Escola Tècnica Superior d'Enginyeria Electrònica i Informàtica La Salle

Treball Final de Màster

Màster Universitari en Enginyeria de Telecomunicació

Continuous integration framework for automotive software platforms

Alumne

Moisés Ojeda Esquerdo

Professor Ponent

David Badia Folguera

ACTA DE L'EXAMEN DEL TREBALL FI DE CARRERA

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D.Moisés Ojeda Esquerdo

va exposar el seu Treball de Fi de Carrera, el qual va tractar sobre el tema següent:

Continuous integration framework for automotive software platforms

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL

VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

Abstract

The mobility software department of IDNEO is in charge of developing and maintaining software for multiple mobility projects, mainly within the automotive industry.

The software developing process in this industry is standardized and follows a set of rules required by the regulators.

In order to reduce risks due to human errors and increase the competitivity of the company it is key to have this process as much automated as possible. For this reason, the team works within a continuous integration framework that is partially automated. However, this framework is not ideal and can be further improved.

The goal of this document is first to define and analyse the current continuous integration framework and address weak points where the it can be improved. Secondly to implement the key parts of the process that have been defined and finally to test quantitatively this new framework.

Resum

El departament de software de Mobility d'IDNEO està a càrrec de desenvolupar i mantenir software per a diferents projectes de mobilitat, la majoria dins l'indústria de l'automoció.

El procés de desenvolupament de software en aquesta indústria està estandarditzat i segueix un seguit de normes requerides pels reguladors.

És clau tenir aquest procés el més automatitzat possible per tal de reduir els riscos deguts a errors humans i incrementar la competitivitat de la companyia. Per aquesta raó, l'equip treballa dins un entorn d'integració continua que ja està automatitzat en certa mesura. Si més no, aquest entorn no és l'ideal i té molt marge per millorar.

L'objectiu d'aquest document és primer definir i analitzar l'estat del procés actual i posar en relleu els punts febles on pot millorar. Després implementar les parts clau del procés que s'han definit i finalment testejar quantitativament el nou entorn implementat.

Resumen

El departamento de software de Mobility de IDNEO está a cargo de desarrollar y mantener software para diferentes proyectos de mobilidad, en su mayoria dentro de la indústria de la automoción.

El proceso de desarrollo de software en esta industria está estandarizado y sigue una serie de normas requeridas por los reguladores.

Es clave tener este proceso lo más automatizado posible para poder reducir los riesgos debidos a errores humanos e incrementar la competitividad de la compañía. Por esta razón, el equipo trabaja dentro de un entorno de integración continua que ya está parcialmente automatizado.

Sin embargo, este entorno no es el ideal y tiene mucho margen de mejora.

El objetivo de este documento es primero definir y analizar el estado del proceso actual y poner en relieve los puntos débiles donde puede mejorar. Seguidamente implementar las partes claves del proceso definidas y finalmente testear cuantitativamente el nuevo entorno implementado.

Content

1 Introduction 7
2 Industry standards 1
2.1 The engineering process: ISO/IEC 15504[2][3] and ISO/IEC 12207[4] 1
2.1.1 Capability determination 1
2.1.2 The “V” model: 2
2.2 Quality standards 6
2.2.1 Static analysis: MISRA C Rules 6
2.2.2 Runtime analysis 7
2.2.3 Unitary tests 7
2.2.4 Memory map 8
2.2.5 Task profiling 8
2.2.6 Compiler warnings 9
3 Used software tools 11
3.1 Rational Doors 11
3.2 Vector Canape 12
3.3 Vector Canoe 12
3.4 Jenkins 13
3.5 Python scripts 13
3.6 Excel Macros 13
3.7 SVN 14
3.8 Unity 14
3.9 Compilers 14
3.10 EDI 15
3.11 Matlab Simulink 15
4 Initial setup 16
4.1 Initial infrastructure 16
4.1.1 Development PC 17
4.1.2 Validation PC 18
4.1.3 Canoe Server 18
4.1.4 Jenkins Slave 18
4.1.5 Client 18

4.2 Initial process 18
4.2.1 Milestone 2.. 19
4.2.2 Milestone 3.. 22
5 Improved setup 23
5.1 Compiler Warnings Script.. 24
5.2 ITP from Jenkins 24
5.3 Canoe for validation 24
5.4 SENT emulator.. 24
5.5 SVN-Jenkins bug solved.. 25
5.6 Jenkins server without slave 25
6 Economic impact 26
6.1 Changes performed.. 26
6.1.1 Compiler Warnings Script.. 26
6.1.2 ITP from Jenkins 27
6.1.3 Canoe for validation and SENT emulator 27
6.1.4 SVN-Jenkins bug solved.. 27
6.2 Economic results 28
6.2.1 Reduction of spending 28
6.2.2 Productivity 28
7 Future improvements 29
7.1 Task profiling 29
7.2 Jenkins using scripts 29
7.3 Git.. 29
8 Conclusions 30
9 References.. 31

Acronyms

- CAPL: Communication Access Programming Language
- CAR: Bug opened by the client
- DUT: Device Under Test
- ECU: Electronic Control Unit
- GPIO: General Purpose Input Output pin
- GUI: Graphical User Interface
- HTML: Hypertext Markup Language
- HTTP: HyperText Transfer Protocol
- IC: In Circuit
- IQ: Request from the client
- ITP: Integration Test Plan
- MDF: Signal storage file
- MISRA: Motor Industry Software Reliability Association
- MS: Milestone
- OEM: Original Equipment Manufacturer
- QA: Quality department
- RAM: Random Access Memory
- ROM: Read-Only Memory
- SW: Software
- UDS: Unified Diagnostic Service
- XCP: Calibration Protocol

1 Introduction

Software has become a key element in our modern industrialized world and its importance has grown exponentially during the last 50 years. It has reached the point in which it can be considered as one of the most important industries of our time as it is present in almost every aspect of our daily life.

The automotive world is not an exception of this, and it has been drastically reshaped in the last few years by these advances.

The electronics of a car consist on a network of different Electronic Control Units which are electronic components that have a microcontroller. Each ECU is capable of receiving and sending messages using the CAN (Controller Area Network) protocol to the rest of the network, where every single one has a specific purpose, such as the control of the engine or the lock of the doors.

The number of ECUs in an average car has grown from 2 or 3 to almost 100 during the last years and with them the engineering complexity of the whole system and hereby the number of lines of code.

The nature of the industry involves high standards of quality as any aspect of a vehicle has to take into account safety in a degree that is not seen in other industries. That generates the need for engineering processes to be heavily standardized, and software development is one of them.

This project was developed in IDNEO, a successful automotive business located in Barcelona out of the need to optimize the standardization process of the ECU programming. In this project the key objectives are:

- Introduce the current standardized process of the industry
- Go through the implementation of this process within the company
- Analyse the process and find its weaknesses
- Implement improvements to the process
- Analyse the changes performed to the process
- Set future improvements and ideas

2 Industry standards

The automotive industry has several standards that should be followed in order to deliver a new product to the market. These standards must be complied in order to meet the requirements of the industry as they are requested by the offers made by the clients and are regularly audited.

Those standards involve all aspects of the engineering process: Organizational aspects and technical details of the whole the system. From all the standards followed in the process we summarize here the ones that affect the part we are aiming to automate.

2.1 The engineering process: ISO/IEC 15504[2][3] and ISO/IEC 12207[4]

ISO 15504 defines the Software Process Improvement and Capability Determination (SPICE). It defines a set of documents and processes a product should follow to reach the market and sets the maturity degree of an organization. The process is based on the software lifecycle standard ISO 12207.

2.1.1 Capability determination

"Capability determination" sets the degree of "maturity" of the process within a company. It determines how well adapted to the SPICE process a company is. IDNEO is currently rated as "Level 3" which means that the process is implemented using a defined process that can achieve its outcomes. This rating allows us to market our products to almost all the big OEMs although it is true that improving it to level 4 or 5 would give us more credibility and access to specific projects we cannot reach with the current level.

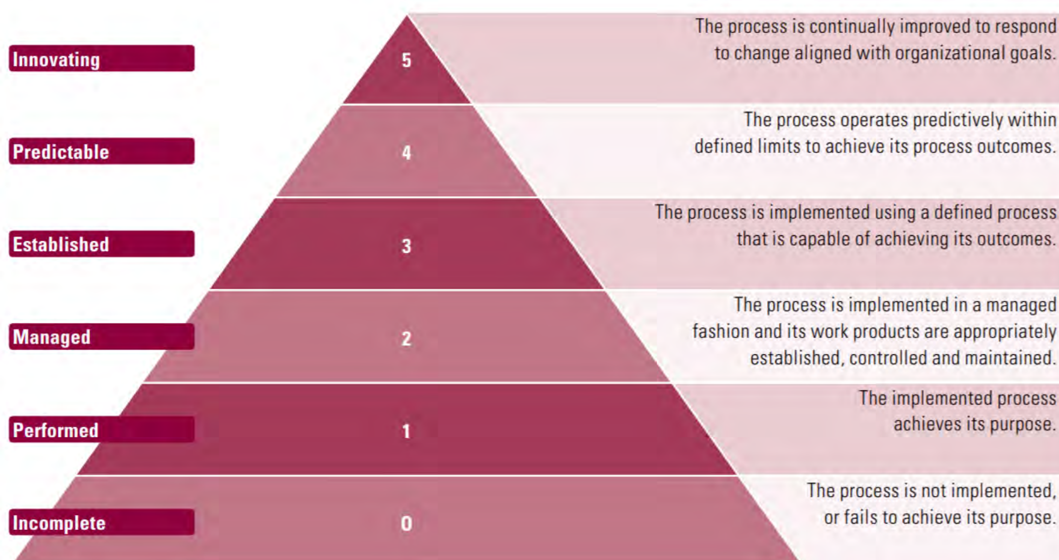


Figure 1: Capability determination pyramid that defined the Capability Levels defined by SPICE

2.1.2 The “V” model:

The V-model summarizes the main steps to be taken in conjunction with the corresponding deliverables within the project life cycle development. It describes the activities to be performed and the results that should be produced during product development.

The left side of the “V” represents the decomposition of requirements and creation of system specifications. The right side of the “V” represents the integration of parts and their validation.

SPICE applies to the development of mechatronic systems focusing on the software and system parts of the product. However, as of version 3.1 of the standard it is possible to add further engineering disciplines (e.g. hardware engineering, mechanical engineering etc.) and the corresponding domain-specific processes to the scope of Automotive SPICE, depending on the product to be developed.

The key idea of the V-cycle is that all the processes have their own atomic validation: each process of the V-cycle on the left is validated its counterpart on the right side. For example: The software architectural design (SWE.2) is validated by the software integration test (SWE.5).

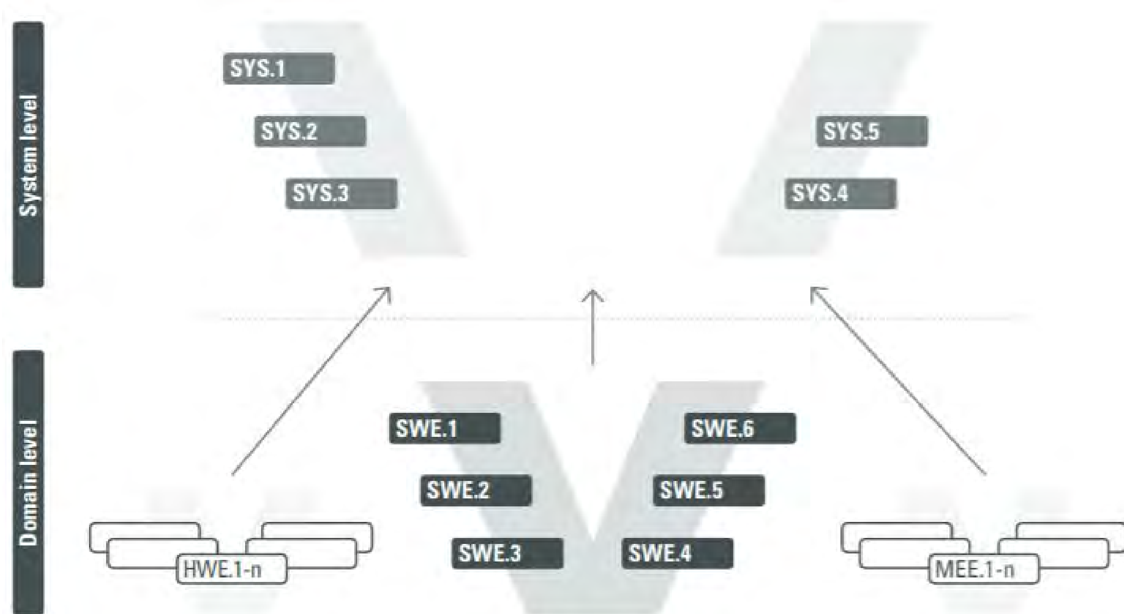


Figure 2 The V-Cycle defined by SPICE

Systems Engineering Process Group	
SYS.1	Requirements Elicitation
SYS.2	System Requirements Analysis
SYS.3	System Architectural Design
SYS.4	System Integration and Integration Test
SYS.5	System Qualification Test
Software Engineering Process Group	
SWE.1	Software Requirements Analysis
SWE.2	Software Architectural Design
SWE.3	Software Detailed Design and Unit Construction
SWE.4	Software Unit Verification
SWE.5	Software Integration and Integration Test
SWE.6	Software Qualification Test

Figure 3 Parts of the V-Cycle

I this project we will focus on the low part of the ”V”, which is the one that manages the software development. The studied processes go from part SWE.1 to part SWE.6. and also SYS.4 and SYS.5.

There should be 3 independent teams of engineers through this phase which are:

- **QA Team:** The Quality Team verifies that all the standards defined by SPICE are followed. This means that they must check that all the documents and evidences are up to date in each one of the parts of the process. That proves that the process is correctly followed. They should be strict, but they should also understand the specific singularities of each project.
  They also provide the documentation and they answer all the questions in the case of a SPICE audit.
- **Development Team:** The Development Team is in charge of designing the code, implementing it and performing all the software tests. It also maintains the software and if it is required it makes changes according to what is requested by the Validation Team.
- **Validation Team:** The Validation Team understands the specifications and verifies that each one of them is met. It performs a full validation which involves the full system, not only the software. In the case it finds an issue it is capable of determining which team should be addressed: software, hardware or mechanical. It then exposes the problem to the team that has to fix it and verifies that it has been solved afterwards.
  The Validation Team also manages the complains and questions of the client. It reproduces the problems found by the client and in the case of a real issue it addresses the right team.

The process is organized using milestones. A milestone is a stage of the process, it proves that a certain level of maturity has been reached. In order to prove that a milestone has been passed there is always a meeting between the QA Team and the Development and the Validation teams. In a milestone the QA Team ask for evidences to the other teams.

A milestone can have 3 different outcomes:

- **Passed**: When all the evidences are ok.
- **Not passed:** When the evidences are not OK and the milestone has to be repeated after correcting the issues.
- **Approved with comments:** Some evidences are not OK but the issues are not considered important. The process can continue until the next milestone but there are tasks that have to be performed before a specific due date.

There are 4 milestones, each one has its own deliverables and tasks that should be performed by a specific team:

2.1.2.1 Milestone 0

Team: Development

V-cycle parts: SYS.1

Deliverables:

- **Overall Project Overview**: PowerPoint presentation that summarizes the new specs that have to be covered by the new software loop and identifies the potential risks. It also sets the deadlines for all the other milestones.
- **Gantt**: Diagram that shows the fixed calendar for all the development activities

Description: Milestone 0 proves that the agreement with the client has been closed. That means that the scope is clear and the schedule is set.

2.1.2.2 Milestone 1

Team: Development

V-cycle parts: SYS.2, SYS.3, SWE.1, SWE.2, SWE.3

Deliverables:

- **Customer requirements:** Multiple documents that specify all the requirements of the project. They are all the information given by the client.
- **Doors:** Link to the Doors project where all the client requirements have been entered.
- **Customer acceptance:** Prove that the client has accepted the requirements freeze. This is normally a copy of a communication with the client like an email.
- **Software architecture review:** This document proves that the architecture has been approved by a certified reviewer.
- **Software architecture:** List of documents that define the software architecture

Description: Milestone 1 proves that the architecture and the design are clear, that they fit the specs and that they have been reviewed.

2.1.2.3 Milestone 2

Team: Development and Validation

V-cycle parts: SWE.4, SWE.5, SWE.6

Deliverables:

- **Release files:** Release of the project. That contains the binaries that are flashed into the board and all the complementary files such as CAN databases and Canape projects.
- **Release content:** Excel that shows the specs that have been implemented and the CARs that have been solved.
- **JIRA links:** Links to the JIRA project that proves that all the open CARs have been closed.
- **Software code review:** Document that proves that all the quality process has been followed and passed. This document has links to the evidences of the following tests.
  - **Runtime analysis**
  - **Static analysis**
  - **Unitary tests**
  - **Memory map**
  - **Task profiling**
  - **Compiler warnings**
- **Integration test evidence:** Document that proves that the Integration Test has been passed.
- **Validation scope:** Document that defines the tests that will be covered by the validation process and the schedule for those tests to be run.

Description: In milestone 2 the binaries and all the quality documentation are handed in by the Development Team and the tests that will be run by the Validation Team are specified.

Note: for further explanation about all the quality evidences those topics see chapter 2.2.

2.1.2.4 Milestone 3

Team: Validation

V-cycle parts: SYS.4, SYS.5

Deliverables:

- **DVP results:** Document that shows all the tests and evidences that they have been passed.
- **Test evidences:** Evidence of each one of the tests passed. They are different documents such as PowerPoints with screenshots, oscilloscope captures or communication logs.
- **Link to Doors:** Link to the test run in Doors which proves that the tests are documented in Doors.
- **Link to JIRA:** Link to JIRA which proves that all the CARs that were closed in MS2 are now validated.

Description: Milestone 3 proves that everything is validated. Between milestone 2 and milestone 3 the validation team performs the validation that has been planned before. If everything works correctly Milestone 3 is passed but if something is found during the validation

process another release candidate should be started. That involves starting the process again from Milestone 0.

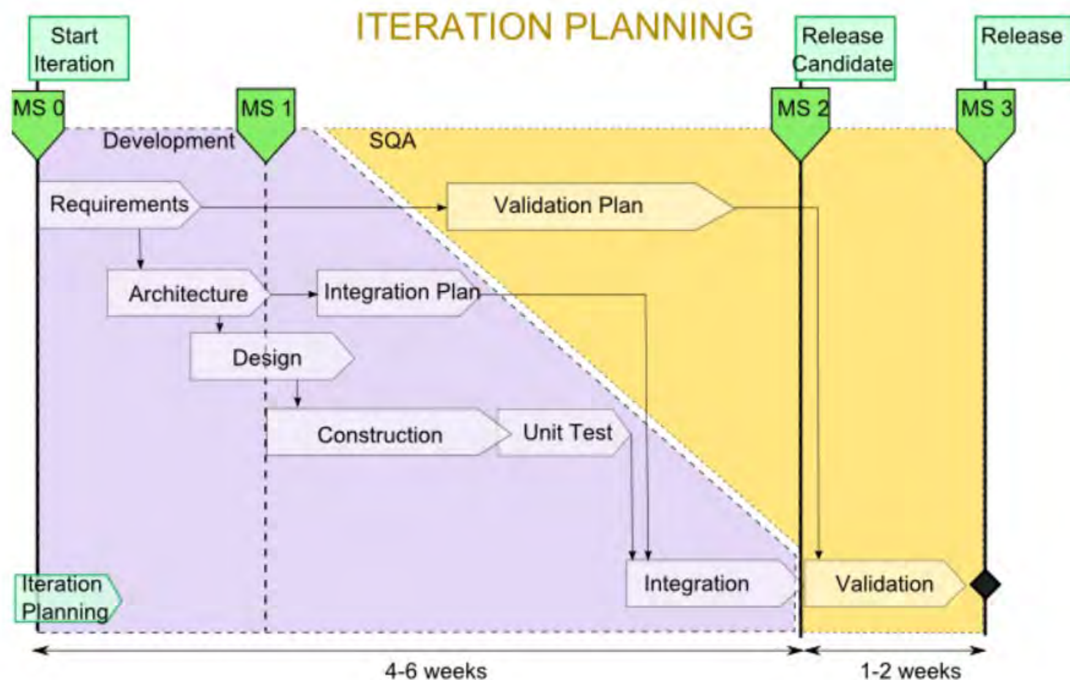


Figure 4 Software loop cycle

2.2 Quality standards

The software written for the automotive industry within the SPICE process needs to follow specific standards of quality. A group of tests should be passed. They involve multiple parts of the coding process. As explained before those tests are passed between MS1 and MS2 and the evidences are handed in MS2.

2.2.1 Static analysis: MISRA C Rules

MISRA C is a set of software development guidelines for the C programming language developed by MISRA. Its aims are to facilitate code safety, security, portability and reliability in the context of embedded systems. The rules consider writing aspects of the code. The analysis of the code is made without compilation as MISRA rules are about the written code itself.

Below you can find two examples of of MISRA rules:

- **Cyclomatic complexity rule:** The maximum amount of cases in a switch-case statement cannot be higher than 30.
- **MISRA rule number 10.1**: No implicit casts are allowed, they all have to be explicit, for example:

```
- UI_16 a;
```

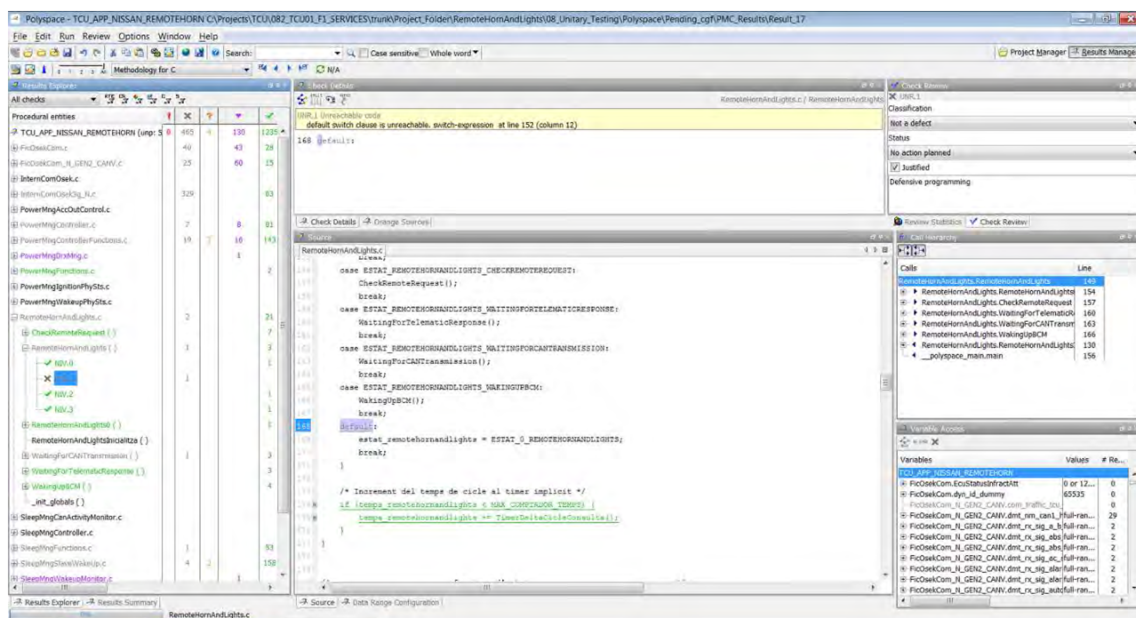
```
-    UI_8 b = 3;
-    a = b; //this is not allowed as there is an implicit cast
-    a = (UI_16) b; //this is allowed
```

MISRA rules are automatically checked using Polyspace Bug Finder. This is a software provided by Mathworks which is the standard in the industry. The software analyses all the written code and it shows the results of the analysis using a graphical interface. The developer can then correct the rule violation or justify it.

2.2.2 Runtime analysis

Runtime analysis analyses the errors occurred during the execution of the program. They are important to detect, as they may cause critical safety and security concerns. Software run-time errors include logic errors such as arithmetic exceptions. They can also include control and data flow related defects such as non-initialized variables or pointers, memory related defects such as buffer overflows, or concurrency defects such as race conditions.

Runtime analysis is performed using the Polyspace Code Prover which uses an interface similar to the mentioned Polyspace Bug Finder



2.2.3 Unitary tests

Unitary tests test the most atomic software component which is the function. The used program compiles with its own compiler each function of the code and test that their behaviour is the expected one. For each function all the functionalities should be covered. That means that the whole function should be tested.

Tests are written by the development engineer at the same time he is writing the code. By doing so the engineer is reviewing his own work so errors are less likely to exist.

Two key concepts of the Unitary Tests are line coverage and branch coverage.

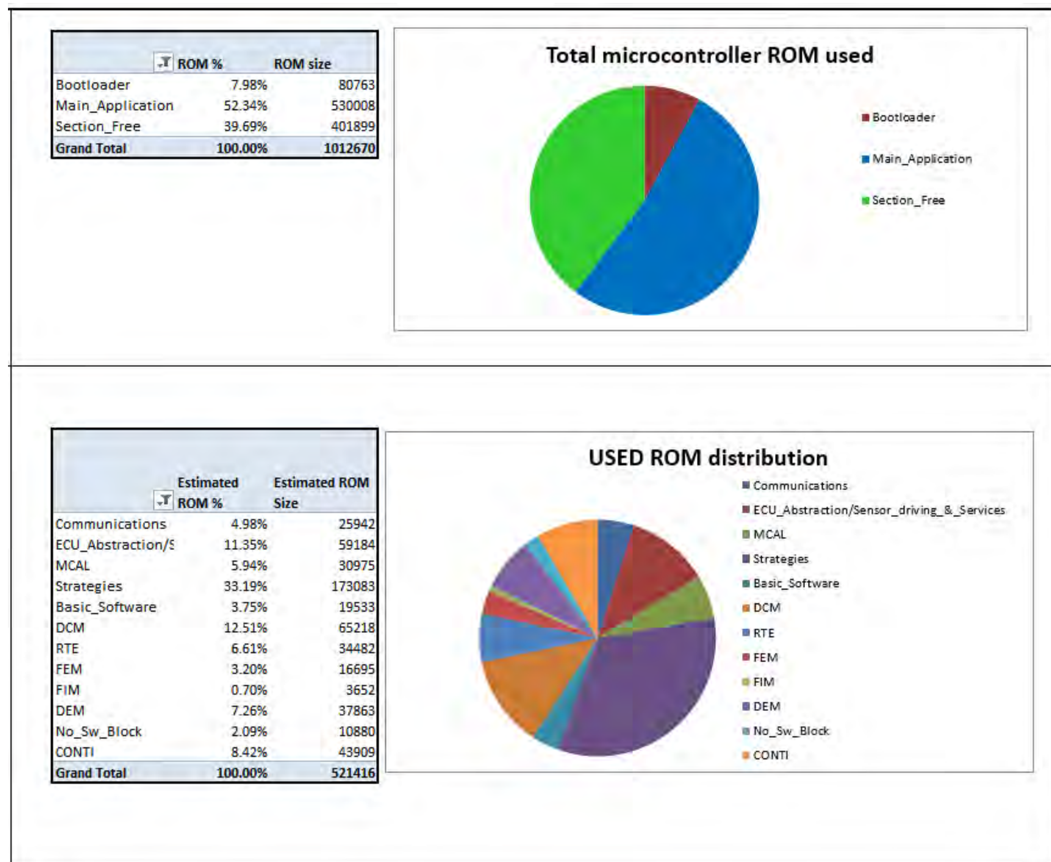
Line coverage represents the percentage of lines that have been used. For example, if we have a function with an if-else statement we will need two test cases to have a 100% line coverage.

Branch coverage represents the percentage of the branches that have been covered. A 100% branch coverage represents that all the possible paths of the code have been covered. For example, a condition with an *or* (like A || B) needs 3 test cases to reach 100% line coverage:

1. A == TRUE, B == FALSE
2. A == FALSE, B == TRUE
3. A == TRUE, B == TRUE

2.2.4 Memory map

The memory map relates the memory locations and the software components. It specifies how much memory each software component is using. It does so for both the RAM and ROM memories.



	ROM %	ROM size
Bootloader	7.98%	80763
Main_Application	52.34%	530008
Section_Free	39.69%	401899
Grand Total	100.00%	1012670

	Estimated ROM %	Estimated ROM Size
Communications	4.98%	25942
ECU_Abstraction/S	11.35%	59184
MCAL	5.94%	30975
Strategies	33.19%	173083
Basic_Software	3.75%	19533
DCM	12.51%	65218
RTE	6.61%	34482
FEM	3.20%	16695
FIM	0.70%	3652
DEM	7.26%	37863
No_Sw_Block	2.09%	10880
CONTI	8.42%	43909
Grand Total	100.00%	521416

2.2.5 Task profiling

The task profiling says how much time within the main code loop each process is using and what is the total time used.

In real time systems there is always a main loop that is running permanently. Within that loop all the periodic tasks are run. There is normally a time specification that has to be fulfilled.

2.2.6 Compiler warnings

In the case warnings appear when compiling and if they cannot be avoided, they should be justified. That is why there is a document that specifies a list of warnings present in the final code and a plausible justification for each one of them.

3 Used software tools

In order to understand the automation, it is necessary to explain the tools used for it. In this chapter we go through each used tool.

Any tool used in the process has to be qualified. That means that the tool has been analysed and tested by someone qualified within the company and has been certified as a qualified tool. In some cases, such as 3rd party tools like compilers it is also needed that the tool is qualified as automotive compliant. That means that it is certified externally, and it is sold as an automotive tool. The same thing happens not only with the software but also with the hardware used: components such as microcontrollers have to be automotive compliant too.

3.1 Rational Doors

Rational Doors is a requirements management tool that captures, traces, analyses, and manages changes in information. In a SPICE process it is required to have all the information traced. Rational Doors is the chosen program for doing so.

All the requirements are entered to Doors and each one of them is traced to a test. For each loop of the development the key tests are performed and the results are also updated to Doors.

This allows to have all the information about the whole process at any given time so it is possible to answer questions about issues quickly and trace problems when they arise.

In the figure below you can see Doors with a group of requirements already entered.

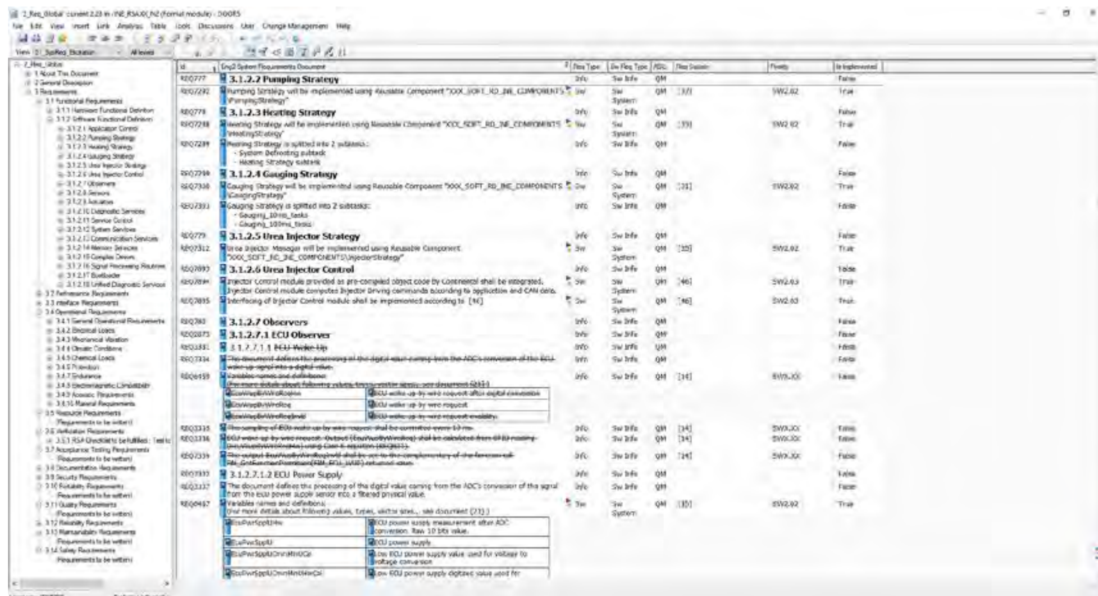


Figure 5: Doors screenshot

3.2 Vector Canape

Vector's CANape is a multifaceted tool that is available for ECU development, calibration, and diagnostics as well as for measurement data acquisition. The tool has all the functionalities that allow any automotive project to be connected to the PC. It is the chosen tool for debugging and testing the projects in IDNEO and it is the standard program used in the industry.

The specs that we use are:

- **Communications**:
  - **CAN**: Canape can process a CAN library and receive and send CAN messages.
  - **UDS**: Canape is able to receive diagnostic services through CAN.
  - **XCP**: XCP allows to see any variable of the code in real time while the program is running. It is used for debugging purposes.
- **Data storage**: Canape can store data of any of the communications mentioned above and it is able to open data stored files from the client (MDF format). This is used mostly for reproducing issues and communicating with the client.
- **Calibration:** Canape is used for calibrating the software which consists on writing variables in a specific memory zone.

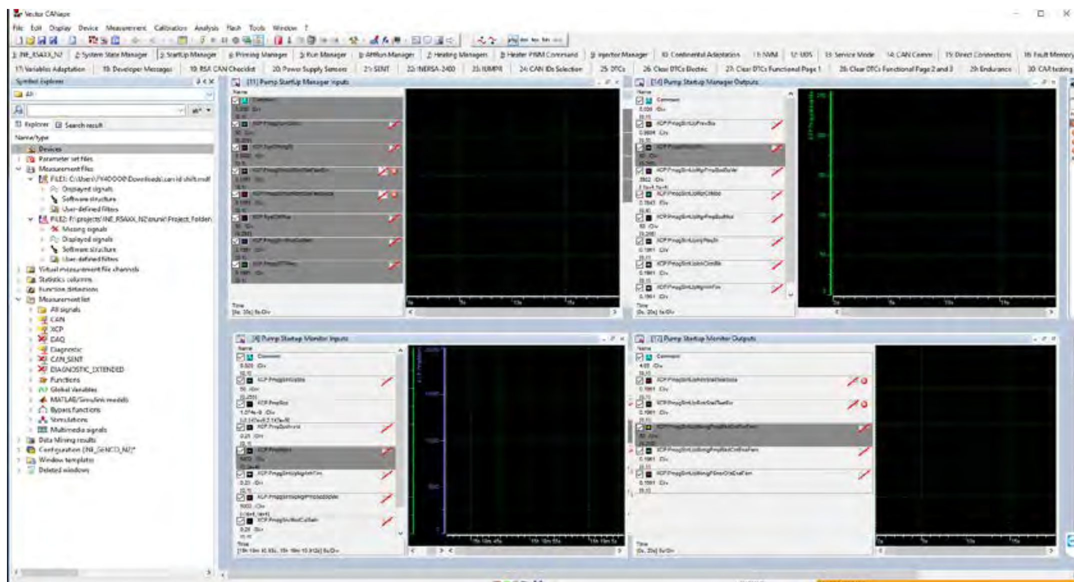


Figure 6: Canape screenshot

3.3 Vector Canoe

Vector Canoe is a tool similar to Canape. It is also able to connect a PC with the automotive hardware under development and it also supports CAN, UDS and XCP.

However, on top of this, it has much more complex features such as being able to automate any process and performing automated tests on the real hardware. It is also able to simulate an entire CAN network and it is able to perform tests mixing real components connected to the PC with simulated components programmed within the computer.

Tests can be written in both CAPL or C# and it has an easy interface that allows the test to be configured by any developer without a strong scripting knowledge.

It is also interesting because it can be controlled through a COM port by an external application. That opens the possibly of automating the tests externally using tools like Jenkins.

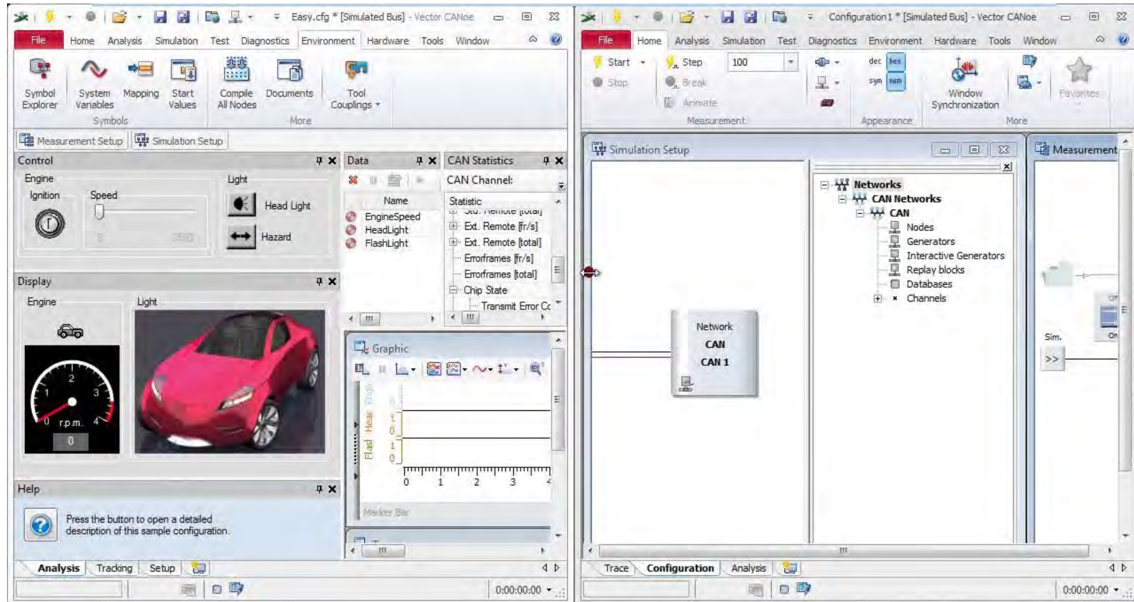


Figure 7: Canoe screenshot

3.4 Jenkins

Jenkins is an automation server which helps to automate the non-human part of the software development process, with continuous integration and facilitating technical aspects of continuous delivery. It supports version control tools, including Git or SVN as well as arbitrary shell scripts and Windows batch commands.

Jenkins is used in our SW group to automate some of the tasks. Our configuration consists on a Jenkins Master which receives all the petitions from all the Developers and a Jenkins slave which perform the tasks commanded by the master.

3.5 Python scripts

Python is an interpreted, high-level, general-purpose programming language. Its popularity, big community and easy usage makes it an ideal scripting language for managing multiple parts of the process.

3.6 Excel Macros

There are multiple parts of the documentation that end up in an excel file as a result. Although excel macros can be useful, they are substituted by python scripts in a lot of cases. This is because Python is easier, free and is does not have issues with different versions. However, there are still parts of the process that use excel Macros.

3.7 SVN

SVN is a software versioning and revision control system that is used in IDNEO and it is widely used in the industry. It allows to control and maintain the code by providing a change history and is able to trace back any change made. It is also able to manage reusable code which is maintained separately and used in multiple projects and it also provides the possibility of making branches and tags.

The main difference with GIT which is the other widely extended tool for software versioning is that GIT manages the repository in a non-centralised way. With git it is possible to perform a "push" without a "commit". The "push" concept is non-existent in SVN as the repository is always exclusively in the server. It is necessary to be connected to the server in order to perform a "commit".

3.8 Unity

Unity is a framework that allows unitary testing in C. It is a very light platform that is able to easily allow the user to perform a unitary test with a few scripting commands. Unity is able to generate mocks of called functions and to show a test report once it has finished. However, it has some weak points such as limitations when working with pointers or the need for extra scripting if multiple tests are needed.

```
35     *      Range of values:
36     *       * Maximum: 4096
37     *       * Minimum: -4096
38     *         [OUT] rty_motor_curr_stat: Reliability of the measured motor current consumption (based in a plausability test)
39     *                                    Valid values according t_curr_stat type.
40     *        [OUT] rty_bsw_err: Basic software error in case of problems during the process of the requested API function.
41     *                            Valid values according t_bsw_err type.
42     */
43  7 void BLDCMotorCurrentConsumption(uint8 rtu_motor_side, uint16
44      *rty_motor_curr, sint8 *rty_motor_curr_stat, uint8 *rty_bsw_err)
45    {
46      /* MultiPortSwitch: '<S2>/motor_curr_selcted' incorporates:
47       *  SignalConversion: '<S2>/TmpLatchAti_l_motor_currOutport1'
48       *  SignalConversion: '<S2>/TmpLatchAti_r_motor_currOutport1'
49       *  SignalConversion: '<S2>/TmpSignal ConversionAtmotor_sideOutport1'
50       */
51  7   if (rtu_motor_side == 1) {
52  1     *rty_motor_curr = l_total_meas_current;
53
54        /* MultiPortSwitch: '<S2>/motor_curr_stat_selcted' incorporates:
55         *  SignalConversion: '<S2>/TmpLatchAti_l_motor_currOutport1'
56         *  SignalConversion: '<S2>/TmpLatchAti_l_motor_curr_statOutport1'
57         */
58  1     *rty_motor_curr_stat = i_l_motor_curr_stat;
59      } else {
60  6     *rty_motor_curr = r_total_meas_current;
61
62        /* MultiPortSwitch: '<S2>/motor_curr_stat_selcted' incorporates:
63         *  SignalConversion: '<S2>/TmpLatchAti_r_motor_currOutport1'
64         *  SignalConversion: '<S2>/TmpLatchAti_r_motor_curr_statOutport1'
65         */
66  6     *rty_motor_curr_stat = i_r_motor_curr_stat;
67      }
68
```

Figure 8: Example of Unity report

3.9 Compilers

The compiler is the software that translates C language into a machine language which is flashed into the microcontroller. In automation it is needed to use automotive-compliant compilers. That kind of compiler is always bond to a license that is managed from a server.

Some projects use open source compilers such a GCC but they are normally used in early stages of the project when the product has not reached the market yet.

Some examples of automotive-compliant compilers are IAR or Greenhills.

3.10 EDI

EDI is an application that allows the definition of the software design graphically and makes the design of finite state machines easier. The output file is a XML document and using a XSLT transformation sheet it is possible to generate automatically the source code that implements the defined FSM.

EDI is a software developed within IDNEO internally and it is not commercially available. It is quick, easy to use, reliable and it is for free when used internally for IDNEO projects. However not all the clients agree to work with a non-validated software tool. That is why EDI is sometimes substituted by Matlab Simulink.

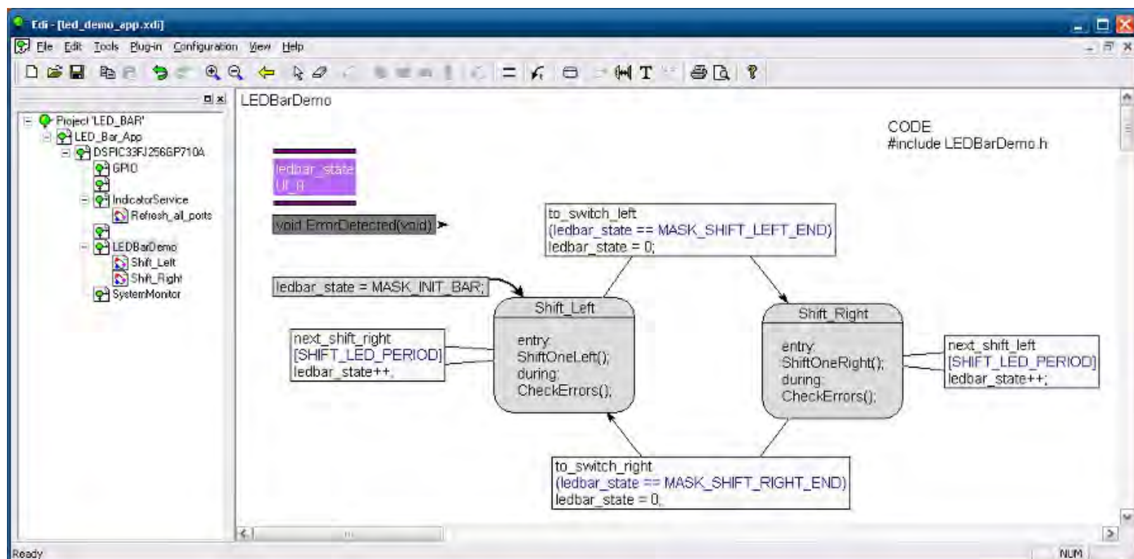


Figure 9: Example of finite state machine defined in EDI

3.11 Matlab Simulink

Simulink is a graphical programming environment for modelling, simulating and analysing multidomain dynamical systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It is specially useful for us because with Simulink it is possible to export the design to C language. This way we can have a very reliable way of implementing models and state machines.

Its principle is very similar to the EDI program but its capabilities are greater and as it is a de facto standard in the industry it makes it easy for us to work with when sharing a project with a 3rd party.

4 Initial setup

Before the performed modifications, the process was partially automated using some of the tools exposed in chapter 3. In this chapter we will explain the initial configuration of our setup and we will link it to the previously explained process. The idea is to show how the process was achieved technically. For doing so we will go through the architecture of the setup while explaining how it is used during the process.

4.1 Initial infrastructure

The net consists on a group of computers, all of them except the PC of the client are currently located in the IDNEO office itself.

We can differentiate between 3 types of computers:

- **Human users:** These are the personal computers of the different users of the system that need different resources from the servers and they are the actual inputs and outputs of the system. They are:
    - **The PC of the developer:** It is connected to all the needed resources to be able to perform all the activities of MS0, MS1 and MS2
    - **The PC of the validator:** It is connected to all the needed resources to be able to perform all the activities needed for MS3.
    - **The PC of the Client:** It is exclusively connected to the FTP to download the code and the documentation in MS3 and to the Jira service to manage the different bugs that can be found after the release.
- **Non-human users**: This is the **Jenkins slave**. It is considered a user because it performs automated actions that would be done by the validator or the developer in the case the process was not automated. It needs access to all the services and licenses the human users need. It performs all the activities of the development that can be automated. It is used in MS2 and MS3.
- **Service servers**: These are the servers that provide services to all the users. Those services can be programs that are run externally or files. They are:
    - **SVN Server**: It provides all the file management of all the users. It is used during the whole process as it contains all the information.
    - **Canoe Server**: It is accessed by the developer to perform Canoe tests in MS2.
- **License servers:** Software licenses are provided by servers that manage them. Licenses can be used by any computer connected to the server but there is a maximum number of computers that can be using each license at the same time. This is called floating license and it is very common in this industry. Those are the servers that provide floating licenses of proprietary software:
    - **Compiler license server:** It provides licenses of the compilers which are needed by the developer between MS2 and MS3. They also provide the license to the Jenkins so the compilation can be done by the Jenkins server in the case that is automated.
    - **Matlab Polyspace license server:** It provides Polyspace licenses to the developer and the Jenkins server. They use them between MS1 and MS2.

The following figure shows whole network of servers and PCs that has been explained above:

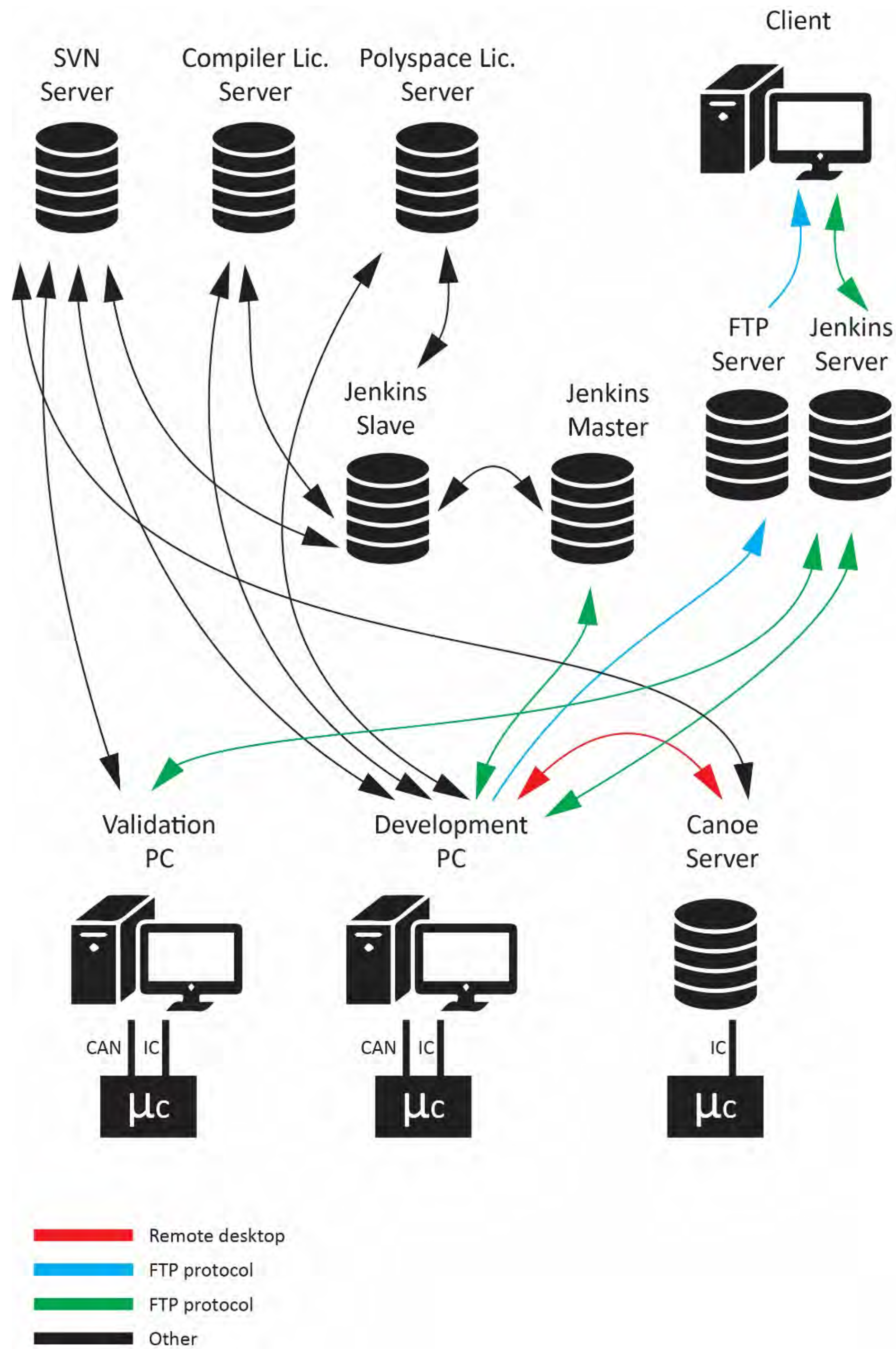


Figure 10: Initial setup configuration

Below we will go through each one of the computers and we will detail each one of its connections to the network:

4.1.1 Development PC

The PC of the developer had access to the following services:

- **SVN:** The developer must access the whole repository, the code and the documentation.

- **Compiler Licence Server:** The developer compiled in its own computer.
- **Polyspace Licence Server:** The developer was able to run Polyspace in its own computer in order to iterate quickly between code changes and Polyspace runs.
- **Jenkins Master:** The developer was able to launch Jenkins jobs and configure them. He did so through a Web interface.
- **FTP Server:** At the end of the process the Developer shared the binaries and all the documentation with the client via FTP.
- **Canoe Server:** Canoe was used to automate the ITP that is run by the Developer before MS2.
- **Jira:** The developer must be able to manage the issues found by the client and the validation team.
- **Microcontroller:** The developer needs the microcontroller to be connected directly to the computer to be able to communicate through the in-circuit connector for flashing and through CAN bus.

4.1.2 Validation PC

The PC of the validator had access to the SVN to be able to download the binaries and manage the documentation and the Jira to be able to manage the CARs with the client and the developer. The validation engineer had access to a Microcontroller to flash the code and test it using CAN and in-circuit connection.

4.1.3 Canoe Server

The Canoe Server had access to the SVN to be able to download the code. It was controlled by the Developer via remote desktop. It also had a Microcontroller and was able to perform automated tests although they would never involve flashing the board.

4.1.4 Jenkins Slave

The Jenkins Slave needs to have access to the same resources as the developer because it needs to automate some of the activities a developer does. It is connected to the Jenkins Server which is the one that manages its automation.

4.1.5 Client

The client was connected to the FTP and also to the Jenkins server. It used the FTP for downloading the binaries and the documentation and the Jenkins server for opening and managing CARSs.

4.2 Initial process

In this part we will go through all the stages of a software loop using the setup that has been described in 4.1.

Milestone 0 and milestone 1 were not automated because they only involve documentation that is filled in manually and handed in to the QA department. Activities such as the agreement with the client or the architecture design of the software cannot be automated.

However, in MS1 and MS2 we find a lot of work that can be automated. We try to explain it in the following chapters:

4.2.1 Milestone 2

Milestone 2 was partially automated. From all the documentation that had to be handed in to the QA team this is the stage of the automation:

File	Automatized	Automatable	Automatization description
Release files (binaries)	No	Partially	-
JIRA links	No	No	-
Software code review	No	Yes	-
Runtime analysis	Yes	Yes	Script that runs Polyspace Runtime (*1)
Static analysis	Yes	Yes	Script that runs Polyspace Static (*1)
Unitary tests	Yes	Yes	Script that runs Unity (*2)
Memory map	Yes	Yes	Script that analyses the binaries (*3)
Task profiling	Partially	Yes	C code modified for this purpose (*4)
Compiler warnings	No	Yes	-
ITP	Partially	Yes	Automated using Canoe [1] (*5)
Validation scope	No	No	-

The details about how the parts were automated can be found below:

4.2.1.1 Runtime analysis and Static Analysis (*1)

Polyspace can either be run via its own GUI or via command line. Thanks to the command line feature Polyspace can be automated.

Polypsace is finally called using a single command line that is run in a Jenkis job. However, the process has some specific features that have to be build using makefiles and Python scripts before calling Polyspace itself. Those are the features:

- **Importing old comments:** Once Polyspace is run the developer needs to justify or fix each violation found by the program. After entering all the new justifications and modifying the C code the user runs Polyspace again. Therefore, Polyspace needs to be able to import all the old comments and put them in the right place even if the code has changed.
- **Preparing the code:** Polypsace Runtime compiles the code with its own compiler and runs it in order to find runtime issues. For this reason, the code needs to be prepared for that compiler which is very different to the one used for the project.
  This is done using the #ifdef statements with the macro ANALYSIS_TOOL in lines such as the following one:

```
#ifdef ANALYSIS_TOOL
```

  That hides to Polyspace parts that would not compile, such as assembler code or some types of macros.
- **Starting points:** Polyspace Runtime analyses the code from the main using the main loop and changing the variables to test all the code paths. That works for most of the project, however, there are parts that are not directly called form the main loop such as interrupts. Interrupts need to be forced into the analysis using what is called "starting

points". Starting points are definitions of places of the code that Polyspace should call by itself. They are defined by the user.

Scripting architecture:
Polyspace is called by a Jenkins job that does the following:

1. Download all the needed parts of the project which are:
    - The code
    - The Polyspace results folder
    - The tools folder that contains the scripts
2. Make a copy of the Polyspace results folder and rename it to *Polyspace_backup*. It does this so the justifications are saved.
3. Delete the Polyspace results folder.
4. Run the main makefile with the command rte_cfg (for runtime analysis) or sta_cfg (for static analysis). That creates the configuration file that is needed to call the Polyspace. This file contains information about the Polyspace options such as:
    - Files to include
    - Polyspace run options
    - Macros to include (such as ANALYSIS_TOOL as explained)
5. It then runs the Polyspace Python script that has those inputs:
    - Code folder
    - Polyspace results folder
    - Old Polyspace folder
    - Options file

4.2.1.2 Unity (*2)

The Unity environment is prepared for performing different test cases of a single C file. Each file needs a new unity project with its own folder structure and test harness. The results are also shown in a single HTML file for each unity project.

On top of this, not all unitary tests come from the Unity tests for the project. Tests that should be handed in can also come from:

- **Matlab Simulink test harneses:** Test of part of the code that is generated by Simulink are not tested using unity but using the Matlab environment.
- **Unity test from reusables:** The reusable modules of the project have their own tests and they do not have to run every time the project changes, only when the reusables itself change.

It is needed that we manage to run all the tests all in once and to be able to gather all the results from the 3 different sources in one single file.

For doing so there is a group of scripts that perform that task:

Scripting architecture:

There is a Jenkins script that manages this task by performing the following tasks:

1. Download all the needed parts of the project which are:
    - The code (test folders included)
    - The unitary test results folder
    - The tools folder that contains the scripts
    - The reusables test folders with the reusable test results
2. Run the python script *test_report.py* which does the following:
    - Run the program cloc.exe that counts the lines of each single files of the code
    - Run all the unitary tests of the project using another script
    - Gather the information of the Polyspace unitary tests
    - Gather the information of the reusable unitary tests
    - Write all the test results information in an excel and calculates the resulting percentages of line coverage and branch coverage

1	File	Code lines	Line Cov	Branch	Branch	IsIDNE	IsProje	IsJustifie	Justification
148	..\..\..\..\ThirdParty\HAL_Spc574k\SentDriver_private.c	299	0.99	88	0.94	1	1	1	Some lines of the code are not reachable as they are defensive coding
282	Source\FicOsek\I2CFrameWrapper.c	93	0.96	22	0.86	1	1		Needs update
283	Source\FicOsek\SentFrameWrapper.c	365	1	71	0.85	1	1		Some lines of the code are not reachable as they are defensive coding
284	Source\FicOsek\SpiFrameWrapper.c	263	0.94	105	0.87	1	1		Needs update

Figure 11: Example of unitary test results

4.2.1.3 Memory map(*3)

Memory map is performed by running a script that processes all the information within the *.map* file. The *.map* file is generated when the code is compiled and has the information about the location and the size of each variable of the code. Below you can find an example of some lines of a *.map* file:

```
2423   Load Map Fri Apr 12 17:29:58 2019
2424   Global Symbols (sorted alphabetically)
2425
2426                    00000000+000092 _ACTION_RASL_GetRagInjCurBuf
2427                    00000000+00004e _ACTION_RASL_GetRagInjPulse
2428                    00000000+00012a _ACTION_RASL_GetRagInjVpRdu
2429                    00000000+00001a _ACTION_RASL_GetRagInjWinMesStat
2430    .text           0009d33c+00003c _ACTION_RASL_SetRagInjCtrl
2431    .text           0009d394+00001e _ACTION_RASL_SetRagInjCtrlConf
2432    .text           0009d378+00001c _ACTION_RASL_SetRagInjIntvCtrl
2433    .text           0009d13c+0001a4 _ACTION_RASL_SetRagInjIntvStart
2434    .text           0009d2e0+00005c _ACTION_RASL_SetRagInjIntvStop
2435    .text           0009cf84+0001b8 _ACTION_RASL_SetRagInjStart
2436    .text           0009d3b2+0000b2 _ACTION_RASL_SetRagInjWinMes
```

Figure 12: Example of a *.map* file

After processing all that information the script generates a text file which is used to populate an excel with all the results.

4.2.1.4 Task profiling (*4)

In order to perform a task profiling it is necessary to know the time spent in each task of the project. For doing so there is a simple trick that is used by using a GPIO pin that is not used for the project:

Every time the task that is being monitored starts the pin is raised, when the time finishes the pin is brought to 0 again. This way if we measure the length of the generated pulse with an oscilloscope, we are able to know how much time is needed for a specific task.

The code is written in a way that there is a variable reserved for receiving which task we want to monitor. This part of the code is only compiled when using a specific macro thus is not handed in to the client.

When we change the value of that variable via XCP the monitored task changes.

4.2.1.5 ITP (*5)

The ITP used to be done manually but it is now run using a Canoe script since the automation was performed last year (for more information about this process see [1]).

At the beginning of the project the Developer must write the ITP script that checks that the basic functionalities are implemented correctly. Once this is done every time an ITP test is performed (once every SW release) it is only needed to flash the board, connect it to the Canoe server and perform the test.

4.2.2 Milestone 3

Milestone 3 is the part of the process performed by the validation engineer. So far this is the whole picture of what has been automatized:

File	Automatized	Automatable	Automatization description
DVP results	No	No	-
Test evidences	Partially	Yes	Only endurance test performed
Link to Doors	No	No	-
Link to JIRA	No	No	-

The only part that can be automated is the tests itself and the evidences. This part takes a lot of hours, normally one or two weeks for every SW loop. That is why the automation of this is key to the process. The only test that was automated was the endurance test. The endurance test consists on a 6 to 12 hours test that performs standard operations in the board to check that the code can run during long periods of time.

This test was run using a Canape script in the PC of the validation engineer without using Canoe.

5 Improved setup

After analysing all the automatable parts that were not automated and all the weak points of the previous process, we came to a new improved architecture and process. This chapter explains the changes applied.

The following image shows the final architecture of the system, following the picture we explain each one of the changes.

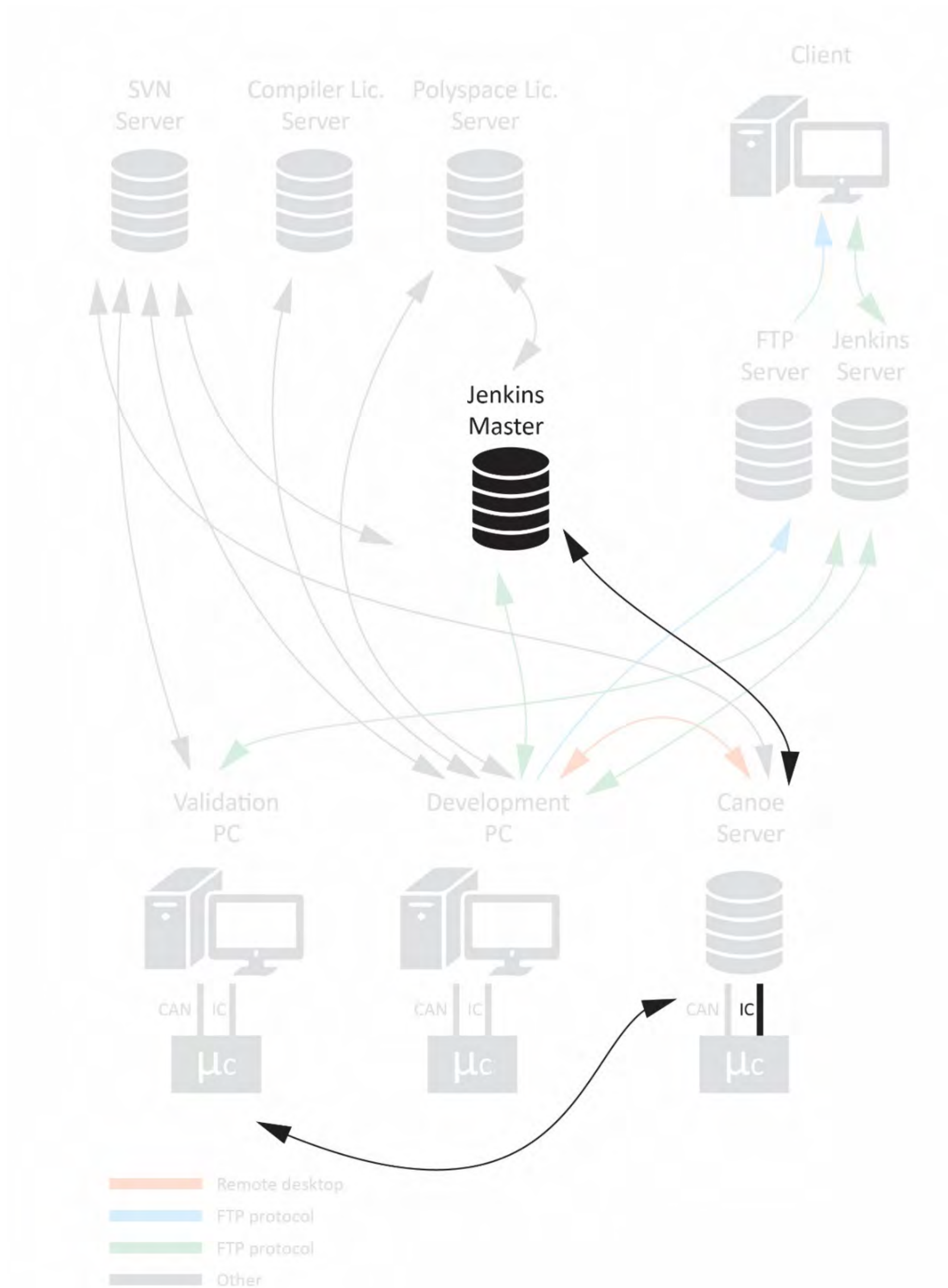


Figure 13: New setup after modifications

5.1 Compiler Warnings Script

There is a new script that processes the output text from a compilation. The script is able to find the warnings and store them in an excel file. Once they are stored, they can be justified by the user. In the case the script is rerun it is able to maintain the old justifications. This way the warning justifications can be treated the same way as Polyspace justifications.

5.2 ITP from Jenkins

The ITP can now be processed from Jenkins and the board can be flashed from the same Canoe PC or via CAN or IC. This means that the ITP itself is now a simple Jenkins job. That means that it can be automated and it can even be run periodically.

5.3 Canoe for validation

Before the modifications all the test run by validation (except the endurance) were performed manually. They are now all automated using Jenkins. This way the validation has shortened its length by more than a week in most of the cases.

That is also important for the robustness of our process because each test written in Doors has a script linked to it so it will be run under the same exact conditions the next time is needed.

5.4 SENT emulator

Most of our projects use the SENT protocol. SENT protocol is a standard two-wire protocol that is widely used to transfer information between sensors and microcontrollers.

So far, we had to connect the actual sensor to the board to perform the tests. That was messy, not automatable and not reliable. For this reason, we have developed an emulator of any sensor. The emulator is connected to the board itself via sent and to the PC via CAN. Using CAN messages the PC can configure the SENT messages that the emulator will send to the board.

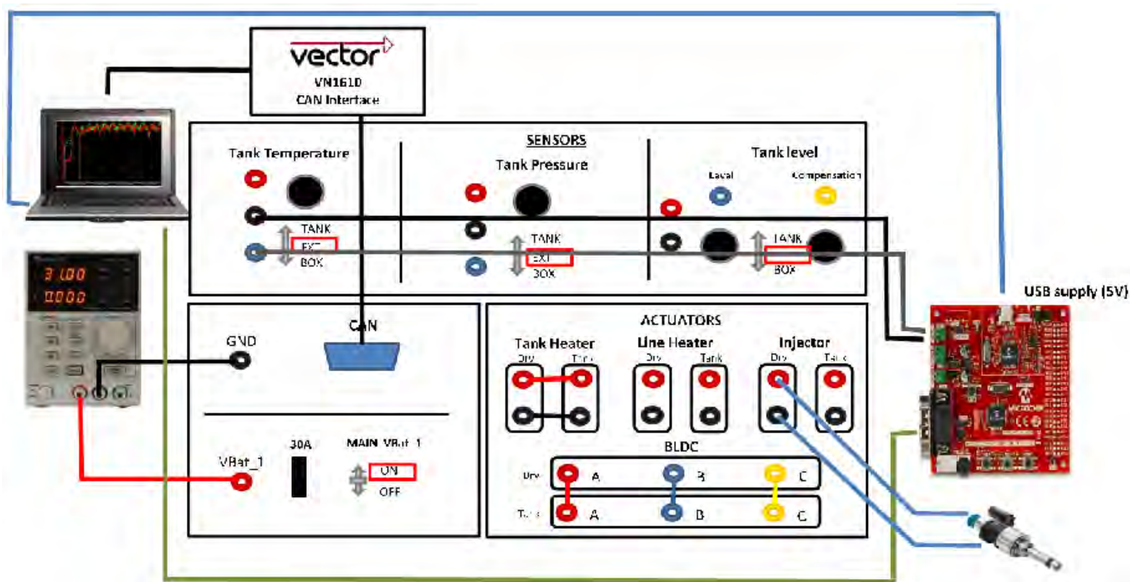


Figure 14: Sent emulator connected to the setup

5.5 SVN-Jenkins bug solved

Jenkins used to have a bug when working with SVN. That made anything done with Jenkins very unstable as the process of updating the repositories was interrupted very often.

After finding out the root cause we have managed to solve it.

This might seem trivial, but it was key for out continuous integration process as it all relies on Jenkins.

5.6 Jenkins server without slave

Having 2 servers, a slave and a master had sense in the previous office where we were more than 7 different teams of developers. There was one server and multiple slaves, one for each group. Now this has change and in our new office we have only one team therefore having a Jenkins master and a Jenkins slave has no sense anymore. For this reason and in order to simplify the architecture we have now one single Jenkins server that performs both tasks. We consider it to not to have an economic impact but it is a way of simplifying the setup.

6 Economic impact

In this chapter the changes made in the automation of the process will be analysed quantitatively. We will use the data that has been stored in the system in order to calculate the impact in number of hours of the improvements. This will be used to calculate the increment in productivity and the economic impact.

The data has been extracted from those sources:

- **Easyredmine**: It is a management tool that is used to manage the number of hours each project has used and the engineers that worked on each project.
- **Quality engine**: It is the tool used to upload all the information in each Milestone.
- **Jenkins:** As is the main automation tool it contains information about how many times each script has been run and the number of errors.

For performing the analysis, an example project has been chosen: *INERGY_RSA*. This is a big project within the company and has been chosen because it has all the standard elements of a project in our SW group and therefore the conclusions can be easily extrapolated to any other project. It is also interesting because it is an old project that has been in the company for 4 years now so we have a big amount of data.

INERGY_RSA is a system that consists on an injector and a urea tank. The system is the responsible of injecting urea in the exhaust gases of the vehicle, so the pollution and CO_2 are reduced. It is the system present in most of the diesel vehicles that can be found nowadays. It is what is commercially called within the VW group as *AdBlue*.

Those are some of the numbers of this project for the year 2018:

	All year	Average
Number of SW loops	7	1
Number of hours	1743	249
Development hours	1308	186,86
Validation hours	435	62,14
Number of bugs opened by the client	21	3
Number of Validation test perfomed	84	12

6.1 Changes performed

In the following points we analyse each one of the changes individually and we will check how each one of the affects the spent amount of time. All these calculations take into account 12 months: from January 2018 until December 2018.

6.1.1 Compiler Warnings Script

This is run once every release. Before the script this part of the process lasted 3 hours and it can be now done in 30 minutes. This means that in each release we save 2.5 hours.

Therefore, during 2018 this saved **17,5 saved hours**.

6.1.2 ITP from Jenkins

Before this change each ITP run took 2 hours because all the setup should be mounted and after the PC should be accessed remotely and Jenkins launched. This is now reduced to 15 minutes if the project is already connected to the Canoe server which is normally the case.

This represents 1,75 hours every time ITP is run.

ITP is run officially once every release. However during the development phase the developer runs it several times to perform a general validation during the implementation. We can approximate this to 10 times every loop.

This give us the following number of hours saved:

1,75 hours * 7 loops *11 runs = **134,75 saved hours**

6.1.3 Canoe for validation and SENT emulator

For this calculation we do not take into account the number of hours spent writing each test. As we suppose that all the tests have been already scripted.

Test time may vary a lot but taking into account the number of validation hours and the number of tests we can calculate an average:

435hours / 84 tests = 5,17 hours/test

This has been reduced drastically because each test now does not last more than 15 minutes. However we will assume a conservative number of 30 minutes/test.

This makes a numer of saved hours/test:

5,17-0,5 = 4,68 saved hours/test

If we take into account the full year:

4,68 saved hours/test * 84 tests = **393,12 hours**

6.1.4 SVN-Jenkins bug solved

For calculating the number of hours saved by solving this bug we should know how many times this bug happened.

Sadly we do not have access anymore to all the outputs of all the Jenkins scripts of the whole year as they get deleted after a specific number of runs.

However, we can access to all the outputs of the last SW loop so we can extrapolate it to the whole year. The number of times the bug happened during the loop was 34.

If we assume that every time the engineer has to fix it manually it takes 15 minutes we can calculate:

34 errors/SW loop * 7 loops * 0,25 hours = **59,5 hours**

6.2 Economic results

After the addition of the previous results we get the following numbers, those are the numbers of saved hours in total:

Change	Hours
Compiler Warnings Script	17.5
ITP from Jenkins	134.8
Canoe for validation and SENT emulator	393.1
SVN-Jenkins bug solved	59.5
Total	**604.9**

6.2.1 Reduction of spending

If we make the following assumptions:

- The average salary for a firmware engineer in Barcelona is 38.294€/year [2] [3]
- The engineer works 1764 hours/year [3]
- The social security costs are a 33% of the salary of the worker

It is possible to calculate this cost of an engineering hour:

(38.294 €/year * 1.33) / 1764 hours = **28,87 €/hour**

The resulting savings made in this project in the year 2018 would be:

604,9 saved hours * 28,87€/hour = **17.463 € saved**

6.2.2 Productivity

To calculate the percentage of engineering hours saved we perform this calculation:

604.9 saved hours / 1743 total hours = **34,7 %**

This result implies an increase in the productivity of a **153 %**

7 Future improvements

In this chapter we summarize the improvements that can be done in the future in order to obtain a better system.

7.1 Task profiling

Although task profiling is partially automated there is still a human task that could be automated. When measuring the times of each process the developer should perform the following task:

1. Set XCP variable
2. Connect the pin to the oscilloscope
3. Read the step and measure it
4. Write down the measurement

This is performed during roughly 40 measurements.
This task could be automated by a program that changes the XCP variable and afterwards measures the pulse using a signal analyser connected to the PC and that would be an interesting project to do.

7.2 Jenkins using scripts

So far Jenkins is configured by using a web interface in which the SVN repositories are selected and the scripts itself are written.

Each time a new project is configured everything needs to be done again in another Jenkins project. This is time consuming and prone to error.

That is why it is recommended that when such a degree of complexity is reached it is better to stop using the web interface and use a pipeline project instead.

Pipeline projects in Jenkins allow to store each Jenkins job as a piece of code. This way each script can be a configurable reusable and will have all the features of the code itself such as reproducibility and version control.

7.3 Git

After the new moves within the automotive industry and the current increment of projects that use linux or linux embedded more projects are starting to use GIT instead of SVN.

In order to use the same tool for all the projects it would be interesting to study the possible use of GIT as a version control system.

8 Conclusions

After all the work done there are a few ideas that can be shared about the process which can be summarized in the following points:

- **The use of automation increases the productivity: the changes made suppose a 153% of productivity increase**
  The number of hours spent automating the software is always worthy. Each hour spent writing a script, benchmarking a new tool or learning about the process generates much more money for the business than an engineering hour spent coding directly for a project charged to a client.

- **Good expensive software is worthy**
  It is very common that from the management point of view some software is seen as expensive. However most of the times this statement is based on views that do not take into account the whole picture.
  To put an example: A Canoe license costs roughly 20.000€/year. That can be seen as an expensive software but if the hours of Engineering saved thanks to Canoe are taken into account what is expensive is not buying it.

- **There is no magic tool**
  The process of making a process fully automated is not something that can be done easily. Although there are very good tools in the market it is important to understand the process deeply and to choose the tools wisely. There is no tool that can do everything, and it is important to know them well in order to fine-tune them. The knowledge of the tools and the software is what makes automation possible.
  Once that is fully understood automation should be introduced smoothly so the team can fully go through the learning curve without causing issues to the current projects.

- **Continuous integration is not yet fulfilled**
  For having a continuous integration all the process of every release should be performed automatically and tests should be performed daily or weekly.
  We cannot say yet that the current process can be considered as continuous integration, however, we can say that we are much closer to it.

9 References

[1] *Advanced Automated Integration Testing for Automotive Framework* by Eric Marín - (UPC)

[2] ISO/IEC 15504 https://www.iso.org

[3] *Automotive SPICE pocked guide* by Kugler Maag Cie - https://www.kuglermaag.de/fileadmin/05_CONTENT_PDF/2-10_automotive-spice_version_3_pocketguide.pdf

[4] ISO/IEC 12207 https://www.iso.org

[5] Glassdoor – https://www.glassdoor.com/Salaries/barcelona-firmware-engineer-salary-SRCH_IL.0,9_IM1015_KO10,27.htm

[6] Linkedin – www.linkedin.com note: paid subscription to get salaries information