Test script design approaches supporting reusability, maintainability and review process

Master Thesis Work

Authors:
B.S.E. Aleksandar Aćimović
B.S.E. Aleksandar Bajčeta

Supervisor:
Adnan Čaušević, Ph.D.

Examiner:
Kristina Lundqvist, Ph.D.

Västerås,
2019.
Abstract

Software testing is widely considered to be one of the most important parts of software development life-cycle. In this research, we investigated potential improvements in the testing process and design of automated test scripts inside Bombardier Transportation. For the creation of automated test scripts BT is using a group of programs called TAF (Test Automation Framework). These scripts are used for testing Train Control Management System (TCMS), software that is used for managing the train. TAF can export its test scripts in XML format. XML scripts are analyzed in order to identify the most frequent changes. To better understand the life cycle of automated Test scripts official documentation that defines the Verification and Validation process inside BT was analyzed. Also, an interview was conducted with one of the responsible persons for testing. We believe that we have found a possible solution for improving testing process and creation of automated test scripts in BT, and to evaluate it proof of concept tool was developed. The main idea behind the tool is to write the test script using keywords which are based on analysis that was conducted on test specification documentation. These keywords represent frequent actions that are being tested on the train. By storing those actions in keywords re-usability of test script is being increased. Also, because they are based on naturally language, they are having positive effect on readability and maintenance of the test script.

Keywords: Automated test scripts, readability, re-usability, maintainability, review process, train systems, testing tools, natural language, robot framework, development life cycle
## Contents

List of figures ........................................ iv
List of tables ........................................ v
Abbreviations and Terminology ............................... v

1 Introduction .......................................... 1
   1.1 Background information and problem formulation .......... 1
   1.2 Related work ........................................... 2
   1.3 Method .................................................. 3

2 Data Analysis ......................................... 5
   2.1 Background information ................................... 5
   2.2 Initial Analysis .......................................... 7
   2.3 Steps and SubSteps analysis ............................... 9
   2.4 Initial changes analysis ................................... 9
   2.5 Changes analysis by category ............................. 11
   2.6 Changes analysis inside signals .......................... 12
   2.7 Data analysis conclusion .................................. 16

3 Development Life Cycle of Automated Test Scripts ... 17
   3.1 TCMS Verification and Validation .......................... 17
       3.1.1 Testing Levels ans Scopes ........................... 17
       3.1.2 Phases and Documentation ........................... 19
       3.1.3 Test specification ................................... 22
   3.2 Interview ................................................. 22
       3.2.1 Interview analysis ................................... 22
   3.3 Chapter conclusion ....................................... 24

4 Proof of Concept Tool ................................ 25
   4.1 Testing Environment ....................................... 25
   4.2 Connections inside Testing Environment .................... 26
   4.3 Concept and Ideas for Suggested Solution ............... 29
   4.4 Tool Implementation ...................................... 31
       4.4.1 DCUTerm Singleton .................................. 32
       4.4.2 Signals Library ....................................... 33
       4.4.3 Keywords ............................................. 34
       4.4.4 Test cases ............................................ 35
   4.5 Tool Discussion .......................................... 35
List of Figures

2.1 Test script editor .................................................. 5
2.2 Test script structure ........................................... 6
2.3 Payload frequencies by categories .............................. 9
2.4 Percentage of relevant changes ................................ 10
2.5 Percentage of relevant changes among signals ............. 11
2.6 DigitalSignal Changes .......................................... 15
2.7 AnalogSignal Changes ........................................... 15
2.8 CustomSignal changes ........................................... 16

3.1 Identified Test Levels ............................................ 18
3.2 Vehicle Breakdown to the System Level ...................... 19

4.1 General architecture of the Test Environment ............... 26
4.2 DCUTerm ............................................................ 28
4.3 Tracked signals .................................................... 28
4.4 Potential Final Architecture of The Proposed Solution .... 30
4.5 Architecture of The Proposed Solution ....................... 31
4.6 Architecture of implemented solution ......................... 32
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Payload type names and frequencies</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Payload frequencies by categories</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Number of lines changed by category and operation</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Number of lines changes by category (with and without irrelevant operation)</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Number of signal changes by category (with and without irrelevant operations)</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Number of control changes by category (with and without irrelevant operations)</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>TCMS Checkpoints Model</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Test Documentation Overview</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Test case example</td>
<td>22</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Today, the software is an essential part of our life, it is everywhere. Our businesses, everyday activities, even our lives depend on the quality performance of some software. Malfunctioning of some part of the software can give us a big headache, or even put our lives in danger. Because of that, it is important to make sure that software is reliable. Software testing is widely considered to be one of the most important parts of software development life-cycle. It represents actions that are needed to be performed in order to confirm that software is firm, reliable, and that is behaving according to its requirements. Software testing is never ending process, there is never enough time or resources for testing. Even when it is finished by software testers, you can not be sure that software is bug-free. After its release, users can be seen as testers. Unfortunately in the context of safety critical systems sometimes that can be catastrophic. The latest example of that is an accident that occurred on Boeing 737 [1]. Because of the costs and time that is needed for software testing, in the industry, a lot of companies are trying to implement automation testing in their development process. Automation testing comes with its advantages and disadvantages. Studies showed that biggest advantages of automation in software testing are re-usability of test cases and reduced amount of time that is needed for test execution, but drawbacks are costs that are needed for buying a tool, training the employees and designing test cases [2].

This thesis is divided into the five chapters. The Chapter 1 gives a brief description of a problem that we are addressing, and how we planned to conduct our research. Data Analysis that was needed in order to investigate most common changes in test script is described in Chapter 2. Further analysis of testing processes and life cycle of automated test scripts is discussed in Chapter 3. We proposed a new testing tool in Chapter 4. Our conclusions are drawn in the final Chapter 5.

1.1 Background information and problem formulation

Bombardier Transportation has developed internally a platform for testing train’s software referred to as a Virtual Train environment. This platform supports automated execution of test scripts, and for creating those scripts they are using tools that are developed in Bombardier Transportation. In order to improve their testing process, Bombardier is interested in writing test scripts as close as possible to the natural language. With this thesis we seek to address challenges that are preventing longer life-cycle and re-usability of test scripts. System testing is time consuming and expensive process. The aim of our research is
to identify the possible improvements to the testing process by evaluating approaches of designing and creating executable test scripts. Conventionally, before writing or modifying a test script several phases have to be run. Usually, the mentioned phases include the following: requirement specification, test specification and the test case itself. Every phase needs to be reviewed and validated in a processes which involves management of internal resources but also third parties such as quality control inspectors. The process is followed by a lot of bureaucracy and is really time and resource consuming. All the changes to the system usually lead to some, or all, of the phases to be run again leading to more time and resources being spent. This is posing a special problem for the environments in which the changes are rather frequent.

Re-usability of test cases is also an interesting problem that we want to investigate. There are several projects in Bombardier Transportation with a similar requirements where test cases can be re-used. With re-usable test cases, the process of software testing might become more affordable.

This thesis aims to provide answers to the following research questions:

- **RQ1** - What are the common events triggering changes to automated test script?
- **RQ2** - In what way does the testing life-cycle affect development of automated test scripts?
- **RQ3** - How can a natural language based tool, for test script creation, affect the testing processes from the maintainability, readability and re-usability aspects?

### 1.2 Related work

In order to better understand what problem we are facing, we investigated articles from two scientific databases IEEE Xplore and ACM Digital Library. The focus was on several subjects that are related to our topic. Those are testing of safety-critical systems, re-usability of test cases or test scripts, and their maintenance. The time span in which the papers were explored stretches from 1994 up to present.

In the context of safety-critical systems, our focus was on train system testing and on avionic system testing, topics extensively investigated in research ([3, 4, 5, 6]). They are focused on researching different tools, natural language semantics, and other test automation strategies that could improve the testing of safety-critical systems. In the context of our problem, these findings are relevant and useful for our research.

In the train system testing context, Zhang et al. [3] proposes a tool for generating test sequences for CTCS-3 Train Control System from MS Word formatted test cases, the tool, at the time of publishing the paper, was not fully able to generate test scripts without additional user inputs and, as authors described, was used as more powerful editing tool for test sequences. The difference in contribution when it comes to our research is that we created a proof of concept tool for writing test cases or test specifications. The second paper in the same context, by Venticinque et al. [4], investigated how the natural language semantics can be used to map and trace test scripts to a requirement.

The study was conducted on the ETCS (European Train Control System). Our proposed tool is a natural language based. This study shows the feasibility of a tool such as our proposed one and that the connection of the semantic terms used in the phases of the Test Script Design Life Cycle exists.

On the other hand, Niu et al. [5] explores the approach of using a central system signal
library where they in the end claim that such an approach reduces the need of modifying and recompiling of the software. In our case, we investigated how a similar approach can reduce the need of modifying of test scripts in the context of Bombardier Transportation and what are the most common reasons for modifying them in the same context.

When it comes to the context of testing avionic systems the paper by A. K. Jha [6] extensively researches the test automation in it and puts some focus on the Data-Driven Test Strategy in which scripts structures remain the same and only inputs and outputs are modified by the need. In their context, they prove that this approach is having a significant effect on reducing the effort of developing test scripts. This is one of the aspects that our tool is based on.

In the context of the re-usability of test cases or test scripts, this is extensively investigated in research ([7], [8], [9]). Research is focused on different approaches of identifying reusable test cases. The conclusions of these papers are of interest in our research. Dhareula et al. [7] are investigating identifications of the attributes of test cases for re-usability in the context of regression testing. They categorized the regression test selection techniques proposed in two groups: code-based techniques and requirement based techniques. When it comes to our research where we are investigating re-usability of test cases, and what is affecting those changes.

In the Black Box testing context, Vignjevic et al. [8] proposed a tool as a solution for the high re-usability of test suites. They claim that the proposed tool saves time for an adaptation of reused test suites between projects. Their tool is designed as a group of software tools that should complete the testing cycle of the project, from gathering the requirements, test execution, and generation of the report. Our tool is intended to merge, or at least to reduce the differences between writing requirements, test cases, and test scripts. Test Case Re-usability Metrics Models (TCRMM) is introduced by Juan et al. [9]. It is measuring re-usability of test cases and with that, it is possible to identify the most effective reuse strategies. According to [9] there are four re-usability factors: understandability, changeability, independence, and universal. These factors will help in developing the re-usability assessment model. When it comes to our research, identified re-usability factors were useful during identification of reusable test cases.

1.3 Method

In order to successfully perform this research, we conducted a case study in Bombardier Transportation, the following steps that are suggested by Wohlin et al. [10]. The goal of the study was to help us understand the testing process in Bombardier Transportation, give us more information about test script life-cycle, find opportunities for re-usable test cases, as well as investigate the linguistic semantic used in the test specification and test case documents. Data for this case study has been provided to us, and previous versions of Bombardier Transportation’s test scripts are saved in their Version Control System, system requirement, test specification, and test case document. We have analyzed this data, and more details about this are written in Section 2 and introduce ourselves with a detailed grasp of the process used in Bombardier Transportation. Further analysis gave us answers on how changes in the test script have affected the testing process and why were they changed in the first place. Also, we investigated parts of the test script that haven’t been changed. They gave us insight about re-usability opportunities in test scripts. And finally, the specific semantic used in the test specification and test case documents is going to be noted as it is important for the natural language tool to be based precisely on it.
Introduction

Further steps include creating a proof of concept natural language based tool with the previous findings in mind. The tool demos were presented to Bombardier Transportation’s engineers involved in the testing life cycle. This way we want to extract the data if the semantic structure of the proposed tool is similar enough to formal documents produced in different phases of the test script development life cycle so that it can replace them to the extent.
Chapter 2

Data Analysis

2.1 Background information

The test scripts that were analyzed for this research were written during the development for the C30 project. The C30 project represent one of the trains developed by Bombardier Transportation. These test scripts were created using the TAF (Test automation framework) tool. The framework consists out of two tools. The first tool is used for creating and editing scripts, Test script editor (TSE). The TSE offers test engineer various benefits when it comes to creating automated test scripts that simulate different signals inside the train. It is user-friendly application with a proper GUI that allows the user simple drag and drop scripting. The tool and one of those scripts are presented in Picture 2.1. The second tool, Test Execution Manager (TEM), is used for handling and executing test written in TSE. The TSE can export scripts in XML format, which we used for our
script analysis. Structure of these scripts is straightforward. After tags, that are used for initialization of the scripts, its script’s version and its name, tool’s version, the script is constructed from one or more Step tags. Later on, each step contains the different number of SubStep tags, that can be categorized into actions, reactions, and snippets. Operations that are performed on System Under Test (SUT) are represented with actions and reactions are used to verify the SUT response. The Snippets are used to merge one or more sub-steps inside the sub-step. It is useful for creating and defining instructions that are frequently used. The tags that are of interest for our research are Payload tag. They can be of a different type, and they are used for defining the information that is going to be sent so the script could be executed successfully. The structure of the test script is presented in figure below.

![Figure 2.2: Test script structure](image)

Figure 2.2: Test script structure
2.2 Initial Analysis

As already mentioned before the needed data for the thesis kick-off was provided by Bombardier Transportation in the form of Visual SourceSafe (VSS)\textsuperscript{11} virtual library containing XML formatted testing scripts. An initial analysis of the test script structure was performed to gain insight into what script parts are relevant. The conclusion made by this analysis was that the blocks with the tag name "Payload" carry the most relevant information when it comes to our research. Example of one of these "Payloads" is presented bellow.

**Listing 2.1: Example of payload**

```xml
1  <Payload xsi:type="AnalogSignal">
2    <Description>A2_EKEY2:Badge is present and valid in desk station (BOOL)</Description>
3    <FullPath>Signals.Electrical.Key.IP.A2_EKEY2:Badge is present and valid in desk station (BOOL)</FullPath>
4    <VCSPath>35.IP_PD_Inputs.1.0</VCSPath>
5    <BusType>IP</BusType>
6    <MWTSignalName />  
7    <ReactionTimeOutInMS>6000</ReactionTimeOutInMS>
8    <SetSignal>true</SetSignal>
9    <LockSignal>true</LockSignal>
10   <MeasureDurationInMS>0</MeasureDurationInMS>
11   <MWTVersion>0</MWTVersion>
12   <VCSVersion>1</VCSVersion>
13   <VCSID>20303</VCSID>
14   <RequireFlankOnReaction>false</RequireFlankOnReaction>
15   <ToleranceInMS>0</ToleranceInMS>
16   <WaitInMS>0</WaitInMS>
17   <BusID>3</BusID>
18   <BusName>cst1</BusName>
19   <BusAddress>37000101:10.0.7.41</BusAddress>
20   <BusAddressOffset>6</BusAddressOffset>
21   <DataType>BOOLEAN8</DataType>
22   <DCUIPSignalName>A2_EKEY2_Badge_is_present_and_valid_in_desk_station_(BOOL)</DCUIPSignalName>
23   <SignalAddress>6</SignalAddress>
24   <SignalType>Reaction</SignalType>
25   <UnitTypeName>IP</UnitTypeName>
26   <SignalDirection>In</SignalDirection>
27   <UnitID>161</UnitID>
28   <IsHex>false</IsHex>
29   <AddressingScheme>ByteInTelegram</AddressingScheme>
30   <MinValue>−2147483648</MinValue>
31   <MaxValue>2147483647</MaxValue>
32   <CurrentValue>1</CurrentValue>
33   <BetweenValue>0</BetweenValue>
34   <CurrentReactionOperator>Equal</CurrentReactionOperator>
35   <SignalDataType>Int</SignalDataType>
36 </Payload>
```

As it can be seen above these "Payloads" can contain various information with various formats and thus they can be of different types. Having this in mind we conducted a second analysis aiming to gain information on what these types are and their frequencies. The analysis was run on 56 different script files, each of them being the final version.
As it can be read in Table 2.1, the results of the analysis identified 18 different Payload types. These Payload types can be further grouped in 3 different categories, in Signals, Controls, and Other. In Table 2.1 the Signals are marked by the green color and the Controls by the blue. Important information is also that the Comment Payloads are making more than 17% of scripts, as they aren’t relevant for scripts functioning the statistics were performed with them excluded. Having in mind previous conclusions the results were summed up by the defined categories as well.

When the data is summed up by categories the most interesting information is that the Signal Payload types make more than 51% (42% with the comments included) of all script Payloads. Data summed up by categories is presented in the Table 2.2, and graphical representation of the table is presented in Figure 2.3. This is interesting as it supports the idea of Signals being one of the main reasons for modifying the test scripts.

**Table 2.1:** Payload type names and frequencies

<table>
<thead>
<tr>
<th>Payload Type</th>
<th>Count</th>
<th>%</th>
<th>% w/o Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snippet</td>
<td>163</td>
<td>3,159</td>
<td>3,816</td>
</tr>
<tr>
<td>Comment</td>
<td>889</td>
<td>17,229</td>
<td>0,000</td>
</tr>
<tr>
<td>AnalogSignal</td>
<td>122</td>
<td>2,364</td>
<td>2,856</td>
</tr>
<tr>
<td>CustomSignal</td>
<td>928</td>
<td>17,984</td>
<td>21,728</td>
</tr>
<tr>
<td>DigitalSignal</td>
<td>1138</td>
<td>22,054</td>
<td>26,645</td>
</tr>
<tr>
<td>SelectionControl</td>
<td>551</td>
<td>10,678</td>
<td>12,901</td>
</tr>
<tr>
<td>WaitControl</td>
<td>594</td>
<td>11,512</td>
<td>13,908</td>
</tr>
<tr>
<td>ParallelReactionStart</td>
<td>65</td>
<td>1,260</td>
<td>1,522</td>
</tr>
<tr>
<td>ParallelSubReactionInit</td>
<td>65</td>
<td>1,260</td>
<td>1,522</td>
</tr>
<tr>
<td>ParallelSubReactionStart</td>
<td>65</td>
<td>1,260</td>
<td>1,522</td>
</tr>
<tr>
<td>ParallelSubReactionEnd</td>
<td>65</td>
<td>1,260</td>
<td>1,522</td>
</tr>
<tr>
<td>ParallelReactionEnd</td>
<td>65</td>
<td>1,260</td>
<td>1,522</td>
</tr>
<tr>
<td>AnalogControl</td>
<td>181</td>
<td>3,508</td>
<td>4,238</td>
</tr>
<tr>
<td>SemiControl</td>
<td>18</td>
<td>0,349</td>
<td>0,421</td>
</tr>
<tr>
<td>ReInitiateSimulations</td>
<td>2</td>
<td>0,039</td>
<td>0,047</td>
</tr>
<tr>
<td>SelectionIndicator</td>
<td>188</td>
<td>3,643</td>
<td>4,402</td>
</tr>
<tr>
<td>AnalogIndicator</td>
<td>38</td>
<td>0,736</td>
<td>0,890</td>
</tr>
<tr>
<td>Unit</td>
<td>23</td>
<td>0,446</td>
<td>0,539</td>
</tr>
<tr>
<td>Total</td>
<td>5160</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 2.2:** Payload frequencies by categories

<table>
<thead>
<tr>
<th>Payload Group</th>
<th>%</th>
<th>% w/o Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signals</td>
<td>42,403</td>
<td>51,229</td>
</tr>
<tr>
<td>Control</td>
<td>36,027</td>
<td>43,526</td>
</tr>
<tr>
<td>Other</td>
<td>21,570</td>
<td>5,245</td>
</tr>
</tbody>
</table>

When the data is summed up by categories the most interesting information is that the Signal Payload types make more than 51% (42% with the comments included) of all script Payloads. Data summed up by categories is presented in the Table 2.2, and graphical representation of the table is presented in Figure 2.3. This is interesting as it supports the idea of Signals being one of the main reasons for modifying the test scripts.
2.3 Steps and SubSteps analysis

XML scripts that were provided to us consist of the series of Steps and SubSteps. In those tags, further action is called and tested. We wanted to investigate if there is a trend in changes in the number of Steps or SubSteps in test script versions, or those changes are random. This was done using a script that counts Steps and Substeps of each test script version and calculates their average value for a test script. After analysis of gained results, the conclusion was made that changes in the frequency of both Steps and SubSteps are random.

2.4 Initial changes analysis

The next step is to extract and analyse the precise changes made in scripts between each of its versions sequentially. For this, all the scripts and their versions were extracted and sorted resulting in a pool of approx. 250 MB of data. The plan was to find the most appropriate format of the data for the difference analysis, without losing the relevant information in the process and using such formatted data to extract relevant changes and information about them. Having this in mind it was first decided to change the names of Payload types to their corresponding category name. The "Other" category wasn’t taken into consideration here as it only makes approx. 5% of all Payloads with comments excluded. The script was created for modifying the names and it was run on the sorted data pool creating a new pool with the renamed tags. At this point, data was prepared for comparing. Another Python script was created for extracting the differences between test script versions. The Python script used a xmldiff[12], an open source library.
for the purposes of extracting the differences from XML files. The test script versions were compared sequentially and it was decided that initial versions should not be taken in consideration as they carried only the data of the test script initialization, thus the comparison between the initial version and next version would carry a considerable amount of changes and it would distort the results in further analysis. The result of running this Python script was files with all the changes in the form of a list of actions containing information that should be done to transform the first XML file into the second one. Further, an additional Python script was created to analyse the amount and type of changes throughout all test scripts, the result is shown in tables 2.3 and 2.4.

<table>
<thead>
<tr>
<th>Signal</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>197455</td>
<td>57533</td>
<td>58086</td>
<td>121220</td>
<td>13928</td>
<td>1927</td>
</tr>
<tr>
<td>All</td>
<td>73597</td>
<td>12621</td>
<td>10932</td>
<td>38890</td>
<td>3362</td>
<td>1387</td>
</tr>
</tbody>
</table>

Table 2.3: Number of lines changed by category and operation

Figure 2.4: Percentage of relevant changes

Table 2.3 shows the total number of lines changed by two main categories of Signals and Controls, as well as by operation performed to change the line from old to new version. The operations "move", "insert", and rename are considered not as significant as the rest of operations as it fundamentally doesn’t change the information in the file and it is usually a result of other operations performed on the file. Graphical representation of Table 2.3 is presented on Figure 2.4.
Data Analysis

<table>
<thead>
<tr>
<th>Category</th>
<th>SUM Relevant</th>
<th>SUM ALL</th>
<th>% Relevant changes</th>
<th>% ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal</td>
<td>131851</td>
<td>450149</td>
<td>81.507</td>
<td>76.175</td>
</tr>
<tr>
<td>Control</td>
<td>29915</td>
<td>140789</td>
<td>18.492</td>
<td>23.825</td>
</tr>
<tr>
<td>All</td>
<td>161766</td>
<td>590938</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2.4: Number of lines changes by category (with and without irrelevant operation)

Figure 2.5: Percentage of relevant changes among signals

When these numbers are summed up and the percentage is calculated by categories like shown in Table 2.4, it can be noticed that from all the changed line ones that are affecting the Signals contribute to approx. 81% (76% with irrelevant operations included) of all changes between these two categories. With this, we can already draw a conclusion that Signals are in fact parts of automated test scripts that are changed the most, and thus confirming our initial prediction in the Section 2.2.

2.5 Changes analysis by category

After initial data analysis that was explained in 2.2, further analysis by category was conducted. The goal of this analysis was to investigate which category in signals and controls was changed the most. This was done by updating the script that was used for the initial differences analysis script. Based on the result from this analysis it was concluded that among Signals, DigitalSignal was changed the most (65%), changes in CustomSignal were the second (30%), AnalogSignal was changed least times (0.03%), and among Controls, it was SelectionControl (60%). These results are presented on tables 2.5 and 2.6. Figure 2.5 shows graphical representation of the percentage of changes among
Data Analysis

signals. Based on the number of percentages of each category in Signals, the results of changes are as expected. But for the Controls number of changes in one category is of interest, that is AnalogControl. Despite WaitControl is the second most frequent category in Controls, the number of its changes is 0.04848, and the number of the third most frequent one, AnalogControl is 0.31780.

<table>
<thead>
<tr>
<th>AnalogSignal</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>7355</td>
<td>2798</td>
<td>3011</td>
<td>3928</td>
<td>688</td>
<td>86</td>
<td>3.86</td>
<td>4.993</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DigitalSignal</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>141900</td>
<td>34521</td>
<td>37186</td>
<td>81899</td>
<td>7862</td>
<td>1144</td>
<td>65.68</td>
<td>61.215</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CustomSignal</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>63506</td>
<td>20214</td>
<td>17889</td>
<td>33153</td>
<td>5755</td>
<td>697</td>
<td>30.46</td>
<td>33.792</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Number of signal changes by category (with and without irrelevant operations)

<table>
<thead>
<tr>
<th>ParallelSubReactionEnd</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>141</td>
<td>27</td>
<td>52</td>
<td>21</td>
<td>84</td>
<td>0.00004</td>
<td>0.913</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReInitiateSimulations</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.00002</td>
<td>0.010</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ParallelReactionStart</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>26174</td>
<td>2169</td>
<td>2266</td>
<td>14396</td>
<td>523</td>
<td>91</td>
<td>0.00248</td>
<td>16.898</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AnalogControl</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>143</td>
<td>71</td>
<td>74</td>
<td>90</td>
<td>87</td>
<td>0.00332</td>
<td>0.073</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ParallelSubReactionStart</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>974</td>
<td>172</td>
<td>55</td>
<td>442</td>
<td>33</td>
<td>59</td>
<td>0.01209</td>
<td>1.066</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ParallelSubReactionInit</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>177</td>
<td>62</td>
<td>4</td>
<td>32</td>
<td>59</td>
<td>0.00308</td>
<td>1.103</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SelectionControl</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>44502</td>
<td>9440</td>
<td>7617</td>
<td>22706</td>
<td>2222</td>
<td>438</td>
<td>0.00090</td>
<td>66.595</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SemiControl</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>177</td>
<td>59</td>
<td>15</td>
<td>46</td>
<td>63</td>
<td>0.00279</td>
<td>1.153</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WaitControl</th>
<th>move</th>
<th>insert</th>
<th>update-text</th>
<th>rename</th>
<th>delete</th>
<th>insert-attribute</th>
<th>% ALL</th>
<th>% Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>3046</td>
<td>1332</td>
<td>835</td>
<td>754</td>
<td>294</td>
<td>502</td>
<td>0.04848</td>
<td>10.572</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: Number of control changes by category (with and without irrelevant operations)

2.6 Changes analysis inside signals

After initial changes analysis, and changes analysis by category next step was to analyse changes inside the signals. The goal of this analysis was to find out which tags inside Signal was affected with most changes. The first step of this analysis was to investigate the structure of all three types of signals. The conclusions of this investigation were that all three types of signals are constructed with different types of tags. Most of the tags from DigitalSignal and AnalogSignal are the same, with some additional tags that are used in AnalogSignal. CustomSignal has its own child that DigitalSignal and AnalogSignal does not have, it is called CCUUnit and it has its additional tags. Structure of each of these signals is presented bellow.

Listing 2.2: Digital signal

```xml
  <Payload xsi:type="DigitalSignal">
    <Description>AI:Activate train indication</Description>
    <FullPath>Signals.A1:Activate train indication</FullPath>
    <VCSPath>60.MIODX Outputs.27.0</VCSPath>
    <BusType>MIODX</BusType>
    <MWTSignalName />
    <ReactionTimeOutInMS>6000</ReactionTimeOutInMS>
    <SetSignal>true</SetSignal>
    <LockSignal>true</LockSignal>
    <MeasureDurationInMS>0</MeasureDurationInMS>
    <MWTVersion>0</MWTVersion>
    <VCSVersion>1</VCSVersion>
    <VCSID>2389</VCSID>
  </Payload>
```
Data Analysis

Listing 2.3: Analog Signal

```
<Payload xsi:type="AnalogSignal">
  <Description>A1:A1: Wheel rotational speed, axle 1</Description>
  <VCSPath>7.MVB_Analog_Inputs.0.22</VCSPath>
  <BusType>MVB</BusType>
  <MWTSignalName />
  <ReactionTimeOutInMS>10000</ReactionTimeOutInMS>
  <SetSignal>true</SetSignal>
  <LockSignal>true</LockSignal>
  <MeasureDurationInMS>0</MeasureDurationInMS>
  <MWTVersion>0</MWTVersion>
  <VCSVersion>2</VCSVersion>
  <VCSID>2626</VCSID>
  <RequireFlankOnReaction>true</RequireFlankOnReaction>
  <ToleranceInMS>0</ToleranceInMS>
  <WaitInMS>0</WaitInMS>
  <BusID>0</BusID>
  <BusName>MvB</BusName>
  <BusAddress>710</BusAddress>
  <BusAddressOffset>128</BusAddressOffset>
  <DataType>U16</DataType>
  <DCUIPSignalName>A1_A1__Wheel_rotational_speed__axle_1</DCUIPSignalName>
  <SignalAddress>128</SignalAddress>
  <SignalType>Reaction</SignalType>
  <UnitTypeName>MVB</UnitTypeName>
  <SignalDirection>In</SignalDirection>
  <UnitID>86</UnitID>
  <IsHex>true</IsHex>
  <AddressingScheme>BitInWordInTelegram</AddressingScheme>
  <MinValue>-32768</MinValue>
  <MaxValue>32767</MaxValue>
  <CurrentValue>0</CurrentValue>
  <BetweenValue>0</BetweenValue>
  <CurrentReactionOperator>Equal</CurrentReactionOperator>
  <SignalDataType>Int</SignalDataType>
</Payload>
```
Listing 2.4: Custom signal

```xml
<Payload xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CustomSignal">
  <CCUUnitCollection>
    <Name>CCU-O A1</Name>
    <CCUUnits>
      <CCUUnit>
        <Name>CCU-O A2</Name>
        <IPAddress>10.0.1.41</IPAddress>
        <Port>54321</Port>
        <IsValidSignalMWTName>MWT:TC_B_TC_B_Supervision_T3.
          TC_BV_S_Leader</IsValidSignalMWTName>
        <UseValidSignal>true</UseValidSignal>
        <LogTaskLevel>T2</LogTaskLevel>
        <Version>1</Version>
      </CCUUnit>
      <CCUUnit>
        <Name>CCU-O A1</Name>
        <IPAddress>10.0.1.11</IPAddress>
        <Port>54321</Port>
        <IsValidSignalMWTName>MWT:TC_B_TC_B_Supervision_T3.
          TC_BV_S_Leader</IsValidSignalMWTName>
        <UseValidSignal>true</UseValidSignal>
        <LogTaskLevel>T2</LogTaskLevel>
        <Version>1</Version>
      </CCUUnit>
    </CCUUnits>
  </CCUUnitCollection>
  <MWTSignalName>MWT.TC_BI_iCCUS_AToCCUO.TC_BI_CCUS_X_LifeSignA</MWTSignalName>
  <ReactionTimeOutInMS>180000</ReactionTimeOutInMS>
  <CurrentValue>1</CurrentValue>
  <BetweenValue>0</BetweenValue>
  <CurrentReactionOperator>GreaterEqualThan</CurrentReactionOperator>
  <MeasureDurationInMS>0</MeasureDurationInMS>
  <MarkSignal>false</MarkSignal>
  <RequireFlankOnReaction>false</RequireFlankOnReaction>
  <ToleranceInMS>0</ToleranceInMS>
  <WaitInMS>0</WaitInMS>
</Payload>
```

In both DigitalSignal and AnalogSignal tag that is called BusAddress was changed the most. Changes in all other tags were fairly close to the average, while BusAddress was changed 2.4 times more than average. Changes for DigitalSignal and AnalogSignal are presented on Figure 2.6 and Figure 2.7. On these figures "Actual" represents percentage of changes inside signals. In CustomSignal most changes affected tags that are related to CCUUnit. On average CCUUnit and related tags changed 1.45 times more than the rest of them. Changes for CustomSignal are presented on Figure 2.8.
Data Analysis

Figure 2.6: DigitalSignal Changes

Figure 2.7: AnalogSignal Changes
2.7 Data analysis conclusion

Before the analysis began on automated test scripts of C30 project, an assumption was made that the Signals would be the parts of the scripts changed the most. This assumption was based on the fact that the whole train system functions through signal exchanges, and thus they are the most susceptible to changes triggering the chain change reaction to automated test scripts. This was confirmed on few levels. The first indication for this assumption to be true was the fact that Signal Payloads made around 51% of all of the Payloads in analyzed test scripts. The first confirmation came from the fact that out of two major Payload groups, Signal and Control, Signals had more than 81% of relevant line changes in all of the analyzed test scripts. Further analysis showed that in various Signal Payloads most of the changes made were on the specific data correlating to implementation of the train. In DigitalSignal and AnalogSignal Payloads information of a BusAddress changed 2.4 times more than average compared to the rest of information Tags, while in CustomSignal CCUnit and it’s information was changed 1.45 times more than average. These findings exposed an opportunity for potential improvement of automated test scripts maintenance. At this point, the idea for improvement was seen in implementing a better system for handling Signal changes.
Chapter 3

Development Life Cycle of Automated Test Scripts

To understand better the whole Development Life Cycle Automated Test Scripts it is mandatory to at least understand the basic procedures and strategies used for Verification and Validation in context of Bombardier Transportation. For this purposes we have taken few steps, the first one was to read and analyze the official documentation describing these processes and strategies. The second step was to interview the engineers directly involved in the process with the aim to gain more insight with the assumption that formal documents and things documented within them usually deviate in a way from the real world cases.

3.1 TCMS Verification and Validation

As the whole research and analysis revolved around TCMS scripts and System testing logical step was to further stay in the same scope, following that logic the document of interest was the official procedure document for TCMS Verification and Validation. This document was created for the purposes of describing overall Verification and Validation strategies in order to comply to the EN 50657 [13] and EN 50128 [14] standards. Although this document was created for the purposes of TCMS it is defining overall V&V strategies and should be used in other projects with specific refinements.

3.1.1 Testing Levels ans Scopes

Within their V&V process Bombardier Transportation uses two different verification techniques:

- Review
- Testing

In this case testing, as in process of executing test cases on software and/or hardware, is the point of interest. There are in general six levels of the TCMS architecture and the requirements and because of that there are six test levels identified in the figure 3.1.
From this it can be concluded that the System is highly complex and consists of multiple parts and all of the parts have to be checked and made sure they function properly. Regardless of this how many levels there is the top priority is to reach the System Testing as soon as possible because these tests validate main System functionalities and are only the first, and the lowest, level functionalities of the whole system in which trains are gonna be functioning. To have a better understanding just how important is to reach System level Testing figure [3.2] represents the breakdown of the System in which TCMS has to function flawlessly.
The whole product consists of various parts and has to function in yet another higher system environment and for all of this to function properly extensive testing has to be done on all of these levels. This is why the System level Testing poses as a one of the top priorities, it has to be reached as soon as possible as it influences the processes on the higher abstraction levels on a great scale.

### 3.1.2 Phases and Documentation

From the previously mentioned facts it can already be concluded that not only the whole testing process has many layers but that due it’s complexity strict rules need to be followed in order to organize and track the whole V&V process. According to the TCMS Verification and Validation document in regards of Testing a batch of documents has to be created in various checkpoints. Most of these documents have a life cycles of their own and they usually consist out of three:

- Preparation and creation of the document
- Document review
- Document update and release
In this case Document review is considered as an internal review and it is mandatory to be conducted on every document as part of documentation is generated by specialized tools or are simple reports. For every document there is are defined times when they should be started and delivered. These times are defined as checkpoints in an internal model called TCMS Checkpoint Model. This model, checkpoints codes, and checkpoints titles are presented on Table 3.1. These checkpoints represent the phases of the TCMS Development Life Cycle and according to TCMS Checkpoint Model Instructions Document they are agreed for governing every new project and generalizingly market with Tx. The documents overview can be seen in the Table 3.2.

For the purposes of System Testing, the point of interest for this research, all of the TCMS test are based on TCMS System requirements and the test cases are documented in TCMS System Test Specification document. These test are usually ran using different tools and environments, mostly simulated ones. Because of the simulated environments always deviate in a way from the real world ones the differences are always calculated, documented and taken in consideration. Only regression test have to be ran on the real world environment. All of the test results are documented in the System Test Report document with all the relevant details. In the end, the formal review and test reports are submitted for TCMS Verification, Validation ans Assessment.

<table>
<thead>
<tr>
<th>Checkpoint code</th>
<th>Checkpoint title</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>TCMS Project Start</td>
</tr>
<tr>
<td>T1</td>
<td>TCMS Subsystem Requirements Freeze</td>
</tr>
<tr>
<td>T2</td>
<td>TCMS Subsystem Architecture Freeze</td>
</tr>
<tr>
<td>T3</td>
<td>TCMS Application SW Requirements Freeze</td>
</tr>
<tr>
<td>T4</td>
<td>TCMS Ready for Integration and Testing</td>
</tr>
<tr>
<td>T5</td>
<td>TCMS Ready for Vehicle Integration</td>
</tr>
<tr>
<td>T6</td>
<td>TCMS Ready for Final Validation</td>
</tr>
<tr>
<td>T7</td>
<td>TCMS Ready for Assessment</td>
</tr>
<tr>
<td>T8</td>
<td>TCMS Ready for Vehicle Homologation</td>
</tr>
<tr>
<td>T9</td>
<td>TCMS Project Closure</td>
</tr>
</tbody>
</table>

Table 3.1: TCMS Checkpoints Model
<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
<th>Internal review required</th>
<th>Starting checkpoint</th>
<th>Ending checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Plan</td>
<td>The document contains general requirements and is a part of Test Execution Plan.</td>
<td>Yes</td>
<td>Not specified</td>
<td>T1</td>
</tr>
<tr>
<td>Test Execution Plan</td>
<td>The document contains specific requirements for related test levels.</td>
<td>Yes</td>
<td>T1 - SW Component</td>
<td>T4 - SW Component</td>
</tr>
<tr>
<td>Classification Tree</td>
<td>The document is a part of Test Specification and is used to indentify all of the test cases that are meeting the requirements.</td>
<td>Yes</td>
<td>T1 - SW Component</td>
<td>T4 - SW Component</td>
</tr>
<tr>
<td>Test Specification</td>
<td>The document describes the test design.</td>
<td>Yes</td>
<td>T1 - SW Component</td>
<td>T4 - SW Component</td>
</tr>
<tr>
<td>Test Script</td>
<td>Test Scripts contain instructions for automatical execution of test cases or test sequences.</td>
<td>Yes</td>
<td>T1 - SW Component</td>
<td>T4 - SW Component</td>
</tr>
<tr>
<td>Test Traceability Matrix</td>
<td>An automatically generated document linking the other relevant documents and artifacts to each other.</td>
<td>Yes</td>
<td>T1 - SW Component</td>
<td>T4 - SW Component</td>
</tr>
<tr>
<td>Test Record</td>
<td>The document contains the results of the test cases, it may be created either manually or automatically.</td>
<td>No</td>
<td>Not specified</td>
<td>T4 - SW Component</td>
</tr>
<tr>
<td>Test Iteration Report</td>
<td>The document contains the results of the test cases defined in the Test Execution Plan for the assigned test specification. It is an intermediate report.</td>
<td>No</td>
<td>Not specified</td>
<td>T5 - System level</td>
</tr>
<tr>
<td>Test Report</td>
<td>The document contains the same things as a Test Iteration Report but it is a final report and it will be used for validation.</td>
<td>No</td>
<td>Not specified</td>
<td>T5 - System level</td>
</tr>
<tr>
<td>Test Metrics</td>
<td>Various documented data about previously done testing collected for the purposes of better understanding, comparing and improving various approaches.</td>
<td>No</td>
<td>Not specified</td>
<td>T5 - System level</td>
</tr>
</tbody>
</table>

**Table 3.2: Test Documentation Overview**
### Table 3.3: Test case example

<table>
<thead>
<tr>
<th>Step</th>
<th>Test case</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Initial State:</strong> Train is in Energy save mode</td>
<td><strong>Check that:</strong> - Main menu is shown on IDU - Battery contactors in A1 and A2 are Closed under “Kraft-försörjning” - ACM state is “In Operation” under “Kraft-försörjning” - an event is set indicating that cab A1 has been activated</td>
</tr>
<tr>
<td>2</td>
<td><strong>Wait until Main Reservoir Pressure is 1001kPa</strong></td>
<td>On IDU check that: - Main compressor status for B1 and B2 under Tryckluft is shown “OFF”</td>
</tr>
<tr>
<td>3</td>
<td><strong>Press “Körklar” on drivers desk</strong></td>
<td><strong>Check that:</strong> - Holding brakes are applied - All the MCMs are in state “Charged”</td>
</tr>
<tr>
<td>4</td>
<td><strong>Deactivate the cab A1</strong> - <strong>Activate the cab A2 with valid authorized user</strong> - <strong>Repeat the steps 1-3 from Cab A2</strong></td>
<td></td>
</tr>
</tbody>
</table>

### 3.1.3 Test specification

Test specification represents a document that contains list of test cases that should be executed. Each test case contains set of steps that are needed to be done in order to execute test case properly. Also, for each step there is a reaction that should occur. This document is used by test engineers for performing manual testing. One test case from the document is presented on the Table 3.3.

### 3.2 Interview

In order to gain better insight about testing processes in Bombardier Transportation, an interview [3.1] was conducted. The interview took place inside Bombardier Transportation and respondent was Markus Einarson. He has been working in Bombardier Transportation in the area of testing the TCMS software and was a logical choice for this interview, because of his involvement in companies testing process and he is the responsible person for previously analyzed TAF test scripts. The goals for this interview were to find out:

- What are the factors that are affecting testing process?
- What are advantages and disadvantages of TAF test tool used in C30 project?
- How are changes inside the project affecting testing process, and what is causing them?
- Are there any potential opportunities for better re-usability of automated test scripts and/or their parts?
- Are there any semantic standards that could be used in keyword based writing of the test scripts?

#### 3.2.1 Interview analysis

At the beginning of the interview, the permission was gain by the respondent for using his full name and surname in the report. After his introduction, he briefly explained the testing process inside the company that was used for the C30 project. He explained what different types of regression testing are used in C30 project. The first type is general regression testing with the purpose of checking if new changes introduced to the system
affected a normal drive cycle of the train, in other words if the normal behaviour is affected by them. This kind of tests do not follow any complicated procedure for their creation, meaning that they aren’t based on any type of documentation and the only artifacts are the automated test script instead. On the other hand, two other types of regression testing are documented and their documentation is kept in Rational DOORS\[15\]. These two types are regular and safety test cases and they are required to be run manually first, based on created test cases, then converted to automated test scripts. These test cases are created based on the software requirements, also documented and stored in Rational DOORS\[15\].

After that questions regarding test tool were asked. The tool that was used is TAF tool, and it consists of three parts, TAF script editor, TAF execution manager, and TAF test engine. One of the big advantages by Markus is that TAF tool is closely integrated with manual testing environment VCS (Vehicle Control Simulator). Also, TAF is user-friendly, the tester does not need to write the code, instead it is possible to select certain components, such as signals, buttons to push, etc... The drawback is that there is not a good export format for the TAF tool, which makes readability of test scripts much harder. When it comes to re-usability feature, with TAF tool it is possible to perform certain action multiple times by creating snippets. These snippets represent reusable parts that can trigger a chain update of TAF scripts if updated themselves.

Next set of questions were about changes inside the project, and how are they affecting the testing process. According to our respondent most of the changes are triggered by SCRs (Software Change Request), direct request for the change. These changes can be two sides as they can trigger changes in both directions, from software under test to automated test scripts and vice versa. Also, in case some changes are done in test cases that are documented in Rational DOORS\[15\], it is quite simple and straightforward to change the test script. When it comes to automation test script, which is executed every night, in case some failure happens during the execution test script is updated. The time that is needed for changing the test script varies from minutes to several hours, even days. Most of the time is spent for investigation of what actually needs to be updated. Our respondent confirmed conclusions that were made in previous data analysis, most of the changes were done on signals and on small parameters such as timings and changing signal names. When it comes to signal changes the TAF environment has a built in support function when a signal VCS library is changed. This function enables the user to check if the script has any of the signals in it changed but it requires the person to open every script, check for the possible change with the provided function and then manually update the signal. At the end of this question section we expressed and explained the idea for a data driven design of test where signals would be given aliases and those aliases would further be used in automated test scripts instead of direct signal names from the VCS library. In respondent opinion this could potentially increase re-usability of the scripts between different projects while having in mind no trains are the same and there would still be a need for adapting the scripts for the specific project.

When it comes to the standardized semantics for writing test specifications there isn’t any at the company level. But it is possible to identify some similarities in the documentation, so there is some informal standard on the project level. Also, the respondent gave us positive feedback about our idea for the keyword-based testing tool, although he also stated the concern that it the success of such tool depends on willingness of the staff to use and adapt to it.
3.3 Chapter conclusion

Through analyzing the official documentation for the procedures and strategies in relation to the TCMS Verification and Validation, and conducting an interview with Bombardier Transportation, few conclusions could be made. The first important fact was that the two techniques used in the V&V process, review and testing, tightly depend one on another. This is due to fact that most of the phases leading to the final testing result require extensive documentation. This documentation has to be internally reviewed and approved first before moving to a next phase. This further poses as a challenge as the process is time consuming on it’s own but also it has to be done for multiple levels. The thigh relation between these two techniques and the consequential time consuming problem they create was yet again confirmed through conducted interview, the respondent stated that, although it is preferable for the whole testing process to start as soon as possible, it is waited for the whole project to reach certain maturity before the test start being implemented. The reason for this, as stated by the respondent, was the pure amount of changes the whole project goes through the early stages which would further create a chain reaction of changes through all of the deliverable down to the automated test scripts.

The second major conclusion from the research conducted for the purposes of this chapter was that the problems created from the facts in the first conclusion created a deviation from the official procedures and strategies for TCMS V&V. This deviation is seen through concrete practices of testing where separate, undocumented, tests are made. These tests represent general regression tests based purely on experience of test engineers. They are system level tests, which further implied the importance of the whole process to reach system level testing as soon as possible as the whole product has to be tested further on higher functionality levels.

Further, the important information gained from the interview confirmed our results from the Chapter 2. From the experience of the respondent, the changes made the most to automated test scripts are mainly from the fact that implementation specific data, such as signals, change often. The whole process, in the context of TAF scripts, is semi-automated and, although straightforward, can consume a lot of time. It was also stated that the TAF tool lacks in the ability to export the created scripts in a readable format. The best way to read the scripts is through the Script Editor itself and, although it is simple to install it, it is not convenient to do in case a third party has to gain access to the scripts.

Unfortunately, we were unable to conduct more interviews regarding TAF scripts as the respondent was the only one responsible for creating them within C30 project. The information from the interview could be highly personal. Due to this fact the results and conclusions extracted from the interview should be treated with considerable caution.
Chapter 4

Proof of Concept Tool

4.1 Testing Environment

The process of creating and executing test script in Bombardier Transportation is quite complex, and in order to successfully finish it, several tools are used. As it was mentioned in Chapter 3, a tool that was used for creating the test scripts, the Test Automation Framework (TAF). The TAF was developed inside Bombardier Transportation, using C# programming language, with the intention to speed-up and automatize testing process. It consists of three different tools, TAF test script editor (TSE), TAF execution manager (TEM), and TAF test engine (TE). The TSE provides a GUI editor that is used for writing test scripts, TE is an application that serves to execute test scripts created with the TSE against TCMS or SoftTCMS, and the TEM serves to create and execute sessions. A session contains a list where tests scripts are defined to be executed. Bombardier Transportation is dealing with safety-critical systems, and because of that when it comes to changes of signal’s values timing is crucial. There is a huge amount of mathematical analysis behind the TE for dealing with responses from the SUT. With this it is possible to analyse signal’s values in different situations during the execution of the test script. The TAF has access to VSS Database. This database stores information about Sessions, Test Scripts and Snippets.

The TEM receives a script from the TSE, and it converts it to simple String which contains a set of commands that need to be executed. These commands are executed in Vehicle Control Simulator tool (VCS). This tool is consist of two parts, VCS Database and VCS Application. In the VCS Database information about the VCS application is stored. The VCS Application simulates train surroundings and parts of the TCMS. The TCMS is also tested with the VCS by using manual testing techniques. The VCS has its own development cycle, and it is being updated during the whole project. The TAF tool and the VCS are closely integrated and they are sharing the same data. In order to monitor the status and values of signals that are being sent, TAF is using tool that is called Data Communication User Terminal (DCUTerm). With the DCUTerm it is possible to send commands in order to set or retrieve a signal, track its value, etc.. In order to communicate with some other system DCUTerm can use TCP/IP connection. The SoftTCMS represents a simulated environment of the Train Control Management System (TCMS). Both of them represent a test object which is built on standard Ethernet technology that allows easy integration with other systems. The architecture of the Testing Environment is presented on Figure 4.1.
4.2 Connections inside Testing Environment

Communication between the TAF and the VCS can be established in two ways. One way is a simple Internet Protocol connection. The TAF is connected to the VCS using its IP address and port number. It sends a String which contains a set of commands that shall be executed inside the VCS. These commands consist of commands type, the path to the object, and value that is going to be set. Command types can be to set a signal value, get a value of a signal, lock a signal, etc... Each signal that is being manipulated with performs specified functions. Their identifiers and description of their functions are kept in the VCS database. In the respond, VCS sends a String where each command that is supposed to be run by the VCS is set to either true if that command passes or false if it fails.

The second approach is to use the DCUTerm between the TAF and the VCS in order to log the signals. It communicates with other applications using the TCP/IP connection. The DCUTerm communicates with other applications by sending different types of command,
these commands differ between target systems. Some of them are opening a variable (signal), forcing a new value, or starting to log a signal. All of them can be executed by typing them directly to the DCUTerm. Also, the DCUTerm provides and API that exposes functions to control it from the third party software, this way it can be used to as an middle man in the communication between the third party software and the SUT. For example, in order to establish a connection with the SoftTCMS, DCUTerm is triggered by command "ConnectWith". For this command to be executed successfully DCUTerm needs to receive type of the connection (TCP/IP), IP address of the machine that runs the SoftTCMS (it can also be Localhost), and port number that is used by SoftTCMS. In order execute some of the commands on SUT, the DCUTerm can be triggered with "ExecuteCommand" function. This function also needs a String that contains command that should be executed. In our case for changing the value of a signal that could be "O SingalName/SignalValue". For confirming the changes on the signals that are made sign dot (".") is used. A simple example of invoking the DCUTerm and connecting it to the VCS can be seen in the Listing 4.1.

Listing 4.1: Initial Analysis Python Script

```python
1 import ctypes
2 from win32com.client import Dispatch
3 import time
4
dcuTerm = Dispatch("DCUTerm.App")
5 time.sleep(10)
6 dcuTerm.ConnectWith("TCP/IP", "Localhost", "6001")
7 print("Connecting...")
8 time.sleep(10)
9
dcuTerm.ExecuteCommand("o PLACEHOLDER.Enable_Device/1/.")
10 time.sleep(10)
```

An example of the DCUTerm and use of its commands is shown in Figure 4.2. The tracking of the signal’s values is presented in Figure 4.3.
Figure 4.2: DCUTerm

Figure 4.3: Tracked signals
4.3 Concept and Ideas for Suggested Solution

After analysis of test scripts, test processes inside Bombardier Transportation, and their testing environment next step was to develop a proof of concept tool that would suggest a new approach in creating test scripts for Bombardier Transportation. With our solution we aimed to improve couple of aspects:

- The re-usability of created automated test scripts within, as well as between, projects
- The readability of automated test scripts
- The maintenance time

For the first aspect the idea for the solution was firstly based on the fact that the products being developed by Bombardier Transportation belong to same family of products, which are trains. From this fact we know that products, although never completely the same, will in the end have a fairly big number of functionalities the same. All of these functionalities are in the end tested in one way or another and the tests are designed and documented in the Test Specification documents. These documents describe the tests on the high level, describing what should be tested and in what sequence leaving out the technical details. From the interview conducted for the purposes of Chapter 3 we know that Test Specification documents usually have the same or similar semantic between the projects, and within the projects the nomenclature is almost always informally standardized. This trait supported our idea that a natural language based tool, with an ability to create automated test scripts as similar as possible to tests described in Test Specification, would be a good idea for improving the re-usability aspect between and within projects. The ability of this tool to create test scripts on the very high level of an abstraction would definitely improve the readability aspect of the automated test scripts, especially for the third parties with limited or non-existent specific knowledge of the project. This especially can be useful for the third party reviewers and inspectors, thus reducing the time needed for reviews done by and further reducing the waiting time for the product to go to the next phase towards the production. At this point a logical question is raised, where would the technical details go within the test scripts? The idea behind this part is that specific words used in test specification, as well as in proposed tool, would act as functions. These functions, for example "Login", "Start", "Activate" etc., would have the technical specifics and behaviour implemented in the background. To make these implementations more reusable, the specific data such as Signals would be given aliases that would be linked to specific Signal names in project specific libraries. This way not only the specific implementations would need less modifications for new projects but the maintenance of the whole scripts in a single project would improve as the Signals data would be centralized. Having in mind the conclusions from Chapter 2 an approach where a single change in a Signal library would be enough would in fact significantly improve the maintenance time. One of the ideas how the general architecture of the tool could look like was inspired by M. Taylor from his presentation in RoboCon 2018. In his presentation he described his experiences with building efficient multi-platform testing tools. His idea was modified and is represented as a potential architecture for our proposed tool in Figure 4.4.
For the purposes of this research our goal was to create a concept tool that would, for now, be able to replicate some of the test cases from the C30 project. Because of that the idea architecture for these purposes look as shown in the Figure 4.5.
4.4 Tool Implementation

Our proof of concept tool is developed using Python programming language, with the help of Robot Framework. Robot Framework is open source automation framework that has tabular test data syntax and it supports the keyword-driven testing approach [17]. There are several reasons why are we using Robot Framework for our tool. One of those reasons is structure of test cases that are written in Robot Framework. With its natural language based keywords and tabular structure, test cases looks like simplified sentences. We believe that this structure will improve readability and maintenance of test scripts. With Robot Framework it is also possible to create create and use self-made libraries. Test script would be run against the previously described SoftTCMS. The DCUTerm would be use as a tool for connecting and sending commands to the SoftTCMS. We also used functionality of the DCUTerm to log and track the signals and changes of their values during the execution of the test script. The main part and controller for our implementation is a singleton class DCUTERMSingleton. This class contains methods that are sending command to the DCUTerm. Our singleton class is used by Keywords library of our tool. This component contains set of methods that represent custom made keywords that are used for manipulating with signals. It is using Signals Library that can be splitted in two parts, SignalsName and SignalsValues. Architecture of this implementation is presented on Figure 4.6.
4.4.1 DCUTerm Singleton

As main part and controller of our tool the DCUTermSingleton class is communicating with the DCUTerm tool. For communication with the DCUTerm tool it is using DCUTerm’s API that has methods for connecting, sending commands, terminating app, etc. The class is implemented as a singelton class. The reason for this is a logical choice as invoking the class should be the same as invoking, or starting, a session. To put it more simply, it is important for the state of the connection to the DCUTerm to be the same in all of the parts of Test Case, or multiple Test Cases depending on the desired test configuration.

Method for sending connection request with the SoftTCMS takes as input parameters type of connection, IP address of the device, and port that is used by SoftTCMS. Implementation of that method is shown bellow.

**Listing 4.2: Sending connection request**

```python
@staticmethod
def connect(connectionType, IPAddress, PortNumber):
    # Connection details to SoftVCS
    DCUSingelton.connectionType = connectionType
    DCUSingelton.IPAddress = IPAddress
    DCUSingelton.PortNumber = PortNumber

    # Connect
    DCUSingelton._.dcuInstance.ConnectWith(
        DCUSingelton.connectionType,
        DCUSingelton.IPAddress,
        DCUSingelton.PortNumber
    )

    # Give us some time to connect
time.sleep(10)
```
For setting signal values in our tool a method setSignal is implemented. As input parameters it takes name of the signal and value that is going to be set. For sending command to the DCUTerm tool it is using its ExecuteCommand method.

**Listing 4.3: Setting signal value**

```python
@staticmethod
def setSignal(signalName, signalValue, waitTime = 2):
    DCUSingelton.__dcuInstance.ExecuteCommand("o " + signalName + "/" + signalValue + "/.")
    time.sleep(waitTime)
```

For getting value of the previously changed signal a method getResponse is implemented. As input parameters it takes name of the signal that was changed. The method from the DCUTerm’s API that we are using is GetTraceWinLastSampleAsString. This method will return value of the signal as a String.

**Listing 4.4: Getting signal value as a response**

```python
@staticmethod
def getResponse(signalName):
    return DCUSingleton.__dcuInstance.GetTraceWinLastSampleAsString(signalName)
```

With our tool it is also possible to set signals into a watch list. This watch list is giving user a possibility to track signal values during execution of a test script. To enable this functionality two methods are implemented. One is adding signals to the watch list, and the second one is sending commands to the DCUTerm.

**Listing 4.5: Tracking signal values**

```python
@staticmethod
def addToWatchList(signalName):
    DCUSingleton.watchList.append(signalName)

@staticmethod
def watchSignals(watchList, waitTime = 2):
    DCUSingleton.__dcuInstance.ExecuteCommand("FLS 0")
    time.sleep(waitTime)
    for signal in watchList:
        DCUSingleton.__dcuInstance.ExecuteCommand("FLC " + signal)
        time.sleep(waitTime)
```

### 4.4.2 Signals Library

Signals Library is a component whose main function is to store names and values of signals that are being used. It can be splitted into two part a SignalNames Library and a SignalsValue Library. The SignalNames Library stores names of the signal from the signals database into variables whose names are much easier to read and understand by parties that are not directly involved in development of the TCMS. Naming convention of these variables is topic that should be discussed in future. This way, as mentioned before, test scripts stop depending on direct, implementation specific, signal names but instead use the chosen signal name aliases.

**Listing 4.6: Signal names**

```python
```
Proof of Concept Tool

#Enable development
ENABLE_DEVELOPER = "PLACEHOLDER_Enable_Dev"

#Enable developer commands
DEV_COMMAND = "PLACEHOLDER_CI_V_DevCom"

#Enable forced signals
FORCE_SIGNALS = "PLACEHOLDER_TC_CI_IDU_FrcEn"

#Driver id set signal
DRIVER_USERID_SET = "PLACEHOLDER_TC_CI_IDU_X_UsrId_S"

#Driver authentication code set
DRIVER_AUTHENTICATION_CODE_SET = "PLACEHOLDER_TC_CI_IDU_X_AuthCode_S"

#Cab Activation set
ACTIVATE_CAB_BUTTON = "PLACEHOLDER_TC_CI_IDU_S_CabAct_S"

Listing 4.7: Signal values

DRIVER1_USERID = "10001"
DRIVER1_AUTHENTICATION_CODE = "1001"

4.4.3 Keywords

Our DCUTermSingleton class is used by Keywords component whose main purpose is to keep methods that would be used as a keywords for creating test cases. It is using methods from DCUTermSingleton class for execution of commands, tracking values, and getting a response from the DCUTerm tool. For our proof of concept tool we implemented simple keyword that is used for logging driver in one cab. For this action to be executed several signals needs to be set. These signals are stored in Signals Library.

Listing 4.8: Login keyword

```python
def Login(driver, cab):
    #Set IDU in proper state for testing
    DCUSingleton.setSignal(SignalLib.ENABLE_DEVELOPER, "0")
    DCUSingleton.setSignal(SignalLib.DEV_COMMAND, "1")
    DCUSingleton.setSignal(SignalLib.FORCE_SIGNALS, "1")

    #Set signals depending on the driver
    if (driver == "Driver 1"):
        driverID = SignalValuesLib.DRIVER1_USERID
        driverAuthenticationCode = SignalValuesLib.DRIVER1_AUTHENTICATION_CODE
    else:
        driverID = ""
        driverAuthenticationCode = ""

    #Set cab signal
    if (cab == "A1"):
        cabSignal = SignalLib.ACTIVATE_CAB_BUTTON
    else:
        cabSignal = ""

    DCUSingleton.addToWatchList(cabSignal)
    DCUSingleton.watchSignals(watchList)
    #Login driver
```
DCUSingleton.setSignal(SignalLib.DRIVER_USERID_SET, driverID)
DCUSingleton.setSignal(SignalLib.DRIVER_AUTHENTICATION_CODE_SET, driverAuthenticationCode)

#Activate the cab
DCUSingleton.setSignal(cabSignal, "1")

result = DCUSingleton.getResponse(cabSignal)
assert_equal(result, 1, 'Cab activation failed!')

4.4.4 Test cases

As mentioned before we used Robot Framework for creating our test cases. These test
cases have tabular structure and they are easy to read and understand. For our proof
of concept tool we created one test case, that is testing if a driver can login to the
cab. For performing this action previously implemented keyword "login" is used. For
asserting specific result of an action a keyword, "Should be equal", from standard Robot
Framework’s library BuiltIn can be used as well. Final test case is presented bellow.

Listing 4.9: Login test case

***Settings***
Library keywords//StandardActions.py

***Test Cases***
The driver can login to the cab
Login Driver 1 A1

4.5 Tool Discussion

The proposed tool, created using Python and Robot Framework [17], finally, consists out
of 4 main parts. The problems defined to be tackled have solutions proposed by the parts
of this tool. In order to improve the maintainability of the test script, a proposed Signal
Library provides a more direct way to handle signals by using aliases, this way a more
centralized and direct changes can be made to Signals in case that is needed. Use of this
approach also eliminates the possibility that some of the test scripts won’t be updated,
and as well creates a opportunity for the scripts to be reused by keeping the aliases and
importing a new signal names from a different project.

We have managed to improve the idea of reuse of the automated test scripts through
defining every action as a keyword, or a function. With proposed approach the re-usability,
that was being achieved up until now in TAF through use of snippets within a project, is
being upgraded to all of the actions, not only the most used ones, hence creating further
opportunity for creating more test scripts and in that way increasing the test coverage.

The final test script in the end mimic the Test Specification that is written by the test
designers. We feel strongly that using this approach couple of things can be achieved. The
re-usability of the test scripts doesn’t have to be limited to within project anymore but the
scripts can be reused between different projects as test specifications are usually similar, or
the same. The test cases don’t have to be rewritten by the test engineers to the automated
test scripts anymore, but the test designers can directly write test specifications that will
represent automated test scripts using created keywords developed and designed by test
engineers. We have managed to improve another thing by this proposed approach, the readability of the test script, especially for the third parties not involved in any part of the development process.
Chapter 5

Conclusion

In this research, we investigated testing processes inside Bombardier Transportation. The conducted investigation was concentrated on analyzing automated test scripts within the C30 project. The main focus was on the maintainability, readability and re-usability aspects. For these purposes, we analyzed how the scripts changed, the ways and procedures through which they are created and maintained. The findings of this analysis supported our initial idea that the most common changes within scripts were caused by changes in implementation specific data, more precisely Signals. These Signals can be of three types: Analog Signals, Digital Signals and Custom Signals, each of whom have specific information payloads and together have more than 81% of relevant line changes through versions from the next major payload groups which are Controls. Further results, made by investigating the Signal changes within their respective information payloads, pointed out that BusAdresses are parts of Analog and Digital Signals which change significantly more frequently, in fact they change approximately 2.4 times more than the average. This evidence points towards the potential maintainability improvements. Later into the research, the above mentioned conclusions were confirmed by the suspicions of a respondent in a conducted interview. From his experience, most of the changes were done to signals and to small parameters such as timings and changing signal name. They are several triggers for these changes, but the most common one is SCR (Software Change Request), a direct request for the change caused by implementation change or discovered flow during a testing process.

Through the investigation of the V&V process, the findings implied the usage of a highly complex and tedious process followed by an extensive documentation and reviewing. As a consequence of this kind of a process and tendencies for the whole testing process to reach a System level, some deviations from the official strategies and procedures have occurred. From the information gained through the interview, these deviations can be seen as a parallel, informal, testing process conducted alongside the official one. This parallel testing process skips the bureaucratic procedures of documentation and reviewing for the sake of sooner test results that are critical for faster production. The only documentation for this is represented in created automated test scripts, which are written from the pure experience of the testing engineers. Results like these imply the possibility of the procedures conducted in V&V process to be a bottleneck for the whole Testing, and possibly for the Development, life cycle.

Next, having in mind the results gained in previous analysis and the information gained about the tools used for the testing within Bombardier Transportation, we have proposed a natural language based tool for writing automated test scripts. The tool provides a
way for the test scripts to be written in a manner similar of writing Test Specifications. This is done using Robot Framework [17], building keywords which describe the technical behaviour of a certain function within a train. The strong point of this approach, as discussed, lies in a possibility where test engineers would be able to design keywords before the test cases are written by the test designers, and additionally these test cases won’t be needed to be translated from the Test Specification documents, but would rather be written directly as proposed tools scripts by the test designers. Additional strength of this tool lies in the way project specific data is handled, precisely signal names. The Signal names are each given respective aliases that are used for implementing mentioned keyword, and are stored in a centralized tool library from which they can be easily changed. This provides a basis for the better maintainability of the test scripts having in mind that these kinds of changes were the most common ones from the evidence gained, and additionally, it creates a possibility of a better automated test script reuse within as well as between the projects.

Finally, a number of potential weak points need to be considered. Firstly, the signal changes analysis was conducted only on a part of a single project. The changes behaviour might be different between the projects and, potentially, between the different test levels. Secondly, an interview conducted included a single responsible person. This is a threat to validity of the data gained from it due to its personal nature. Unfortunately, this threat regarding the information gained about the analyzed test scripts couldn’t be avoided as the respondent is the only person involved in writing them and using the tool for their creation. As the last point, it can be discussed that for the proposed test tool and the improvements it can potentially bring, there is no precise data about how large those improvements can be without showcasing it on a bigger scale. Up until this point it is only proven that the tool can function while writing the scripts in the way we intended.

### 5.1 Future work

Future work has to be done to establish precisely how much is the re-usability of test scripts improved by the proposed tool. Additional analysis should be conducted on top of test specifications written for the other projects. With this generic keywords and test cases should be extracted. We hope that using the information gained through it a precise conclusion can be made what level of re-usability is achievable between different projects. Also, we propose, for the further development of the tool proposed, that a supporting tool is built with an aim of handling signal names based on defined conventions in the Bombardier Transportation. All these improvements could eventually reduce the time that is needed for testing of new train systems, and could improve testing itself by enabling enrolment of third parties easier. One of the ideas for improving the re-usability between the projects is shown in Figure 4.4. This idea proposes a use of a device proxy library which would link the automated test cases with device/product specific data and even allowing the tests to be ran multiple devices simultaneously. This would be achievable through delegating test to be ran on different threads or machines.

From the perspective of the changes conducted on the automated test scripts, further work could include different projects within, and outside, Bombardier Transportation. It is important to expand the context of the conclusions made by analyzing the written test scripts for the C30 project. We suggest analyzing different levels of the testing process for the C30 project as well. This is important in order to validate the already conducted research.
Next point of interest for the research could be deeper analysis and understanding of the deviations between the documented V&V processes and the ones done in practice. The better understanding of these deviations and answers to why they are happening could serve as foundations for the future improvements related to the processes and strategies used in V&V. This way processes could end up being cheaper, resources and time wise, as well as improve the quality of V&V processes.
Appendices
Appendix A

Python Scripts

Listing A.1: Initial Analysis Python Script

```python
import xml.etree.ElementTree as ET
import os
import csv

# Directory path
my_path = os.path.abspath(os.path.dirname(__file__))
path = os.path.join(my_path, "./C30 Test VSS/TestScripts/")

# Dictionary for PayloadType count
payloadCount = {}

for filename in os.listdir(path):
    print(path + filename)
    if (filename == "vssver2.scc"): continue
    tree = ET.parse("./C30 Test VSS/TestScripts/"+filename)
    root = tree.getroot()

    # xsi type
    xsiType = r'\{http://www.w3.org/2001/XMLSchema-instance\}type'

    for payload in root.iter('Payload'):
        if (payload.attrib[xsiType] in payloadCount): payloadCount[payload.attrib[xsiType]] += 1
        else: payloadCount[payload.attrib[xsiType]] = 1

    # Print out the statistics in csv file
    with open('payload_type_stats_raw.csv', mode='w') as outputFile:
        writer = csv.writer(outputFile, delimiter = ';', quotechar = '"', quoting = csv.QUOTE_MINIMAL)
        for t, c in payloadCount.items():
            writer.writerow([t, c])
```

41
Listing A.2: Initial Formater Python Script

```python
import xml.etree.ElementTree as ET
import os

# xsiType const
xsiType = r'{http://www.w3.org/2001/XMLSchema-instance}type'

# Signal types
signals = [ 'AnalogSignal', 'CustomSignal', 'DigitalSignal' ]

# Control types
controls = [ 'SelectionControl', 'WaitControl', 'ParallelReactionStart',
             'ParallelSubReactionInit', 'ParallelSubReactionStart',
             'ParallelSubReactionEnd', 'ParallelReactionEnd',
             'AnalogControl', 'ReInitiateSimulations' ]

# Other types
others = [ 'SelectionIndicator', 'AnalogIndicator', 'Unit', 'Snippet' ]

pathR, dirsR, filesR = next(os.walk("./Scripts Sorted"))

for d in dirsR:
    path, dirs, files = next(os.walk(os.path.join(pathR, d)))
    for f in files:
        if (os.path.splitext(f)[1] == '.xml'):
            xmlFile = ET.parse('./Scripts Sorted/' + d + '/' + f)
        else: continue

        rootElement = xmlFile.getroot()

        print ('−−−−− ' + d + '....... ' + f)
        for payload in rootElement.iter('Payload'):
            if (payload.attrib[xsiType] in signals): payload.tag = 'Signal'
            if (payload.attrib[xsiType] in controls): payload.tag = 'Control'

        xmlFile.write('./Sorted/' + d + '/' + f)
```

Listing A.3: Second Formater Python Script

```python
import xml.etree.ElementTree as ET
import os

# xsiType const
xsiType = r'{http://www.w3.org/2001/XMLSchema-instance}type'

# Signal types
signals = {"AnalogSignal": "AnalogSignal", "CustomSignal": "CustomSignal", "DigitalSignal": "DigitalSignal"}

# Control types
controls = {"SelectionControl": "SelectionControl", "WaitControl": "WaitControl", "ParallelReactionStart": "ParallelReactionStart", "ParallelSubReactionInit": "ParallelSubReactionInit", "ParallelSubReactionStart": "ParallelSubReactionStart", "ParallelSubReactionEnd": "ParallelSubReactionEnd", "ParallelReactionEnd": "ParallelReactionEnd", "AnalogControl": "AnalogControl", "SemiControl": "SemiControl", "ReInitiateSimulations": "ReInitiateSimulations"}

# Other types
others = ["SelectionIndicator", "AnalogIndicator", "Unit", "Snippet"]

pathR, dirsR, filesR = next(os.walk("./Scripts Sorted"))

for d in dirsR:
    path, dirs, files = next(os.walk(os.path.join(pathR, d)))
    for f in files:
        if (os.path.splitext(f)[1] == '.xml'):
            xmlFile = ET.parse('./Scripts Sorted/' + d + '/' + f)
        else: continue
        rootElement = xmlFile.getroot()

        print('------> ' + d + ' .......

        for payload in rootElement.iter('Payload'):
            if (payload.attrib[xsiType] in signals):
                payload.tag = signals[payload.attrib[xsiType]]
            elif (payload.attrib[xsiType] in controls):
                payload.tag = controls[payload.attrib[xsiType]]

        xmlFile.write('./Sorted/' + d + '/' + f)
```

import os

from xmldiff import main, formatting

pathR, dirsR, filesR = next(os.walk("./Sorted"))

for d in dirsR:
    path, dirs, files = next(os.walk(os.path.join(pathR, d)))
    forT = formatting.DiffFormatter()
    with open("./DiffData/"+d+("raw_diff").txt", 'w') as outputFile:
        for i in range(2, len(files)-1):
            print("""" Version " + str(i) + ">" + str(i+1)+")
            diff = main.diff_files("./Sorted/"+d+"/"+d+" v"+str(i)+".xml "./Sorted/"+d+"/"+d+" v"+str(i+1)+".xml", formatter=
            forT, diff_options={"F": 0.5, "ratio_mode": "fast"})
            outputFile.writelines("""Version
            outputFile.writelines(diff)
            outputFile.writelines(""")
Listing A.5: Initial Difference Statistic Analysis Python Script

```python
import os
import csv

#options list
options = ['move', 'insert', 'update_text', 'rename', 'delete', 'insert_attribute']

#sumControl changes
sumControlCount = {}
#sumSignal changes
sumSignalCount = {}
#init counts
sumCount = {}
for i in options:
    sumControlCount[i] = 0
    sumSignalCount[i] = 0
    sumCount[i] = 0

with open('data.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)

    pathR, dirsR, fileR = next(os.walk('DiffData'))

for f in fileR:
    #Change counts per script
    FchangesCount = {}
    #Control changes
    FcontrolCount = {}
    #Signal changes
    FsignalCount = {}
    #init counts
    for i in options:
        FchangesCount[i] = 0
        FcontrolCount[i] = 0
        FsignalCount[i] = 0

    with open('./DiffData/' + f) as fp:

        #Change counts per script
        changesCount = {}
        #Control changes
        controlCount = {}
        #Signal changes
        signalCount = {}
        #init counts
        for i in options:
            changesCount[i] = 0
            controlCount[i] = 0
            signalCount[i] = 0

        line = fp.readline()
        while line:
            with open('./Stats/' + f + '.csv', mode='w', newline='') as outputFileV:
                writerV = csv.writer(outputFileV, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)
                new = False
            line = fp.readline()
```

45
if ("Version\n" == line):
    line = fp.readline()
if (new):
    writerV.writerow(['Signal'] + list(signalCount.values()))
    writerV.writerow(['Control'] + list(controlCount.values()))
    writerV.writerow(['Version All'] + list(changesCount.values()))
else:
    new = True
    writerV.writerow([line[:-1]] + options)
    for i in options:
        FcontrolCount[i] += controlCount[i]
        controlCount[i] = 0
        FchangesCount[i] += changesCount[i]
        changesCount[i] = 0
        FsignalCount[i] += signalCount[i]
        signalCount[i] = 0
    line = fp.readline()
    continue
if ("/Signal" in line):
    for i in options:
        if (i in line):
            signalCount[i] += 1
            changesCount[i] += 1
if ("/Control" in line):
    for i in options:
        if (i in line):
            controlCount[i] += 1
            changesCount[i] += 1
    line = fp.readline()
writerV.writerow(['Signal'] + list(signalCount.values()))
writerV.writerow(['Control'] + list(controlCount.values()))
writerV.writerow(['Version All'] + list(changesCount.values()))
for i in options:
    FcontrolCount[i] += controlCount[i]
    sumControlCount[i] += FcontrolCount[i]
    FchangesCount[i] += changesCount[i]
    sumCount[i] += FchangesCount[i]
    sumSignalCount[i] += FsignalCount[i]
    writerV.writerow(['ALL FOR SCRIPT'] + options)
    writerV.writerow(['Signal'] + list(FsignalCount.values()))
    writerV.writerow(['Control'] + list(FcontrolCount.values()))
    writerV.writerow(['All'] + list(sumCount.values()))
    writer.writerow([f] + options)
    writer.writerow(['Signal'] + list(FsignalCount.values()))
    writer.writerow(['Control'] + list(FcontrolCount.values()))
    writer.writerow(['All'] + list(FchangesCount.values()))
    writer.writerow(['ALL FOR ALL'] + options)
    writer.writerow(['Signal'] + list(sumSignalCount.values()))
    writer.writerow(['Control'] + list(sumControlCount.values()))
writer.writerow(['All'] + list(sumCount.values()))
Listing A.6: Second Difference Statistic Analysis Python Script

```python
import os
import csv

# options list
options = ["move", "insert", "update-text", "rename", "delete", "insert-attribute"]

# signals list
signals = {"AnalogSignal", "CustomSignal", "DigitalSignal"}

# controls list
controls = {"SelectionControl", "WaitControl", "ParallelReactionStart", "ParallelSubReactionInit", "ParallelSubReactionStart", "ParallelSubReactionEnd", "ParallelReactionEnd", "AnalogControl", "ReInitiateSimulations"}

# sumControl changes
sumControlsCount = {}

# sumControlCategories changes
sumControlCategoriesCount = {}

# sumSignal changes
sumSignalCount = {}

# sumSignalCategories changes
sumSignalCategoriesCount = {}

# init counts
sumCount = {}

for j in signals:
    sumSignalCategoriesCount[j] = {}
    for i in options:
        sumSignalCount[i] = 0
        sumControlsCount[i] = 0
        sumControlCategoriesCount[i] = {}
        sumSignalCategoriesCount[j][i] = 0
        sumCount[i] = 0

for k in controls:
    sumControlCategoriesCount[k] = {}
    for i in options:
        sumControlCategoriesCount[k][i] = 0

with open('data.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)

    pathR, dirsR, fileR = next(os.walk("./DiffData"))

for f in fileR:
    # Change counts per script
    FchangesCount = {}
    # Control changes
    FcontrolCount = {}
    # Signal changes
    FsignalCount = {}
    # SignalCategories changes
    FsignalCategoriesCount = {}
    # ControlCategories changes
    FcontrolCategoriesCount = {}
```

48
#init counts
for j in signals:
    FsignalCategoriesCount[j] = {}
for i in options:
    FchangesCount[i] = 0
    FcontrolCount[i] = 0
    FsignalCount[i] = 0
    FsignalCategoriesCount[j][i] = 0
for k in controls:
    FcontrolCategoriesCount[k] = {}
for i in options:
    FcontrolCategoriesCount[k][i] = 0

with open('./DiffData/' + f) as fp:

    #Change counts per script
    changesCount = {}
    #Control changes
    controlCount = {}
    #Signal changes
    signalCount = {}
    #SignalCategories changes
    signalCategoriesCount = {}
    #ControlCategories changes
    controLCategoriesCount = {}
    #init counts
    for j in signals:
        signalCategoriesCount[j] = {}
        for i in options:
            changesCount[i] = 0
            controlCount[i] = 0
            signalCount[i] = 0
            signalCategoriesCount[j][i] = 0
        for k in controls:
            controLCategoriesCount[k] = {}
            for i in options:
                controLCategoriesCount[k][i] = 0

            line = fp.readline()
        with open('./Stats/' + f + '.csv', mode='w', newline='') as outputFileV:
            writerV = csv.writer(outputFileV, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)
            new = False
            while line:
                if ('Version
' == line):
                    line = fp.readline()
                if (new):
                    writerV.writerow(['Signal'] + list(signalCount.values()))
                    for i in signals:
                        writerV.writerow([i] + list(signalCategoriesCount[i].values()))
                    writerV.writerow(['Control'] + list(controlCount.values()))
                    new = False
                line = fp.readline()
for i in controls:
    writerV.writerow([i] + list(controlCategoriesCount[i].values()))
    writerV.writerow(['Version All'] + list(changesCount.values()))

else:
    new = True
writerV.writerow([line[:−1]] + options)
for j in signals:
    for i in options:
        FcontrolCount[i] += controlCount[i]
        controlCount[i] = 0
        FchangesCount[i] += changesCount[i]
        changesCount[i] = 0
        FsignalCount[i] += signalCount[i]
        signalCount[i] = 0
        FsignalCategoriesCount[j][i] += signalCategoriesCount[j][i]
        signalCategoriesCount[j][i] = 0

for k in controls:
    for i in options:
        FcontroLCategoriesCount[k][i] += controLCategoriesCount[k][i]
        controLCategoriesCount[k][i] = 0

line = fp.readline()
continue
for k in signals:
    varIfSignal = '/ ' + k
    if (varIfSignal in line):
        for i in options:
            if (i in line):
                signalCount[i] += 1
                changesCount[i] += 1
                signalCategoriesCount[k][i] += 1

for j in controls:
    varIfControl = '/ ' + j
    if (j in line):
        for i in options:
            if (i in line):
                controlCount[i] += 1
                changesCount[i] += 1
                controLCategoriesCount[j][i] += 1

line = fp.readline()
writerV.writerow(['Signal'] + list(signalCount.values()))
for i in signals:
    writerV.writerow([i] + list(signalCategoriesCount[i].values()))
writerV.writerow(['Control'] + list(controlCount.values()))
for i in controls:
    writerV.writerow([i] + list(controLCategoriesCount[i].values()))
writerV.writerow(['Version All'] + list(changesCount.values()))
for i in options:
    FcontrolCount[i] += controlCount[i]
sumControlsCount[i] += FcontrolCount[i]
FchangesCount[i] += changesCount[i]
sumCount[i] += FchangesCount[i]
FsignalCount[i] += signalCount[i]
sumSignalCount[i] += FsignalCount[i]
for j in signals:
    for i in options:
        FsignalCategoriesCount[j][i] +=
        signalCategoriesCount[j][i]
        sumSignalCategoriesCount[j][i] +=
        FsignalCategoriesCount[j][i]
for k in controls:
    for i in options:
        FcontroLCategoriesCount[k][i] +=
        controLCategoriesCount[k][i]
        sumControlCategoriesCount[k][i] +=
        FcontroLCategoriesCount[k][i]
writerV.writerow(['ALL FOR SCRIPT'] + options)
writerV.writerow(['Signal'] + list(FsignalCount.values()))
for i in signals:
    writerV.writerow([i] + list(FsignalCategoriesCount[i].values()))
writerV.writerow(['Control'] + list(FcontrolCount.values()))
for i in controls:
    writerV.writerow([i] + list(FcontroLCategoriesCount[i].values()))
writerV.writerow(['All'] + list(FchangesCount.values()))
writerV.writerow(['Signal'] + list(FsignalCount.values()))
for i in signals:
    writer.writerow([i] + list(sumSignalCategoriesCount[i].values()))
writer.writerow(['Control'] + list(sumControlsCount[i].values()))
for i in controls:
    writer.writerow([i] + list(sumControlCategoriesCount[i].values()))
    writer.writerow(['All'] + list(sumCount.values()))
import os
import csv

# options list
options = ["move", "insert", "update-text", "rename", "delete", "insert-attribute"]

# signals list
signals = ["AnalogSignal", "CustomSignal", "DigitalSignal"]

# controls list
controls = ["SelectionControl", "WaitControl", "ParallelReactionStart", "ParallelSubReactionInit", "ParallelSubReactionStart", "ParallelSubReactionEnd", "ParallelReactionEnd", "AnalogControl", "SemiControl", "ReInitiateSimulations"]

# sumControl changes
sumControlsCount = {}

# sumControlCategories changes
sumControlCategoriesCount = {}

# sumSignal changes
sumSignalCount = {}

# sumSignalCategories changes
sumSignalCategoriesCount = {}

# init counts
sumCount = {}

for j in signals:
    sumSignalCategoriesCount[j] = {}
    for i in options:
        sumSignalCount[i] = 0
        sumControlsCount[i] = 0
        sumControlCategoriesCount[i] = {}
        sumSignalCategoriesCount[j][i] = 0
        sumCount[i] = 0

for k in controls:
    sumControlCategoriesCount[k] = {}
    for i in options:
        sumControlCategoriesCount[k][i] = 0

with open('data.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)

    pathR, dirsR, fileR = next(os.walk("../DiffData"))

    for f in fileR:
        # Change counts per script
        FchangesCount = {}
        # Control changes
        FcontrolCount = {}
        # Signal changes
        FsignalCount = {}
        # SignalCategories changes
        FsignalCategoriesCount = {}
        # ControlCategories changes
        FcontrolCategoriesCount = {}

Listing A.7: Category Difference Analysis Python Script
FcontrolCategoriesCount = {}

#init counts
for j in signals:
    FsignalCategoriesCount[j] = {}
for i in options:
    FchangesCount[i] = 0
    FcontrolCount[i] = 0
    FsignalCount[i] = 0
    FsignalCategoriesCount[j][i] = 0
for k in controls:
    FcontrolCategoriesCount[k] = {}
for i in options:
    FcontrolCategoriesCount[k][i] = 0

with open('../DiffData/' + f) as fp:

    #Change counts per script
    changesCount = {}
    #Control changes
    controlCount = {}
    #Signal changes
    signalCount = {}
    #Signal Categories changes
    signalCategoriesCount = {}
    #ControlCategories changes
    controLCategoriesCount = {}
    #init counts
    for j in signals:
        signalCategoriesCount[j] = {}
    for i in options:
        changesCount[i] = 0
        controlCount[i] = 0
        signalCount[i] = 0
        signalCategoriesCount[j][i] = 0
    for k in controls:
        controLCategoriesCount[k] = {}
    for i in options:
        controLCategoriesCount[k][i] = 0

    line = fp.readline()
    with open('../Stats/' + f + '.csv', mode='w', newline='') as outputFileV:
        writerV = csv.writer(outputFileV, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)
        new = False
        while line:
            if ('Version\n' == line):
                line = fp.readline()
            if (new):
                writerV.writerow(['Signal'] + list(signalCount.values()))
                for i in signals:
                    writerV.writerow([i] + list(signalCategoriesCount[i].values()))
                writerV.writerow(['Control'] + list(controlsCount.values()))
            new = True
            line = fp.readline()
controlCount.values())
for i in controls:
    writerV.writerow([i] + list(controLCategoriesCount[i].values()))
writerV.writerow(['Version All'] + list(changesCount.values()))
else:
    new = True
writerV.writerow([line[:len(i)] + options])
for j in signals:
    for i in options:
        FcontrolCount[i] += controlCount[i]
        controlCount[i] = 0
        FchangesCount[i] += changesCount[i]
        changesCount[i] = 0
        FsignalCount[i] += signalCount[i]
        signalCount[i] = 0
        FsignalCategoriesCount[j][i] +=
        signalCategoriesCount[j][i] = 0
    for k in controls:
        for i in options:
            if i in line:
                signalCount[i] += 1
                changesCount[i] += 1
                signalCategoriesCount[k][i] += 1
                line = fp.readline()  
continues
for k in signals:
    varIfSignal = '/' + k
    if (varIfSignal in line):
        for i in options:
            if (i in line):
                signalCount[i] += 1
                changesCount[i] += 1
                signalCategoriesCount[k][i] += 1
for j in controls:
    varIfControl = '/' + j
    if (j in line):
        for i in options:
            if (i in line):
                controlCount[i] += 1
                changesCount[i] += 1
                controLCategoriesCount[k][i] += 1
                line = fp.readline()  
writerV.writerow(['Signal'] + list(signalCount.values()))
for i in signals:
    writerV.writerow([i] + list(signalCategoriesCount[i].values()))
writerV.writerow(['Control'] + list(controlCount.values()))
for i in controls:
    writerV.writerow([i] + list(controLCategoriesCount[i].values()))
writerV.writerow(['Version All'] + list(changesCount.values()))
for i in options:
```python
FcontrolCount[i] += controlCount[i]
sumControlsCount[i] += FcontrolCount[i]
FchangesCount[i] += changesCount[i]
sumCount[i] += FchangesCount[i]
FsignalCount[i] += signalCount[i]
sumSignalCount[i] += FsignalCount[i]

for j in signal:
    for i in options:
        FsignalCategoriesCount[j][i] += signalCategoriesCount[j][i]
        sumSignalCategoriesCount[j][i] += FsignalCategoriesCount[j][i]

for k in control:
    for i in options:
        FcontroLCategoriesCount[k][i] += controLCategoriesCount[k][i]
        sumControlCategoriesCount[k][i] += FcontroLCategoriesCount[k][i]

writerV.writerow(['ALL FOR SCRIPT'] + options)
writerV.writerow([' Signal'] + list(FsignalCount.values()))

for i in signal:
    writerV.writerow([i] + list(FsignalCategoriesCount[i].values()))
    writerV.writerow([' Control'] + list(FcontroLCategoriesCount[i].values()))

for i in control:
    writerV.writerow([' All'] + list(FcontroLCategoriesCount[i].values()))
    writerV.writerow([' Signal'] + list(sumSignalCategoriesCount[i].values()))

for i in signal:
    writerV.writerow([i] + list(sumSignalCategoriesCount[i].values()))

for i in control:
    writerV.writerow([i] + list(sumControlCategoriesCount[i].values()))

writerV.writerow([' ALL FOR ALL'] + options)
writerV.writerow([' Signal'] + list(sumSignalCount.values()))
```

55
Listing A.8: Analog Signal Difference Analysis Python Script

```python
import os
import csv

# Analog Signal Tags
analogSignalTags = [

# options list
options = [
    "move", "insert", "update-text", "rename", "delete", "insert-attribute"]

# Analog count
for i in analogSignalTags:
dCount[i] = {}
for j in options:
dCount[i][j] = 0

with open('AnalogSignalData.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)
    writer.writerow(['All Analog Signals'] + options)

    for i in analogSignalTags:
        writer.writerow([i] + list(dCount[i].values()))

    pathR, dirsR, fileR = next(os.walk("./DiffData"))

    for f in fileR:
        with open("./DiffData/" + f) as fp:
            line = fp.readline()
            while line:
                signalW = "/AnalogSignal"
                if (signalW in line):
                    for i in analogSignalTags:
                        tmp = "/" + i
                        if(tmp in line):
                            for j in options:
                                if (j in line):
                                    dCount[i][j] += 1
                                break
                        line = fp.readline()
                writer.writerow(['All Analog Signals'] + options)
        for i in analogSignalTags:
            writer.writerow([i] + list(dCount[i].values()))
```

import os
import csv

# Digital Signal Tags
digitalSignalTags = ['Description', 'FullPath', 'VCSPath', 'BusType', 'MWTSignalName', 'ReactionTimeOutInMS', 'SetSignal', 'LockSignal', 'MeasureDurationInMS', 'MWTVersion', 'VCSVersion', 'VCSID', 'RequireFlankOnReaction', 'ToleranceInMS', 'WaitInMS', 'BusID', 'BusName', 'BusAddress', 'BusAddressOffset', 'DataType', 'DCUIPSignalName', 'SignalAddress', 'SignalType', 'UnitTypeName', 'SignalDirection', 'UnitID', 'IsHex', 'AddressingScheme', 'CurrentValue']

# options list
options = ['move', 'insert', 'update-text', 'rename', 'delete', 'insert-attribute']

# Digital count
dCount = {}

for i in digitalSignalTags:
    dCount[i] = {}
    for j in options:
        dCount[i][j] = 0

with open('DigitalSignalData.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)

    pathR, dirsR, fileR = next(os.walk("./DiffData"))

    for f in fileR:
        with open('"./DiffData/" + f) as fp:
            line = fp.readline()
            while line:
                signalW = "/DigitalSignal"
                if (signalW in line):
                    for i in digitalSignalTags:
                        tmp = "/#" + i
                        if (tmp in line):
                            for j in options:
                                if (j in line):
                                    dCount[i][j] += 1
                                break
                            line = fp.readline()
                writer.writerow(['All Digital Signals'] + options)
                for i in digitalSignalTags:
                    writer.writerow([i] + list(dCount[i].values()))
import os
import csv

#Custom Signal Tags
customSignalTags = ["MWTSignalName", "ReactionTimeOutInMS", "CurrentValue", "BetweenValue", "CurrentReactionOperator", "MeasureDurationInMS", "MarkSignal", "RequireFlankOnReaction", "ToleranceInMS", "WaitInMS"]

#Custom CCUnit Tags
ccUnitTags = ["Name", "IPAddress", "Port", "IsValidSignalMWTName", "UseValidSignal", "LogTaskLevel", "Version"]

#options list
options = ["move", "insert", "update-text", "rename", "delete", "insert-attribute"]

dCount = {}

#custom key
cUnitColName = "CCUUnitCollection-Name"

for i in customSignalTags:
    dCount[i] = {}
    for j in options:
        dCount[i][j] = 0

for i in ccUnitTags:
    name = "CCUUnit-"+i
    dCount[name] = {}
    for j in options:
        dCount[name][j] = 0

for i in ccUnitTags:
    dCount[ccUnitColName] = {}
    for j in options:
        dCount[ccUnitColName][i] = 0

with open("CustomSignalData.csv", mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)

    pathR, dirsR, fileR = next(os.walk("./DiffData/"))

    for f in fileR:
        with open('./DiffData/" + f) as fp:
            line = fp.readline()
            while line:
                signalW = "/CustomSignal"
                if (signalW in line):
                    for i in customSignalTags:
                        tmp = "/" + i
                        if (tmp in line):
                            for j in options:
                                if (j in line):
                                    dCount[i][j] += 1
                                    break
                            for i in ccUnitTags:
                                tmp = "/" + i
                                if (tmp in line and "/CCUUnits" in line):
for j in options:
    if (j in line):
        name = "CCUUnit-"+i
        dCount[name][j] += 1
        break
    break

if ("/Name" in line and not "/CCUUnits" in line):
    for j in options:
        if (j in line):
            dCount[ccUnitColName][j] += 1
            break

line = fp.readline()
writer.writerow(['All Custom Signals'] + options)
writer.writerow([ccUnitColName] + list(dCount[ccUnitColName].values()))
for i in ccUnitTags:
    name = "CCUUnit-"+i
    writer.writerow([name] + list(dCount[name].values()))
for i in customSignalTags:
    writer.writerow([i] + list(dCount[i].values()))
```python
import os
import csv

# Selection Control Tags
selectionControlTags = ["Description","FullPath","VCSPath","SelectedControlPositionIndex","SetControl","LockControl","Comment","IsLockable","ContainerName","VCSVersion","VCSID"]

# Control Position Tags
cPositionTags = ["Description","PositionValue"]

# options list
options = ["move", "insert", "update-text", "rename", "delete", "insert-attribute"]

# Selection Control count
dCount = {}

for i in selectionControlTags:
    dCount[i] = {}
    for j in options:
        dCount[i][j]=0

for i in cPositionTags:
    name = "ControlPosition-"+i
    dCount[name] = {}
    for j in options:
        dCount[name][j]=0

with open('SelectionControlData.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting=csv.QUOTE_MINIMAL)

pathR, dirsR, fileR = next(os.walk("./DiffData"))

for f in fileR:
    with open('./DiffData/' + f) as fp:
        line = fp.readline()
        while line:
            signalW = "/SelectionControl"
            if (signalW in line):
                for i in selectionControlTags:
                    tmp = "/" + i
                    if(tmp in line):
                        for j in options:
                            if (j in line):
                                dCount[i][j] += 1
                                break
                break
            for i in cPositionTags:
                tmp = "/" + i
                if(tmp in line and "/ControlPositions" in line):
                    for j in options:
                        if (j in line):
                            name = "ControlPosition-"+i
                            dCount[name][j] += 1
                        break
```

Python Scripts

break

line = fp.readline()
writer.writerow(['All Selection Control'] + options)
for i in cPositionTags:
    name = "ControlPosition-" + i
    writer.writerow([name] + list(dCount[name].values()))
for i in selectionControlTags:
    writer.writerow([i] + list(dCount[i].values()))
Listing A.12: Analog Control Difference Analysis Python Script

```python
import os
import csv

# Analog Control Tags
analogControlTags = [
    "Description", "FullPath", "VCSPath", 
    "SelectedControlPositionIndex", "SetControl", "LockControl", "Comment", 
    "IsLockable", "ContainerName", 
    "HasPresetValues"
]

# Control Position Tags
cPositionTags = ["Description", "PositionValue"]

# options list
options = ["move", "insert", "update-text", "rename", "delete", "insert-attribute"]

# AnalogControl count
dCount = {}

for i in analogControlTags:
    dCount[i] = {}
    for j in options:
        dCount[i][j] = 0

for i in cPositionTags:
    name = "ControlPosition-"+i
    dCount[name] = {}
    for j in options:
        dCount[name][j] = 0

with open( 'AnalogControlData.csv', mode='w', newline='') as outputFile:
    writer = csv.writer(outputFile, delimiter=';', quotechar='"', quoting = csv.QUOTE_MINIMAL)

pathR, dirsR, fileR = next(os.walk("./DiffData"))

for f in fileR:
    with open('./DiffData/' + f) as fp:
        line = fp.readline()
        while line:
            signalW = "/AnalogControl"
            if (signalW in line):
                for i in analogControlTags:
                    tmp = "/" + i
                    if(tmp in line):
                        for j in options:
                            if (j in line):
                                dCount[i][j] += 1
                            break
                break
            for i in cPositionTags:
                tmp = "/" + i
                if(tmp in line and "/ControlPositions" in line):
                    for j in options:
                        if (j in line):
                            name = "ControlPosition-"+i
                            dCount[name][j] += 1
                        break
```

62
break

line = fp.readline()
writer.writerow(['All Analog Control'] + options)
for i in cPositionTags:
    name = "ControlPosition-" + i
    writer.writerow([name] + list(dCount[name].values()))
for i in analogControlTags:
    writer.writerow([i] + list(dCount[i].values()))
Appendix B

Interview Transcripts

B.1 Interview 1

Q: Do you want to be anonymous in our report?
A: It does not matter.

Q: Could you introduce yourself, your position and role inside Bombardier Transportation?
A: My name is Markus Einarson, and I have been working in the area of testing the TCMS software. Both manual and automation testing using tools that were developed in Bombardier Transportation.

Q: Can you briefly explain to us the testing lifecycle that you are involved in? What are the inputs and outputs artifacts of each phase in it? What tools are used in each phase?
A: Within C30 projects there are different types of automation testing. One of them is general automated regression testing, that is run every time when new TCMS version is built. The main purpose of this testing is to make sure that new changes didn’t affect previous functions that were working fine. Script that I developed is a drive cycle. Its aim was to simulate normal usage of the train, from opening doors, starting the train, driving it, and finally stopping it. It takes 10-15 minutes to execute. This script was created based on the experience not documented test cases.

Q: So, there is not any documentation of it?
A: There are three types of regression testing. For previously mentioned one only documentation is the script itself. For the second one test cases are documented in DOORS software, and they are previously executed manually. They are split into two types, regular and safety test cases.

Q: In your opinion what are the important parts of DOORS documentation that have an influence on your creation of test scripts?
A: Test cases are written based on the requirements for TCMS. Their goal is to cover all TCMS requirements. These test cases were run manually. My job is to create an automated test script that would give the same result as manually executed test cases.

Q: Can you give us your general thought on tool that was used for creating automated test scripts, its advantages and drawbacks?
A: Three tools are part of the TAF tool, which is used for creating automated test scripts. TAF script editor, TAF execution manager, and TAF test engine.
All of them are implemented in house and they are closely integrated with our manual testing environment VCS (Vehicle Control Simulator). That is a big benefit, they use the same data. Instead of writing test code you can select specific signals, buttons to push, etc... That is making writing test scripts much easier.

Q: Is it easy for the third parties to read scripts that were written using TAF?
A: You must learn how to read them. There is comment item where you can add free text to make it easier to understand specific section. The drawback is that you need to have test script editor installed in order to read it. There is no good export format in TAF.

Q: How do you find the re-usability of the test script?
A: In case you have created script, you can use it as a base for another script. There are several levels inside the script. Lowest ones are actions and reactions. If you want to perform certain action several times you can create a snippet, which is subfunction. If you update snippet you can easily update scripts where that snippet was used.

Q: When does your job start in the process of project development?
A: I think that I should start at day one of the project development, as soon as you have one build of TCMS. In reality, is different. I start when there is more or less TCMS software when it is possible to drive the train. Earlier you start you have more maintenance to do on the test scripts, and that is the drawback of starting too early. For other types I cannot start before DOORS test cases are created and executed manually. VCS, a tool that is trying to simulate the surrounding of the TCMS, has the development cycle itself and it is updated through the project. It is created based on its own design which is not in place at day one.

Q: How frequent are the changes during the project development and how do they affect your job, and how long does it take to modify test script after they occur?
A: In practice are trigger mostly by SCR (Software Change Request). SCRs are written for software changes and updates of overall test cases. When it comes to DOORS test cases if they are updated it is quite simple to update the test script. With automated regression testing, normally it is not changing, because it is simulating driving of the train. But it happens, for example when it is failing during the execution. We run the script during the night and if something fails the script must be updated. Sometimes the scripts must be updated because of changes that are meant to be.

Q: How much time approximately do you need to make changes?
A: The actual update is usually the small part, the bigger part is the investigation in what actually needs to be updated. Most of the time when the test is failing you must investigate the details of the recorder logs and repeat manually same actions and reproduce the error. Is the original script to tight on timing so you need to define a bigger time span, or something similar? But the actual change is the small part. It varies from few minutes to several hours, even days depending what kind of the update it is?

Q: How often do situations occur where you must discard parts of the scripts and how often do situations occur where you must expand the scripts or add more detail?
A: Main changes when you have reached certain maturity of the automated test scripts, so you have the drive cycle than there is not often to remove or add steps, it’s mostly updating parameters such as tweaking the timings. In certain times, of course, certain sections are added or updated if new functionalities were added, but this isn’t often. During the C30 project we had a big change within it where we had implementation for certain Signaling System and then it was decided to use another one. This triggered big changes in the VCS, the simulation of the surroundings, which also affected the TAF scripts.

Q: You mentioned the changes to the Signaling System, during our previous phase of the research we came to the conclusion that signals are the ones that change the most, precisely the Bus Addresses.

A: Signaling System isn’t the part of the TCMS, it is another system that is in charge of making the system sage in the subway. So if the driver isn’t respecting the speed limits Signaling System breaks the train or if the driver is driving where he should not it brakes the train. This system is specific on this project for the subways and it has been decided to be built upon the subway structure but in the middle of the project the customer has decided to stick to the old system which made us adopt to that. This effected a lot VCS and TAF scripts. When it comes to signals, they come from VCS which simulates the entire signal interfaces. These interfaces during the TCMS project are changing many times and then that is documented in different interface documentation which is imported into VCS application. It is part of VCS databases which is part of TAF environment, if this database is updated then TAF's scripts must also be updated. So that is true, and it’s continuous happening frequently during the development lifecycle of the TCMS.

Q: Are those changes updated automatically or manually in the scripts?

A: You could say they are happening semi-automatically, it depends on what kind of changes it is. Firstly, VSC development is taking care of new interfaces, so the new interfaces are imported in the new VSC application and then there is a new VCS release, a new version, which has a new VSC database. That database must be pointed out manually in the Test Script Editor, and then signals must be replaced. So, it depends what kind of change it is, if It’s the name change or adding or removing signals.

Q: Is it time consuming to do those kinds of changes?

A: Yes, it can be. But, it doesn’t necessarily mean that updating interface triggers changes in the scripts, it only relates to the signals that are in the scripts itself. It needs to be analyzed again what has been the change. There is an automatic handling of signal changes in the scripts but you need to manually take those changes into the scripts, that doesn’t happen automatically.

Q: Do you have any opinion on what might be a good idea to reduce the time needed for those kind of changes.

A: There is a good support for TAF scripts within the tools which helps you to identify updated signals. It sounded like it was totally manually done but that is not the case. If you have, for example, 50 scripts, nothing is updated automatically you must open up each and every script and then if you have pointed out the VCS database then the tool will identify that a specific signal has been updated since creation and then you are prompted if you want to update the script. There is a big help in that and I can not really see that
it’s possible to make that even simpler. What you need to do manually, when you have the signal identified as updated then you need to point out that new database and then manually have to open all the scripts to update. If it was possible to update all of the scripts automatically with new signals it raises the question if the functionality of the scripts is valid. It is necessary to have a look in to each and every updated.

Q: We had an idea where we would take a data driven testing concept, the idea is to make an library of signals with aliases, names, for each signal and those could be used in the scripts while the signal addresses would be updated within that library.

A: That’s not a bad idea. With that it could help to increase the reusability between the projects. In many cases in the projects you have the same type of signals with different names and addresses. In case you had a generic name for that signal, and you create a script using that name and in specific application you select the exact address that would definitely a good idea. It’s not easy if you compare different projects, for instance C30 with the BF490, trains aren’t identical, they are quite different, so some test cases might be possible to reuse but not so many.

Q: While writing project documentation like Test Specifications do you use any semantics, formal or informal?

A: Inside the projects, definitely. If you look the documentation in DOORS for C30 project, although it’s written by multiple people you can notice big similarities. Authors use standardized way of writing. Partly it differs between the projects and it’s not so synchronized. It’s usually informal and within the project.

Q: Do you think there is a way to standardize that between projects?

A: Technically it is possible, it is in organization and decisions should be made to use the same tools between projects. And that is also a difficulty in my opinion.

Q: Our second idea is to use the semantics in writing the Test Specification to create proof of concept testing tool based on the natural language. It would be keyword oriented and these keywords correspond to the ones from Test Specification. They would be connected, in this case, to VCS and the point would be to concentrate more on the development of the connection part and reduce the time for rewriting test specifications into the test scripts. Do you thin that would help and improve the process in the way?

A: Absolutely. I have been involved for many years in the development of the test tools within the company. I think what you suggest it sounds really great when it comes to the point that it’s mature. I think you can prove that it works from the technical point of view, but the difficulty might be to actually get it into practice with big scale, to use it with new projects.

Q: One of the motivations for this idea is having in mind review processes from third parties or licensing entities etc. and importance of tests being readable for them. In your case is that important? Are the test scripts reviewed and how long does the process last?

A: I think that would be really good if it’s like that. I am not involved in the assessment of the scripts, I am more involved in assessment of the VCS application which is on the side of TAF environment. There we have external
assessors, we don’t have that for TAF at all. We are currently running the TAF scripts as a Quality check for the software, after that we are manually testing and that is the part of validation and assessment. Therefore, TAF isn’t shown to assessors. The DOORS test on the other hand are the ones that are scripted and are under assessment but not the scripts itself. In the future automatic test might be a part of assessment process but not on the C30 project.


Bibliography


Bibliography


