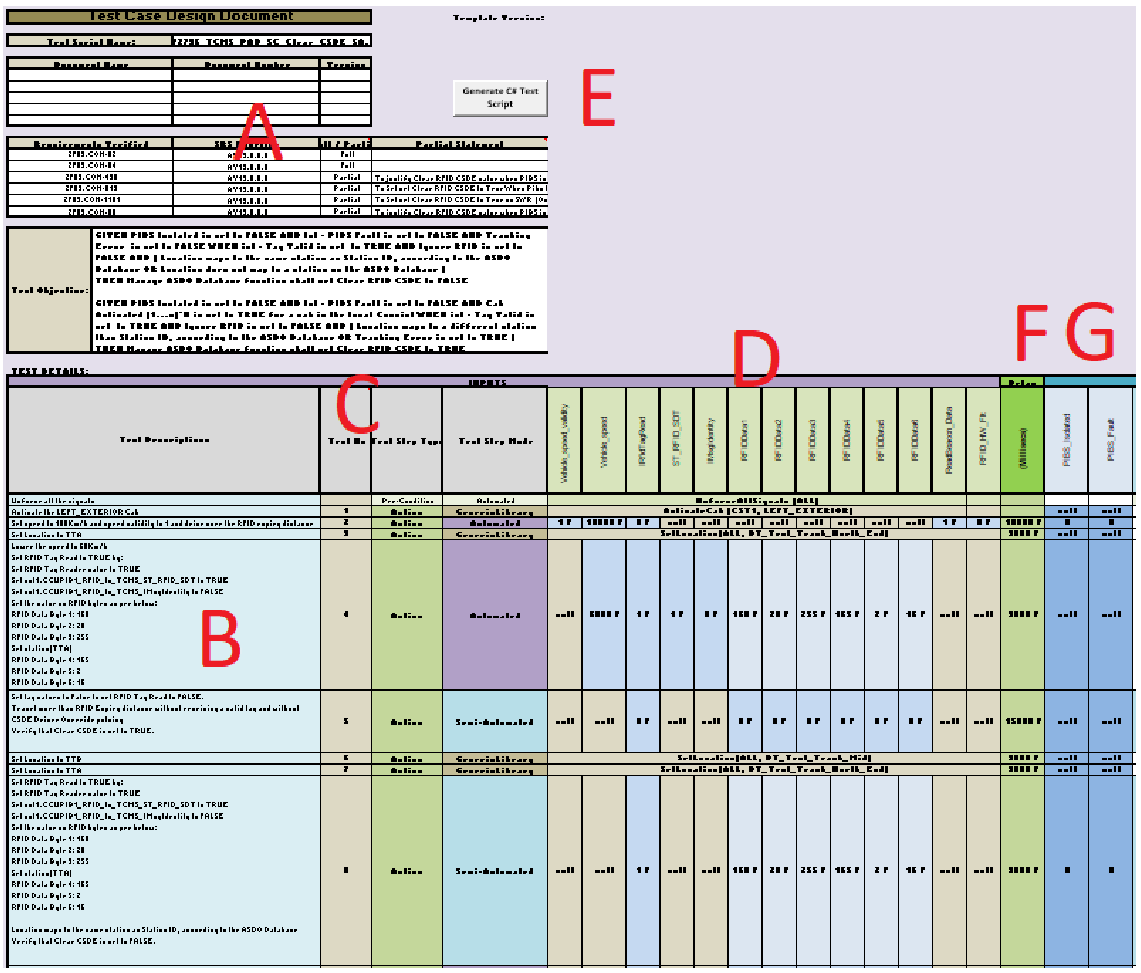
4.1 Effort analysis

This section presents the obtained results using the traditional methods versus the obtained results using the *Sesnando* application regarding the effort spent during testing activities. The results presented here were applied on a real railway project for a rolling stock manufacturer.

In order to write system test specifications for such project, several steps are required, which are illustrated on Figure 4.2. First, new requirements need to be obtained from the Requirement management tool such as IBM Dynamic Object Oriented Requirements System (DOORS). The software under test needs to be downloaded or the remote test racks need to be accessed depending on the nature of the requirement, SIL0 (non-SIL) to SIL2 respectively (See Section 2.5).

The design of a test specification requires the identification of the applicable software signals (technical signals) on the Interface Control Document (ICD), as defined on section 2.5, that satisfy the conditions of those requirements. As well as software signals, the requirement clauses need to be identified and combined according to the applicable requirement coverage criteria in the in the form of test cases.

Once all *Test Requirements* are identified, the tester designs a Test Specification containing all test cases in an Excel spreadsheet and must then generate a test script which is generated by a VBA program provided by the Manufacturer and is then run against the current software release. Figure 4.1 presents a manually written test specification using traditional procedures.



Test Case Design Document

Template Version:

Generate C# Test Script

Test Serial Name: [illegible]

Personnel Name	Personnel Number	Version

Requirements Verified	SRS [illegible]	Full / Partial	Partial Statement
[illegible]	[illegible]	Full	
[illegible]	[illegible]	Full	
[illegible]	[illegible]	Partial	[illegible]
[illegible]	[illegible]	Partial	[illegible]
[illegible]	[illegible]	Partial	[illegible]
[illegible]	[illegible]	Partial	[illegible]

Test Objective: [illegible]

TEST DETAILS:

Test Description	Test No	Test Step Type	Test Step Mode	Vehicle_speed_validity	Vehicle_speed	RFIdTagRead	ST_RFID_SDT	IMagIdentity	RFIDData1	RFIDData2	RFIDData3	RFIDData4	RFIDData5	RFIDData6	[illegible]	RFID_HW_Flt	(Milliseconds)	PIBS_Isolated	PIBS_Fault
[illegible]		[illegible]	[illegible]	[illegible]															
Activate the LEFT_EXTERIOR Cab	1	Action	[illegible]	[illegible]														[illegible]	[illegible]
[illegible]	2	Action	Automated	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	3	Action	[illegible]	[illegible]													[illegible]	[illegible]	[illegible]
[illegible]	4	Action	Automated	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	255 [illegible]	[illegible]	2 [illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	5	Action	Semi-Automated	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	6	Action	[illegible]	[illegible]													[illegible]	[illegible]	[illegible]
[illegible]	7	Action	[illegible]	[illegible]													[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	Action	Semi-Automated	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	255 [illegible]	[illegible]	2 [illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 4.1: System test specification - Traditional method

The items on Figure 4.1 are as follows:

- A - List of requirements under test;
- B - Description of test case;
- C - Test Case Number;
- D - Input Technical Signals;
- E - Button to run a VBA macro to generate a test script;
- F - Delay until checking the output;
- G - Expected Results.

The content within this test specification is not really important for the subject. The main idea is to demonstrate that *Sesnando* is able to generate similar test specifications as presented on Section 3.4.6.

A test report is automatically generated by the Manufacturers' test tool once the test finishes. Both the test specification and test report (which is an HTML file generated by the manufacturer test tool discriminating all pass/fails for every test step) are subject to a peer review. The goal of this activity is to detect possible human errors and validate the test results if no flaws are detected. An introduced human error on the specification will influence the test results, as such, the specification needs to be redesigned and re-executed.

Sesnando significantly reduces the effort involved in this process by automating the activities of Signal mapping (From Logical to Technical signals) and by automatically generating the test specification. This corresponds to steps 2. and 3. of Traditional methods in Figure 4.2, which result in the activities of using *Sesnando* in the right diagram of Figure 4.2. These are compared side by side, where text in gray represents the common testing activities and in bold the activities automated by *Sesnando*.

The requirements of the railway project must be loaded to *Sesnando* which then connects to a remote service (Signal Manager) to acquire the corresponding technical signals (this process is automatic). Once done, it will automatically generate a set of test steps using the railway project requirement coverage criteria and returns a test script compatible with the manufacturers' test tool. The activity of peer reviewing should not be discarded, this tool is not certified and human errors might be introduced on software signals repository.

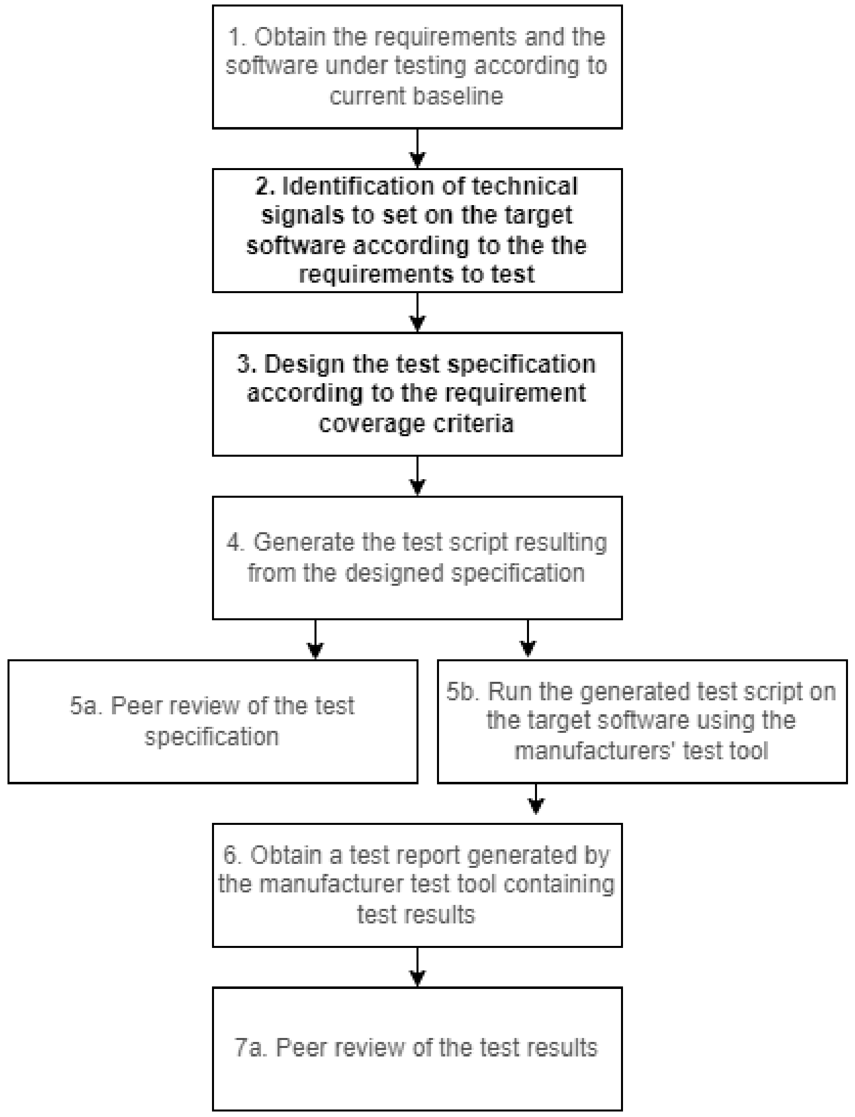
A research was carried out at CSW to accurately determine the effectiveness of *Sesnando*. Four people were inquired about the effort spent to design a test specification for a single requirement. Given that the most common activities between *Sesnando* and manual procedures are the technical signal identification and test generation, those will be considered for comparison, as these are the activities automated by using *Sesnando*.

Table 4.1 illustrates the effort that each participant (identified as P1 to P4) took to determine the software technical signals for a given requirement and the writing of the specification. They have all worked on the same requirement.

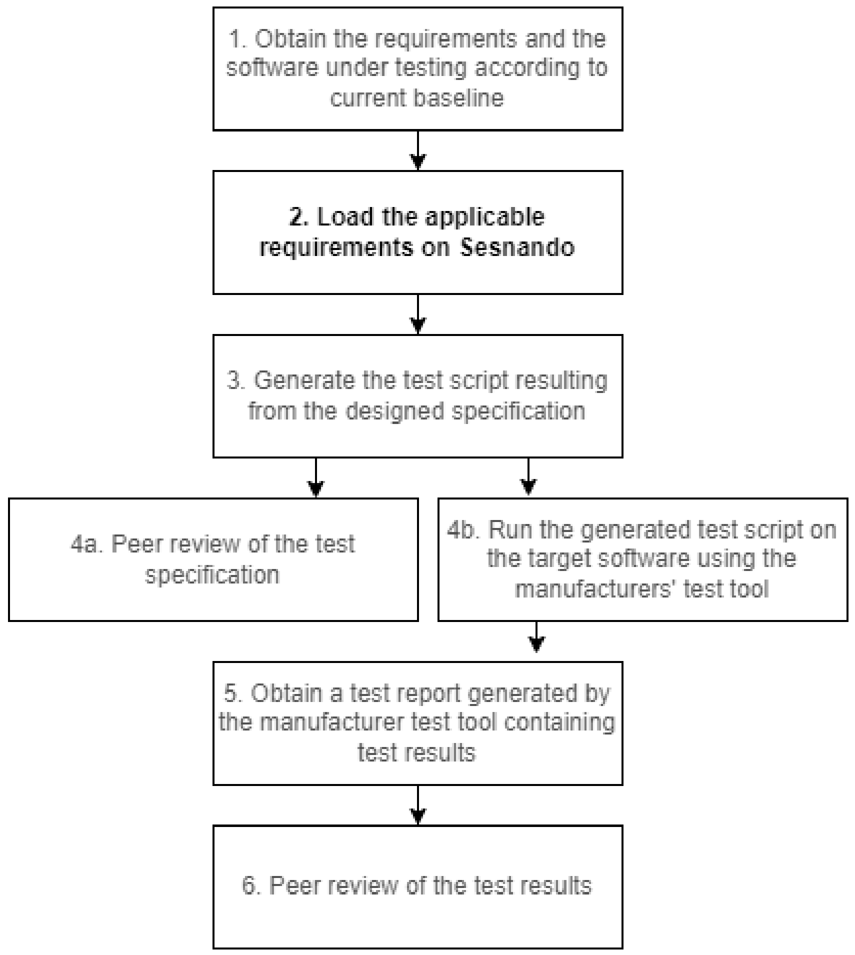
Taking the results obtained from the participants, **Technical Signal Analysis** represents an average effort of 26 minutes and the **Specification Writing** represents an average effort of 123 minutes for a single requirements containing a couple of AND conditions, resulting on an average of approximately 150 minutes per simple requirement. A participant also stressed out that most complex requirements can take up to two days to have a specification ready.

Sesnando is able to execute both activities mentioned on Table 4.1 in less that 1 minute, as through a simple application launch results in a generated specification.

When using *Sesnando*, the requirement signals need to be available on the *Signal Manager*. The ideal scenario would be to have all the necessary signals on the


1. Obtain the requirements and the software under testing according to current baseline
2. Identification of technical signals to set on the target software according to the the requirements to test
3. Design the test specification according to the requirement coverage criteria
4. Generate the test script resulting from the designed specification
5a. Peer review of the test specification
5b. Run the generated test script on the target software using the manufacturers' test tool
6. Obtain a test report generated by the manufacturer test tool containing test results
7a. Peer review of the test results

(a) Testing activities using Traditional methods


1. Obtain the requirements and the software under testing according to current baseline
2. Load the applicable requirements on Sesnando
3. Generate the test script resulting from the designed specification
4a. Peer review of the test specification
4b. Run the generated test script on the target software using the manufacturers' test tool
5. Obtain a test report generated by the manufacturer test tool containing test results
6. Peer review of the test results

(b) Testing activities using Sesnando

Figure 4.2: Testing using Traditional methods versus using Sesnando

Table 4.1: Effort using traditional methods

Participant	***Technical Signal Analysis***	***Specification Writing***
P1	15 Min.	75 Min.
P2	15 Min.	120 Min.
P3	45 Min.	195 Min.
P4	30 Min.	105 Min.
Avg. P	**26 Min.**	**123 Min.**

Signal Manager, so a specification is generated for any given requirement instantly. At the current stage of development, a functionality is being created, so Interface Control Documents (ICDs) and Input Signal Documents can be uploaded to populate the remote Signal Manager Repository 3.4.7, as every tester will benefit from this. However, adding signals manually is already possible.

It is important to enumerate the scenarios where Requirement signals are not available on the remote tool. The participants were asked to add a couple of signals to the Signal Manager. The effort required to generate a specification using *Sesnando* is presented on Table 4.2, for situations where the signals are present on the remote database and when they are not. The Signal Manager is now referenced as SM.

Table 4.2: Effort analysis of Signal management on Signal Manager

Scenario	***Technical Signal Analysis***	***Specification Writing***
Signals available in SM (Best Case Scenario)	$<$ 1 min.	$<$ 1 min.
Signals not available Experienced user in SM	10 Min.	$<$ 1 min.
Signals not available Unexperienced user in SM (Worst Case Scenario)	25-35 Min.	$<$ 1 min.

On the worst case scenario, a tester needs to identify technical signals on the available project resources, such as ICDs, learn how to use the Signal Manager and insert the new technical signals to be used by *Sesnando* in the future. However the generation of the specification is instant.

During this process it was considered the time that each participant took to learn the tool (*Participant 1-4 (Technical Signal Analysis) + Learning SM + Adding signals*).

This process is not applicable for future requirements that use the same signals, as they will already be available. According to the obtained results, *Sesnando* can save up to 90% of effort spent during the design of a test specification on the system testing activities.

4.2 Requirement complexity analysis

This section presents an analysis of the complexity of requirements and how *Sesnando* interprets them and what are the main difficulties on this process.

To analyse the complexity of existing requirements, a set of eighty (80) requirements has been exported and for each one, a complexity value of 1 to 4 was assigned. The result of this analysis is shown on Table 4.3.

Table 4.3: Requirement Complexity Analysis

Complexity	***Number of Req.***	***Req. Percentage***
1 - Interpretable	22	27%
2 - Requires Rewrite	13	17%
3 - Tool to be improved	29	37%
4 - Non-Compatible	15	19%

The level of complexityof each requirement is defined as follows:

- 1 - Interpretable - The existing requirement is written in a way that it is interpretable by *Sesnando* and its test specification can be generated.
  Example: Requirements on Section 3.4.
- 2 - Requires rewrite - The existing requirement contains words, characters or expressions that are not interpretable by *Sesnando*, however, its main logic is supported after the rewriting of such requirement.
  Example:

Requirement 4.1: Guard Panel

```
REQUIREMENT(REQ421-1, 2F03_Doors, TCMS)
{
   GIVEN <Guard Only mode> is equal to true
   AND <Train Speed> is less than or equal to 3kph
   // Should be: <active guard panel> for one cab;
   AND <active guard panel> in a single cab location;
   THEN <Doors Released> is set to True for all sides
   of the train;
}
```

- 3 - Tool to be improved - The existing requirement contains expressions that need additional checkings like timed signal pulses.
  Example:

Requirement 4.2: Doors failed to close

```
REQUIREMENT(REQ421-2, 2F03_Doors, TCMS)
{
   WHEN <TCMS command door close> is equal to true
   AND <Status door closed> is equal to false
   after 10 seconds;
   // Evt - Event
   THEN <Evt Doors failed to close> is set to True;
}
```

- 4 - Non-Compatible - Requirements that in order to be tested require information that is out-of-scope of *Sesnando*. Usually, those are defined as semi-automated tests and require a manual action from the tester. For instance, check that certain information is present on drivers' screen.
  Example:

Requirement 4.3: HMI Vehicle Overrun

```
REQUIREMENT(REQ421-3, 2F03_Doors, TCMS)
{
   // Human machine Interface
   WHEN <Vehicle Overrun> is selected on the HMI;
   // Evt - Event
   THEN <Evt Vehicle Overrun> is set to True;
}
```

This analysis concludes that, from the selected sample of requirements, 27% can be interpreted by *Sesnando*, 17% are not compliant with the predefined grammar and need rewriting, 37% require new developments on *Sesnando*, as they need specific observations e.g. a signal value is set within a given timeframe and 19% cannot be handled by *Sesnando* as they need the observation of the tester, e.g. an icon is blinking on the drivers' desk (on traditional methods, these are semi-automated tests).

Chapter 5

Conclusion

Sesnando has been presented at Critical Software and received very positive feedback by Requirement managers and testers. However, we believe that there is still a lot of room for improvement.

Such improvements are towards the interpretation and validation of all kinds of requirement conditions such as time-frames that aren't yet covered by *Sesnando*, as well as requirements from different markets that are written under the *GIVEN WHEN THEN* blueprint, not only on the Railway markets, but on ASDT (Aerospace Defense and Transportation) markets.

Moreover, it is ambitioned that *Sesnando* could be used on projects that do not solely rely on a signal based architecture systems as it is the case for the Railway project under study, but also, for instance, for Desktop and Web Applications.

Given this realisation, *Sesnando* could be a great tool that supports any tester in its daily activities. It is believed that there is a place out there for *Sesnando* in the world of behavior-driven-development, given that once a requirement is written, a test specification can be instantly generated. The advantage of generating tests in such an early phase can help to detect problems on requirements.

It is intended to turn *Sesnando* into a very robust solution, user-friendly and easy to adopt. Therefore, the following future work is proposed.

Future work

- Ability to include more than one requirement into a single specification, grouping requirements by functionality.

- Improvement of the user interface and the ability to export test scripts in multiple formats. Ability to compile user-defined scripts (written in C#) containing transformation rules from the Test Specification generated by *Sesnando* to the desired test scripts.

- Development of a user interface or a VSCode plugin that is capable to detect syntax errors and compilation warnings on requirements, so it becomes evident whether they are compliant with the grammar installed on *Sesnando*. It could have an internal dictionary to detect ambiguities on requirements.
- Checks for conflicting requirements, i.e. requirements that contradict each other, facilitating the process of requirements' review.
- Integration with Cucumber and defining the grammar of *Sesnando* as an extension to the Gerkin syntax, as Gerkin is very similiar to the *GIVEN WHEN THEN* blueprint presented on this document. *Sesnando* could, for instance, be used as a Cucumber engine for Signal based systems.
- Ability to pull and push the remote signal database, so *Sesnando* could be used offline.
- Integration with Jira. For instance, the ability to collect requirements directly from Jira so their test specifications could be generated using *Sesnando*.

Bibliography

[1] Luciana Provenzano. 3egm015__en TCMS GIVEN WHEN THEN instructions for developers and testers.docx.

[2] Lori Cameron. First Software Engineer | IEEE Computer Society. `https://www.computer.org/publications/tech-news/events/what-to-know-about-the-scientist-who-invented-the-term-software-engineering/`, 2008. [Online; Accessed 14-11-2022].

[3] Robert McMillan. Her Code Got Humans on the Moon—And Invented Software Itself. *Wired*. Section: tags.

[4] William E. Lewis and Gunasekaran Veerapillai. *Software testing and continuous quality improvement*. Auerbach Publications, 2nd ed edition.

[5] Ayba1/4ke Aurum and Claes Wohlin. *Engineering and Managing Software Requirements*. Springer Science & Business Media, July 2005. Google-Books-ID: pUG1IaikDhMC.

[6] Independent Verification and Validation (IV&V) Through the Eyes of DoD. `https://logapps.com/2013/07/independent-verification-and-validation/`, July 2013. [Online; Accessed 15-11-2022].

[7] Alexander Alexandrovich and Kirill Igorevich. INCOSE Guide for Writing Requirements. Translation experience, adaptation perspectives. page 15. [Online; Accessed 04-08-2022].

[8] The Definitive ANTLR 4 Reference. `https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference`. ISBN: 9781934356999.

[9] Mike Mannion and Barry Keepence. SMART requirements. *ACM SIGSOFT Software Engineering Notes*, 20, March 2004. [Online; Accessed 04-03-2022].

[10] Aaron Chou. Derived requirements syntax rules (vehicle functions). page 39.

[11] Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Alexandr Naumchev, Manuel Mazzara, and Bertrand Meyer. The role of formalism in system requirements (full version). type: article.

[12] Klaus Pohl and Chris Rupp. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam, foundation level, IREB compliant.* Rocky Nook, second edition edition.

[13] John Levine. *Flex & Bison: Text Processing Tools.* "O'Reilly Media, Inc.", August 2009. Google-Books-ID: nYUkAAAAQBAJ.

[14] Why you should not use (f)lex, yacc and bison. `https://tomassetti.me/why-you-should-not-use-flex-yacc-and-bison/`. [Online; Accessed 29-05-2022].

[15] Donald E. Knuth and Luis Trabb Pardo. Early development of programming languages. 7:419–493. [Online; Accessed 29-05-2022].

[16] DePaul University. BNF and EBNF. `https://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf`. [Online; Accessed 15-04-2022].

[17] Front-matter. In Robert Oshana and Mark Kraeling, editors, *Software Engineering for Embedded Systems*, pages i–iii. Newnes. [Online; Accessed 18-05-2022].

[18] Test coverage analysis of safety-critical systems. `https://ercim-news.ercim.eu/en75/special/test-coverage-analysis-of-safety-critical-systems`. [Online; Accessed 15-05-2022].

[19] Gregory Zoughbi, Lionel Briand, and Yvan Labiche. Modeling safety and airworthiness (RTCA DO-178b) information: conceptual model and UML profile. 10(3):337–367. [Online; Accessed 14-05-2022].

[20] John Joseph Chilenski and Boeing Commercial Airplane Company. Investigation of three forms of the modified condition decision coverage (MCDC) criterion. `https://rosap.ntl.bts.gov/view/dot/42764`. [Online; Accessed 23-05-2022].

[21] CENELEC 50128 and IEC 62279 standards | wiley. `https://www.wiley.com/en-us/CENELEC+50128+and+IEC+62279+Standards-p-9781848216341`. [Online; Accessed 30-04-2022].

[22] CENELEC - EN 50126-1 - Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 1:

Generic RAMS Process | Engineering360. https://standards.globalspec.com/std/10262901/EN2050126-1. [Online; Accessed 08-04-2022].

[23] CENELEC - EN 50128 - railway applications - communication, signalling and processing systems - software for railway control and protection systems | engineering360. https://standards.globalspec.com/std/14317747/EN2050128. [Online; Accessed 08-04-2022].

[24] European Standards. EN 61131-6.

[25] Grady Booch, Douglas L. Bryan, and Charles G. Petersen. *Software Engineering with Ada*. Addison-Wesley Professional. Google-Books-ID: iPZGJRW9bKgC.

[26] Les Hatton. Safer language subsets: an overview and a case history, MISRA c. 46(7):465–472. [Online; Accessed 30-04-2022].

[27] Rolf Schwitter. Controlled natural languages for knowledge representation. In *Coling 2010: Posters*, pages 1113–1121. Coling 2010 Organizing Committee. [Online; Accessed 14-05-2022].

[28] Deborah Frincke, Dave Wolber, Gene Fisher, and Gerald C. Cohen. Requirements specification language (RSL) and supporting tools. https://ntrs.nasa.gov/citations/19930003157. NTRS Author Affiliations: California Polytechnic State Univ., Boeing Co. NTRS Document ID: 19930003157 NTRS Research Center: Legacy CDMS (CDMS).

[29] Daniel Maciel, Ana Paiva, and Alberto Rodrigues da Silva. From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach:. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 265–272. SCITEPRESS - Science and Technology Publications. [Online; Accessed 23-05-2022].

[30] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. Automatic generation of acceptance test cases from use case specifications: an NLP-based approach. http://arxiv.org/abs/1907.08490. type: article.

[31] Javier J Gutiérrez, María J Escalona, Manuel Mejías, and Jesús Torres. Generation of test cases from functional requirements. a survey. page 10. [Online; Accessed 29-05-2022].

[32] Tom Collings. Controller-service-repository. https://tom-collings.medium.com/controller-service-repository-16e29a4684e5. [Online; Accessed 30-10-2022].

[33] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. 29(8):1157–1166. [Online; Accessed 05-11-2022].

[34] A Alajmi, E Mostafa Saad, and RR Darwish. Toward an arabic stop-words list generation. *International Journal of Computer Applications*, 46(8):8–13, 2012. [Online; Accessed 09-05-2022].

[35] Luís Jordão. CSW-SESNANDO-Requirement-Guidelines, 2022.

[36] Armin-Montigny. Evaluation of boolean expressions and MCDC test cases. `https://github.com/Armin-Montigny/MCDC`. original-date: 2019-04-27T09:08:04Z.

[37] Khenaidoo Nursimulu and Robert L. Probert. Cause-Effect Graphing Analysis and Validation of Requirements. In *In Proceedings of CASCON'95 , IBM Canada Ltd. and National Research Council*, pages 293–293, 1995.

Appendices

A Benchmarking

The Requirement 3.2 from Section 3.4.2 has been benchmarked using the Eclipse IDE with ANTLR4 IDE 0.3.6 from the marketplace, resulting in the Figure 3.6. Benchmark results are as follows:

Table 1: ANTLR Grammar Profiler

Measurement	***Value***
Input Size	198 chars, 6 lines
Number of Tokens	34
Parse Time (ms)	2268
Prediction Time (ms)	0,711 = 31.36%
Lookahead Burden	53/34 = 1.56
DFA Cache miss rate	43/53 = 81.13%

B Thesis proposal

IDENTIFICAÇÃO DA EMPRESA

DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA E DE SISTEMAS

PROPOSTA DE PROJECTO

Ano Lectivo de 2020/2021

Mestrado em Informática e Sistemas (Desenvolvimento de Software)

TEMA

Automatic test generation from software requirements

SUMÁRIO

Pretende-se desenvolver uma ferramenta capaz de gerar testes de software automaticamente. Esta, deverá ser capaz de interpretar um conjunto de requisitos escritos numa dada linguagem e gerar um conjunto de *Scripts* prontos a executar num ambiente de testes.

1. ÂMBITO

Os testes de software e de sistema são uma actividade intrínseca no desenvolvimento de software. Estima-se que um grande esforço no desenvolvimento de um sistema reside na parte de verificação e validação do mesmo. A presente dissertação tem como âmbito o desenvolvimento de uma ferramenta capaz de automatizar a criação desses mesmos testes através da interpretação de requisitos de software escritos no formato *Given When Then* permitindo assim reduzir largamente os custos associados a esta fase.

O trabalho aqui proposto visa uma colaboração entre o Instituto Superior de Engenharia de Coimbra e a Critical Software.

2. OBJECTIVOS

O presente projecto pretende atingir os seguintes objectivos genéricos :

- Desenvolver uma primeira versão da aplicação capaz de interpretar um dado conjunto de requisitos e gerar os respectivos *Scripts* de teste;
- Dissertação/Relatório final de projecto sintetizando o estado da arte, problemas encontrados e como estes foram superados, a conclusão e a sua contribuição científica.
- Prova pública final que será realizada no Instituto Superior de Engenharia de Coimbra.

•

3. PROGRAMA DE TRABALHOS

O projecto consistirá nas seguintes actividades e respectivas tarefas:

- ***T1*** – *Estado da arte* – Estudo das tecnologias e avanços científicos existentes e de que forma estes podem contribuir para o presente projecto.
- ***T2*** – *Análise e desmantelamento do problema* – Esta tarefa consiste na análise das fontes de dados e ferramentas disponíveis, i.e. conjuntos de requisitos e resultados de T1.
- ***T3*** – *Desenvolvimento da solução* – Desenvolvimento de um interpretador capaz de reconhecer o conjunto de dados fornecidos.
- ***T4*** – *Testes* - Actividades de verificação e validação da aplicação desenvolvida e possíveis ajustes.
- ***T5*** – *Relatório final de projecto* – Escrita de um relatório final de estágio contendo o resultado de todas as fases anteriores.

4. CALENDARIZAÇÃO DAS TAREFAS

As Tarefas acima descritas, incluindo os testes de validação de cada módulo, serão executadas de acordo com a seguinte calendarização:

O plano de escalonamento dos trabalhos é apresentado em seguida:

	Meses												
Tarefas		N		N+1		N+2		N+3		N+4		N+5	
T1		🟦	🟦										
T2				🟦	🟦								
T3					🟦	🟦	🟦	🟦	🟦	🟦			
T4										🟦	🟦	🟦	
T5		🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
Metas	INI		M1		M2					M3		M4	M5

INI		Início dos trabalhos
M1	(INI + 4 Semanas)	Tarefa T1 terminada
M2	(INI + 8 Semanas)	Tarefa T2 terminada
M3	(INI + 18 Semanas)	Tarefa T3 terminada
M4	(INI + 22 Semanas)	Tarefa T4 terminada
M5	(INI + 24 Semanas)	Tarefa T5 terminada

5. RESULTADOS

Os resultados do projecto serão consubstanciados num conjunto de documentos a elaborar pelo aluno de acordo com o seguinte plano:

M1	Elaboração do estado da arte
M2:	Apresentação e Discussão do problema
M3:	Tecnologias usadas e solução adoptada
M4:	Resultados e Conclusão da Dissertação
M5:	Relatório final completo

6. LOCAL DE TRABALHO

Não se aplica.

7. METODOLOGIA

Reuniões *Semanais/Bi-Semanais* entre os orientadores e o orientado.

Organização de um **Dossier de Projecto**.

8. ORIENTAÇÃO

ISEC:

João Cunha (jcunha@isec.pt)
Professor Director

Critical Software:
João Gabriel Silva(joao.gabriel@criticalsoftware.com)
Board Member

9. CARACTERIZAÇÃO E REMUNERAÇÃO

- Data de início: 01/10/2020
- Data de fim: A determinar