Lappeenranta University of Technology
Faculty of Technology
Degree Program in Electrical Engineering

MASTER'S THESIS

Author:
Juho MATIKAINEN

Development of a test framework for power converter software releases: case Visedo PowerMASTER™ series

1st examiner: Prof. D.Sc. Pertti SILVENTOINEN
2nd examiner: M.Sc. Tommi KANKAANRANTA

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology
Degree Program in Electrical Engineering

Juho Matikainen

Development of a test framework for power converter software releases: case Visedo PowerMASTER^{TM} series
2016

Master's Thesis

44 pages, 26 figures, 1 table, 5 appendices

Examiners: Prof. D.Sc. Pertti Silventoinen
M.Sc. Tommi Kankaanranta

Keywords: software testing, regression testing, integration testing, test framework, inverters, J1939 protocol

Visedo Ltd. develops electrical drive systems for use in mobile work machines, buses and marine vessels. This includes different power converters, such as inverters and DC/DC converters. These converters need embedded software for different features, e.g. motor control and communication protocols. Software related faults discovered in these devices after their commissioning can result in costly warranty returns. Therefore the testing of the embedded software becomes an especially important part of the product life cycle. The goal of the thesis was to create a test framework for executing regression tests for the different software release variants of Visedo's power converters. Important criteria for the framework were modularity, reproducibility of the tests and the generation of detailed test reports. This thesis describes one way to construct a test framework and presents the different software tools used in building the framework. Two test suite scripts were written in the Ruby programming language for running in the framework and are presented in the thesis. These test suites test the J1939 communication protocol of the PowerMASTER^{TM} inverter, chosen as a case device for the thesis. The frameworks impact on Visedo's software testing process and future improvements for the framework are also discussed.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknillinen tiedekunta
Sähkötekniikan koulutusohjelma

Juho Matikainen

Testikehyksen kehittäminen tehonmuokkainten ohjelmistojulkaisuille: case Visedo PowerMASTER™ sarja
2016

Diplomityö

44 sivua, 26 kuvaa, 1 taulukko, 5 liitettä

Tarkastajat: Prof. TkT. Pertti Silventoinen
DI Tommi Kankaanranta

Hakusanat: ohjelmistotestaus, regressiotestaus, integraatiotestaus, testikehys, invertterit, J1939 protokolla

Visedo Oy kehittää sähkökäyttöjärjestelmiä työkoneisiin, linja-autoihin ja merialuksiin. Näihin kuuluvat erilaiset tehonmuokkaimet, kuten invertterit ja DC/DC-hakkurit. Tehonmuokkaimet vaativat sulautetun ohjelmiston erilaisten ominaisuuksien hallintaan, kuten sähkömoottorin säätöön ja kommunikaatioprotokolliin. Ohjelmistoon liittyvät viat, jotka esiintyvät laitteen toimituksen jälkeen, voivat aiheuttaa merkittävän kalliita takuupalautuksia. Tästä johtuen ohjelmistojen testaus muodostuu merkittäväksi osaksi laitteen elinkaarta. Tämän diplomityön tavoitteena oli luoda testikehys Visedo:n tehonmuokkainten ohjelmistojulkaisujen regressiotesteille. Tärkeitä kriteerejä kehykselle olivat modulaarisuus, testien toistettavuus ja testiraporttien luonti. Työ kuvaa yhden mahdollisen tavan luoda testikehys ja esittää kehyksen rakennuksessa käytetyt eri ohjelmistotyökalut. Kaksi testiskriptiä kirjoitettiin Ruby ohjelmointikielellä suoritettavaksi testikehyksessä ja ne on esitetty työssä. Skriptit testaavat Visedo:n PowerMASTER™ invertterin J1939 kommunikaatioprotokollaa. Työssä käsitellään myös testikehyksen vaikutusta Visedon ohjelmistotestausprosessiin ja esitetään jatkokehitysideoita testikehykselle.

Contents

Abbreviations and symbols

AC	Alternating Current
BEV	Battery Electric Vehicle
BAM	Broadcast Announce Message
CAN	Control Area Network
CANopen	A communication protocol for embedded systems, used especially in industrial applications
CAN 2.0B	Part B for the extended CAN format with a 29-bit identifier
CI	Continuous Integration
CSI	Current Source Inverter
CSV	Comma Separated Values
CTS	Clear To Send
DC	Direct Current
DSP	Digital Signal Processor
DT	Data Transfer
ECU	Electronic Control Unit
FPGA	Field Programmable Gate Array
FW	Firmware
GIT	A version control system, created originally for the development of the Linux kernel
IEA	International Energy Agency
IEC	International Electrotechnical Commission
IGBT	Insulated-Gate Bipolar Transistor
ISO	International Organization for Standardization
J1939	A higher layer protocol of the CAN communication protocol developed specifically for vehicle applications
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
OSI model	Open Systems Interconnection communication model
PCB	Printed Circuit Board
PDU	Protocol Data Unit
PG	Parameter Group
PGN	Parameter Group Number
PHEV	Plug-in Hybrid Electric Vehicle
RTS	Request To Send
SAE	Society of Automotive Engineers
SPN	Suspect Parameter Number
ST	Structured Text
TDD	Test Driven Development
TP	Transport Protocol
USB	Universal Serial Bus
VSI	Voltage Source Inverter
XML	Extensible Markup Language

kbit	Kilobit
Ω	Ohm
s	Second
V	Volt

1 INTRODUCTION

1.1 Background

An electrical drive is a system that converts electrical power, voltage and current, to mechanical power, torque and speed. Modern electrical drive systems also incorporate embedded digital control units that control the power converters of the drive. The control unit also needs embedded software to achieve the wanted functionality. Figure 1 shows the basic components and signals of an electrical drive system.

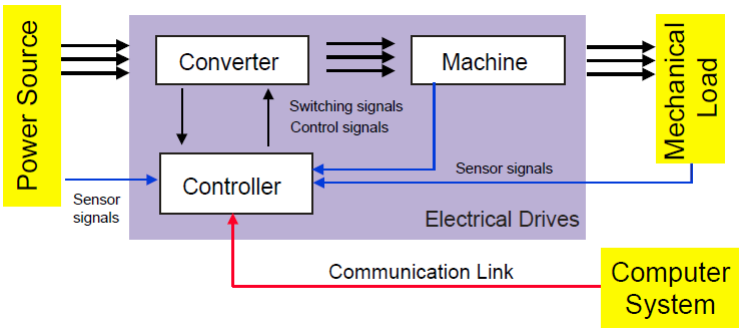


Figure 1: A block diagram describing the basic components of an electrical drive system [1]. The power converter drives an electrical machine connected to mechanical load. The power converter is controlled by the embedded control unit. A computer system is used to communicate with the control unit through a communication link.

The electrical drive business is growing constantly because of factory automation and the increasing number of hybrid and fully electrical work machines, cars and marine vessels. Electrical drives offer considerable savings in energy consumption and are one way to tackle the tightened CO_2 emission regulations and the growing need for more energy efficient systems. Figure 2 shows IEA's (International Energy Agency) report of global sales of plug-in hybrid (PHEV) and battery electric vehicles (BEV) from the start of 2010 to the end of 2014. An annual increase in sales can be seen from the graph.

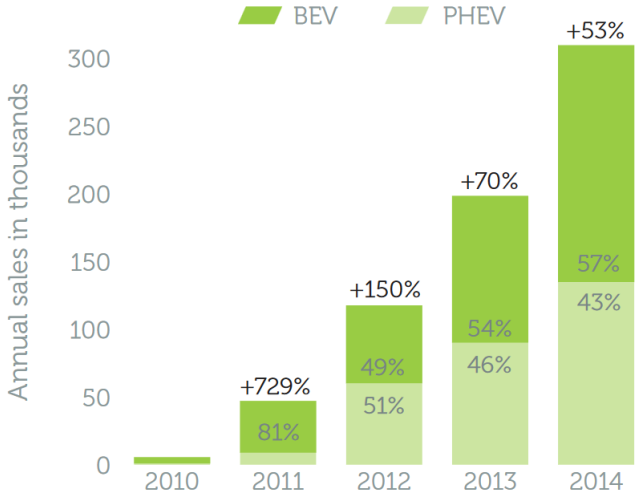


Figure 2: IEA's report of global sales of plug-in hybrid (PHEV) and battery electric vehicles (BEV) [2]. The data is gathered from the 29 member countries. The sales have grown annually, reaching 300 000 sold units in 2014.

1.2 Motivations and goal of the thesis

Visedo Ltd. currently manufactures electrical drive systems for use in mobile work machines, buses and marine vessels. This includes different power converters: the PowerMASTER^{TM} inverter, the PowerBOOST^{TM} DC/DC converter and the PowerCOMBO^{TM} multiconverter.

These converters use embedded software to control different features, e.g. motor control, analog and digital inputs and outputs and communication protocols. As the customer base of Visedo grows and more offers are realized, the testing of embedded software becomes especially important in the product life cycle, as it decreases the number of device failures and costly warranty returns. Automating the testing process can also free resources and remove some of the need for ineffective repetitive manual testing. Figure 3 illustrates how the cost of a software fault grows exponentially during the life cycle of a system.

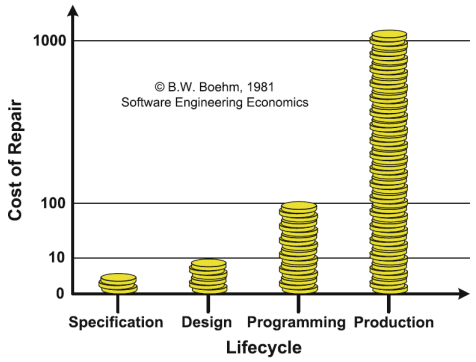


Figure 3: The cost of repair of a software fault during a system's life cycle, as illustrated by B.W Boehm. The cost grows exponentially the later the fault is detected. Note that the values for the cost of repair are relative (e.g. the cost is ten times higher for the production phase compared to the programming phase). [3]

The goal of this thesis is to develop a test framework for testing the software releases of the power converters and also serve as documentation for the use and functionality of the framework. The basic infrastructure for the testing of converter software's features is already existing. It consists of test scripts written in the Ruby programming language and generates Extensible Markup Language (XML) type JUnit test reports of the executed tests. The tests are run from a web based continuous integration tool called Jenkins. This system was used as a basis for the framework. Important criteria for the framework were:

1. The generation of more detailed test reports. The generated XML reports aren't very readable and can't be published and integrated into a quality system as is. They also lack any detailed information about the tests including test pass criteria and test case specifications. Because of this a system was needed for creating more readable reports, that could be added into Visedo's quality system and if needed presented to customers and auditors, e.g. the classification society.

2. Modularity, so that new tests could be easily added in the future.

3. Reproducibility, so tests can be easily re-run and re-produced in the case of a failure.

The thesis focuses on a practical approach of developing a test framework. One possible way to construct the framework is presented and the software tools used in building it are described. The software development process, even though important, is discussed only briefly. The financial aspects of testing (e.g. cost-benefit analysis) are not discussed in this thesis.

For testing the framework itself, the PowerMASTER M-frame inverter was chosen as a case device. The PowerMASTER, shown in figure 4, is a heavy-duty inverter designed for electric or hybrid drive trains of mobile work machines and marine vessels. Typical applications include controlling the speed and torque of electrical traction motors, converting AC from a generator to DC for energy storage and being an active front end for connecting to AC grid. [4]

Figure 4: The PowerMASTER inverter.

For the purpose of this thesis the SAE J1939 communication protocol was chosen as the single testable feature of the inverter. Two example test suites were written for testing the sending and receiving of output and input messages through the communication bus.

1.3 Structure of the thesis

The *theory* section gives an introduction to software testing and its different levels. Some characteristics specific to embedded software testing are also presented. Unit testing and unit test frameworks are discussed in more detail, as a unit test framework is a central part of the overall test framework. Test automation and its benefits are discussed. An introduction to CAN bus and the J1939 protocol is also given. Inverters and their working principles are not discussed in the *theory* section, but to accommodate the reader an overview of inverters is presented in appendix A. The *Visedo's software release and testing process* section describes the structure of software releases for the power converters and the intended testing process. The test framework's part in this process is described. *The test framework* section first gives an overview of the framework and then presents the different parts of the framework in more detail. The *example tests* section addresses the created test suites and the test setup used to run them. The *results* section evaluates whether the framework fulfilled the given criteria or not. The deployment of the framework at Visedo is discussed. Future work considering the framework and the software testing process at Visedo is also presented. In the *conclusions* section the framework's impact on Visedo's software testing process is discussed.

2 THEORY

2.1 Software testing

The main objective for software testing in general is to reveal the existence of faults during the different development phases of a software product and to help ensure that required quality for the software can be achieved. The data collected during testing is also useful for the debugging, maintenance and reliability of the software and improving the software development process. Testing can be seen both as a destructive and constructive activity, as the goal is not only to prove the correctness of the software, but to also discover as many defects as possible. [5, 6]

2.1.1 Different levels of software testing

Software testing can be divided into the following levels depending on the scale and purpose of the tests:

- **Unit testing**. Unit testing is used to test different units of the internal structure of the software, i.e. the source code itself. The purpose of unit testing is to detect faults related to logic and implementation of each unit. A unit can be for example an individual function, a procedure or a method. Unit testing is usually performed by the software developers. [3, 5, 6, 7]
- **Integration testing**. Integration testing is used to verify that the previously tested different units work together in a desired way, e.g. that units don't influence other units in undesired way, sub-functions don't produce unexpected results when combined and global data structures don't cause problems. Common techniques in integration testing are top-down integration, bottom-up integration and "big bang" integration. The top-down method requires stubs for missing high-level components, whereas bottom-up method requires drivers for missing high-level components. In the "big bang" method all components are tested together at once. [5, 7, 8]
- **System testing**. System testing is used to check that the software works as part of the whole system with the target hardware and other required components. System testing is usually performed by dedicated testers who do not know the source code of the software. Proper documentation in this level of testing is important. System testing should assure that the software meets the level of quality set by the customer. [5, 7, 9]
- **Acceptance testing**. This is performed by the customer or an internal department to verify that the product meets the expectations and specifications. Therefore acceptance testing can be part of the software development contract. [5, 7, 9]
- **Regression testing**. This testing is done when changes to the software are made after its release, e.g. new features are implemented or fixes are made. Regression testing is used to verify that these changes did not break any existing functionality. If features are added or removed compared to the previous software version, the regression tests have to be modified accordingly. New test cases are created for new features, and/or old test cases are removed as redundant if the feature they test is no longer present. [5, 7, 10]

One common software development model, that incorporates the above testing levels in the whole development process, is the V-model, shown in figure 5. The V-model describes the different phases of the software development process and the level of testing and the corresponding test plan associated with it.

Software testing is also frequently divided depending on the test method to white-box and black-box testing.

- **White-box testing** is usually done by means of unit testing and it requires knowledge of the internal structure of the software, i.e. the source code itself. [6, 11]
- **Black-box testing** is used to test a complete module, feature or the whole system. Black-box testing verifies that when an input is given to the software the correct output is produced. It therefore focuses on the behavior of the software and the internal structure isn't required to be known to the tester, i.e. it's a black box. Integration testing often uses black-box testing as it's method. [6, 11, 12]

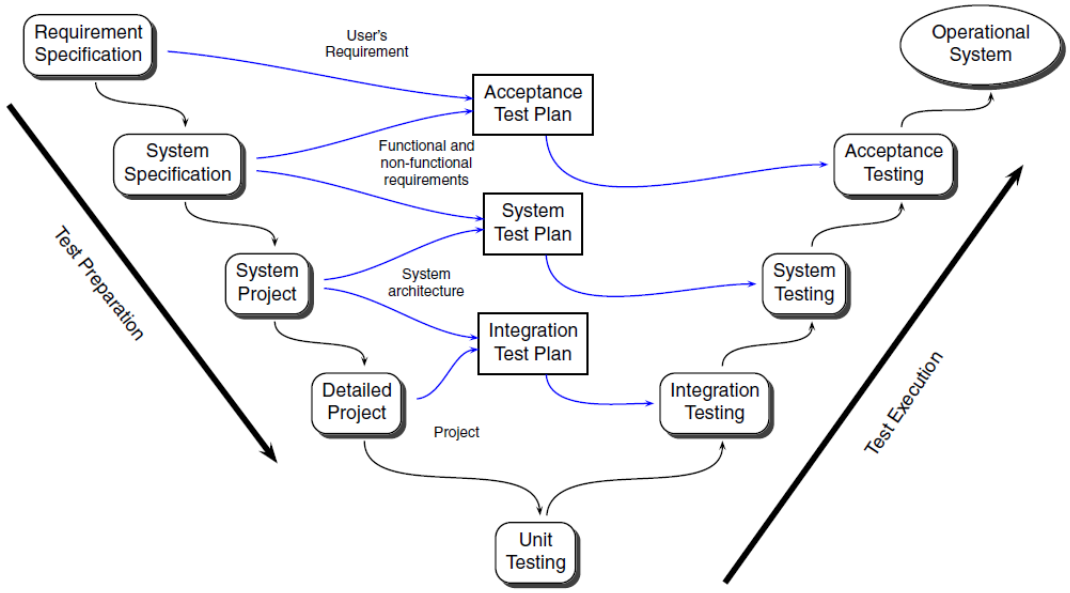


Figure 5: The V-model. The V-model connects the different testing levels (on the right) with their corresponding software development stages (on the left). The test plans associated with each testing level are also presented. [5]

2.1.2 Characteristics specific to embedded software testing

What differs embedded software testing from general software testing is the constant presence of a target hardware platform. The software and hardware are often simultaneously developed, and the software has to take into account the hardware it is meant to run on. This can pose problems for testing the software, as the hardware might not be available until late in the development. Because of this, different ways to mimic the target hardware are used in software industry. [13] Therefore the testing of embedded software is often divided to two categories:

- **Host-based testing**. This is performed on the host-machine (i.e. a PC or a laptop), in the same environment as the development of the software. In host-based testing the behavior of the target hardware is mimicked by simulation and/or emulation. Some examples of this are different hardware mocks, the use of evaluation boards and hardware simulators. [7, 13, 14]
- **Target-based testing**. In target-based testing the software is compiled and uploaded to the actual target hardware. Therefore this kind of testing can be done only after the target hardware has been developed. [7, 14]

Host-based testing alone cannot verify that the software works in the target environment and therefore target-based testing is always necessary for embedded systems [7].

2.2 Unit testing and unit test frameworks

Unit tests are run by software developers to verify that the software's different units of code work and behave as intended. A unit is the smallest testable part of an application. It can be for example an individual function, a procedure or a method. [3]

Unit tests are usually performed by using a unit test framework. Different frameworks exists for different programming languages, some examples being JUnit for Java, CppUnit for C++ and Unity for C. [3] Most of the frameworks have the same basic structure, which is shown in figure 6.

The library contains the assertions of the framework. An assertion is the basic element of a unit test. It checks whether a certain statement evaluates to true or false. In unit testing a failed assertion results

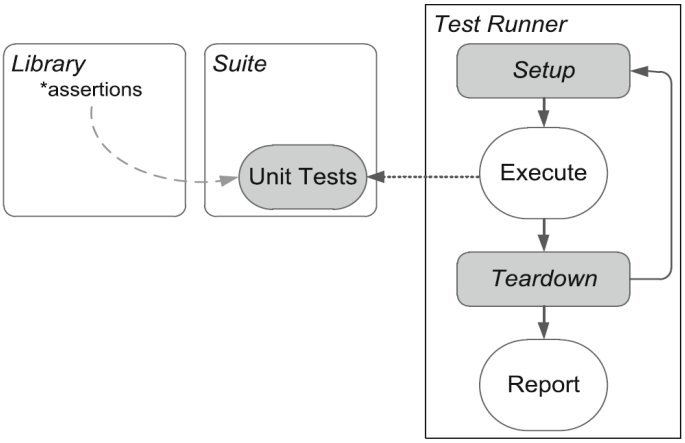


Figure 6: The basic structure of a unit test framework. The framework consists of the library, unit tests and test runner. [3]

in a failed test. [15] Listing 1 shows an example of an assertion from the Ruby programming language's Minitest framework [16].

Listing 1: Example of an assertion from Minitest.

```
1 x = 10
2 y = 20
3 assert_equal(x, y, "x is not equal to y")
```

The *assert_equal* method simply compares the values of x and y. If the values are equal, the assertion succeeds. In this case however, as the values are not equal, the assertion fails, and the failure message ("x is not equal to y") is outputted. *Assert_equal* is only one type of assertion. Different types of assertions exist for any given framework. Minitest, for example, has multiple different assertions in addition to *assert_equal*, e.g. *assert_empty* - fails unless given object is empty, *assert_in_delta* - compares floats, fails unless values are within delta of each other, and *assert_in_epsilon* - compares floats, fails unless values have a relative error less than epsilon [17].

The test runner calls the unit tests, setup and teardown. Setup provides the necessary environment for executing the unit tests. After the tests are executed, teardown cleans the test environment, so that the executed tests won't interfere with other tests. If an assertion fails, the test runner will output a message, which contains information of the failed assertion. Finally a report of the failed and successful assertions is outputted. For organizing purposes, unit tests can also be grouped into test suites. [3]

Unit tests can generally be written in two ways: before the implementation of code, Test Driven Development (TDD), or after, Test After Development. Test Driven Development is a more recent approach to testing. The developer writes the tests before any code for a feature they want to implement. When the tests are run, they should fail, as the code for the testable feature doesn't exist yet. Then the code for the feature is written and the tests are run again to verify that it works as intended. A more traditional approach is Test After Development. In Test After Development the code for a new feature is written first. After this unit tests to verify the code are developed and ran. [18] Figure 7 illustrates these two approaches.

2.3 Test automation

Software developers often lack the time to test software sufficiently due to tight delivery deadlines. Test automation can help overcome this issue by providing automated testing systems, e.g. test execution and management tools and test automation frameworks. Benefits of test automation include:

- Time to market. When performing e.g. regression tests, less time is lost verifying whether the product is ready for release or not. Therefore the time to market does not suffer as much from time lost in running the tests. [19]

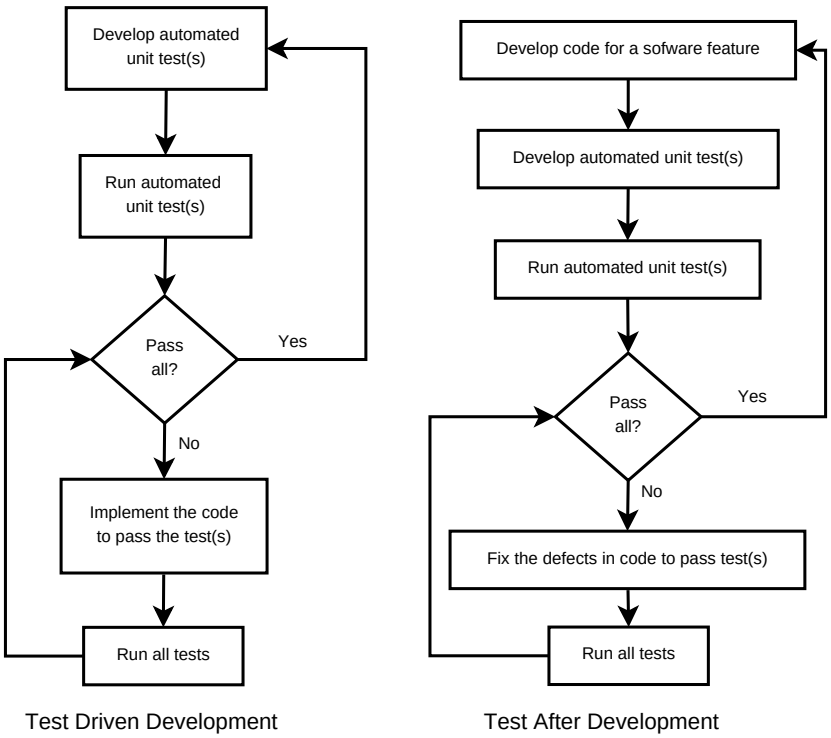


Figure 7: Test Driven Development versus Test After Development. In Test Driven Development the tests are written before any code and in Test After Development vice versa. Own illustration, based on [18].

- Reproducibility. A certain amount of variance is always present when tests are run manually. Using automated test scripts makes it possible to run the tests exactly the same way every time. [19]
- Controllability. Test scripts that show precisely what was tested and how can be used to prove that enough testing was performed e.g. for certification purposes. [19]
- Test motivation. Test that are repeated manually constantly can result in test fatigue. The tester may be less motivated to run the tests and less objective in observing the test results. Test automation reduces this sort of repetitive work and gives the tester time to focus on more productive work. [19]

The tools used in test automation can be divided to dynamic, static and supporting tools. Dynamic tools are used for tests, in which the system is actually used. Static tools are used for example to check the complexity, structure and correctness of the software's source code or to check the system documentation. Supporting tools are used to support the test process, e.g. to perform error logging, planning, test design, reporting of test runs and configuration management. [19]

The use of test tools can have several benefits, for example:

- More testing possibilities. Some tests can be impossible to run manually, e.g. testing an object without a user interface. For these tests a test tool is required to send commands to the object and monitor responses from it. [19]
- Time saving. Some tests take a lot of time to run manually. Examples being security tests, that require a lot of inputs to the system to check for potential security leaks or reliability and endurance tests, that are repeated many times to make sure the system is stable. [19]
- Repeatability. The possibility to execute the same tests repeatedly in the same way. This is useful e.g. for regression and conformity tests. [19]

- Log files. To log information that is impossible to obtain manually. For example logs from performance tests, in which the time between the runs is very small, and therefore logging the results manually isn't accurate. [19]
- Comparing results. Test scripts can be used to compare expected and actual results objectively. For example for large messages sent by a system, manual comparing can be difficult and more error prone. [19]

Not all tests can be automated. Some systems can have requirements that change constantly, be highly complex or have high uncertainty. Manual testing is required for these cases. Test automation is best suited for tests, that are run repeatedly, as this maximizes the benefit gained from automation. It should also be noted that although test automation can save software development costs, it itself isn't free. The test tools have to be implemented and maintained by a developer or a dedicated tester. The test tools can also have bugs and defects of their own. Complex tools can also cause errors. Therefore time is lost in learning and debugging the tools. A cost-benefit analysis is often needed to verify that implementing test automation is worthwhile. [19, 20] However, as the financial aspects of testing are outside the scope of this thesis, they aren't discussed further.

2.4 Controller Area Network (CAN) and the SAE J1939 protocol

As the J1939 communication was chosen as the testable feature for the example tests presented in this thesis, this chapter provides an overview of the Control Area Network and the J1939 protocol.

The Controller Area Network (CAN) is a serial bus communication protocol, that is standardized by the International Organization for Standardization (ISO). It defines the communication between different network nodes (sensors, actuators, controllers, etc.) in real-time applications, e.g. vehicles and industrial automation. [21, 22] Figure 8 shows a typical CAN bus line with multiple nodes.

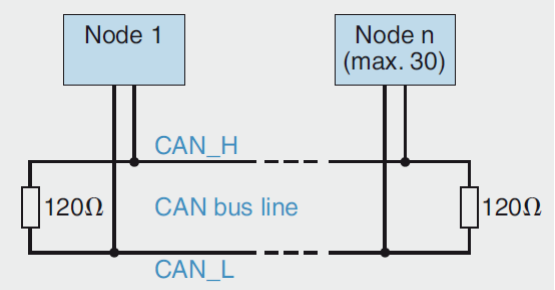


Figure 8: Typical CAN bus line with multiple (n) nodes. The bus lines are terminated with 120 Ω resistors to dampen the reflections of the electrical signals. [23]

CAN bus uses two states for communication, the recessive state (a binary 1) and the dominant state (a binary 0). The data transmission is differential, the CAN_L level is subtracted from the CAN_H. For high-speed CAN, a voltage of 2,5 V on both lines represents the recessive state. In the dominant state, the voltage of CAN_H is 3,5 V and CAN_L 1,5 V. [23] This is illustrated in figure 9.

The J1939 is a higher layer protocol build on top of the CAN protocol and developed specially for vehicle applications by the American Society of Automotive Engineers (SAE). J1939 protocol defines the upper layers of the Open Systems Interconnection (OSI) communication model illustrated in figure 10, whereas CAN protocol defines the lowest two layers, the physical layer and the data link layer. J1939 protocol is a recommended practice, that defines how and what data is communicated between different electronic control units (ECU). J1939 specifies e.g. how to read and write data and how to calibrate subsystems. The speed of J1939 is 250 kbit/s. Applications of J1939 include truck-and-trailer communication, vehicles in agriculture and forestry and marine navigation systems. [21, 24]

The messages send with J1939 are usually broadcast, meaning that the data is transmitted without a specific destination. Therefore any device in the network can use the data without additional request messages. A specific destination address can be included in the message identifier, if the message needs to be sent to a particular device. The different signals that belong to the same topic are defined as

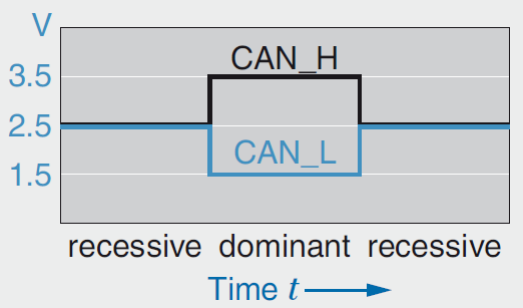


Figure 9: Voltage levels for high-speed CAN. 2,5 V on both lines represents the recessive state (a binary 1). 3,5 V on CAN_H and 1,5 V on CAN_L represents the dominant state (a binary 0). [23]

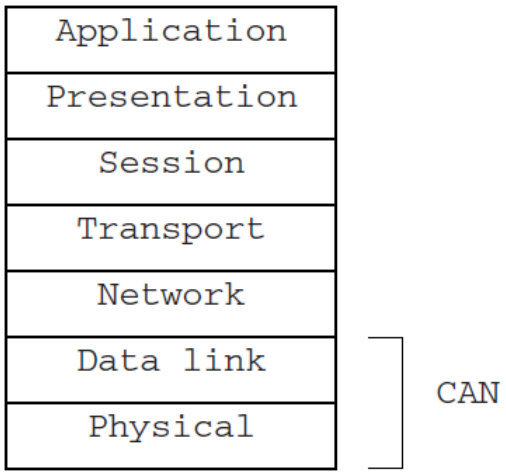


Figure 10: The OSI model. CAN protocol defines the lowest two layers, the physical and data link layers. Higher level protocols, e.g. J1939, are needed to define the upper layers. Fieldbus protocols often don't define the session and presentation layers, as they are not needed in those applications. [21]

parameter groups (PG). Each group has a parameter group number (PGN) to identify them. The transport protocol (TP) of J1939 handles the message packaging, reassembly, connection management, flow control and handshaking for the sent messages. [25, 26] Figure 11 shows the message sequence for a multi-packet message in the case of broadcast and point-to-point transmission.

If the transmitted message is of broadcast type, first a broadcast announce message (BAM) is sent. The BAM message is received by all nodes in the network and allows the nodes interested in the message to prepare for it, e.g. reserve the necessary amount of buffer memory. Following the BAM message, the actual data is transmitted using sequential data transfer (DT) messages. If the transmitted message is of point-to-point type, a request to send (RTS) message is first sent by the transmitting node. The receiving node sends a clear to send message (CTS) in response. After this the transmitting node sends the portion of the data specified by the CTS message. This cycle continues until all data is sent. [26]

J1939 uses the 29-bit identifier of the CAN 2.0B protocol. The identifier consists of the priority, reserved, data page, PDU (Protocol Data Unit) format, PDU specific and source address parts. The first three bits of the identifier define the messages priority, the value of 0 having the highest priority. Time critical messages, e.g. a torque control message, are often given higher priorities. The next bit is reserved and is set to 0 for transmitted messages. The data page bit is used to select the data page, each data page containing a set of parameter groups. The PDU format tells whether the message is to be broadcasted or sent with a destination address. If the message is sent with a destination address the PDU specific field contains the destination address. If the message is broadcasted, it contains a group extension. The source address field contains the address of the device transmitting the message. The reserved, data page, PDU format and PDU specific fields form together the parameter group number. A

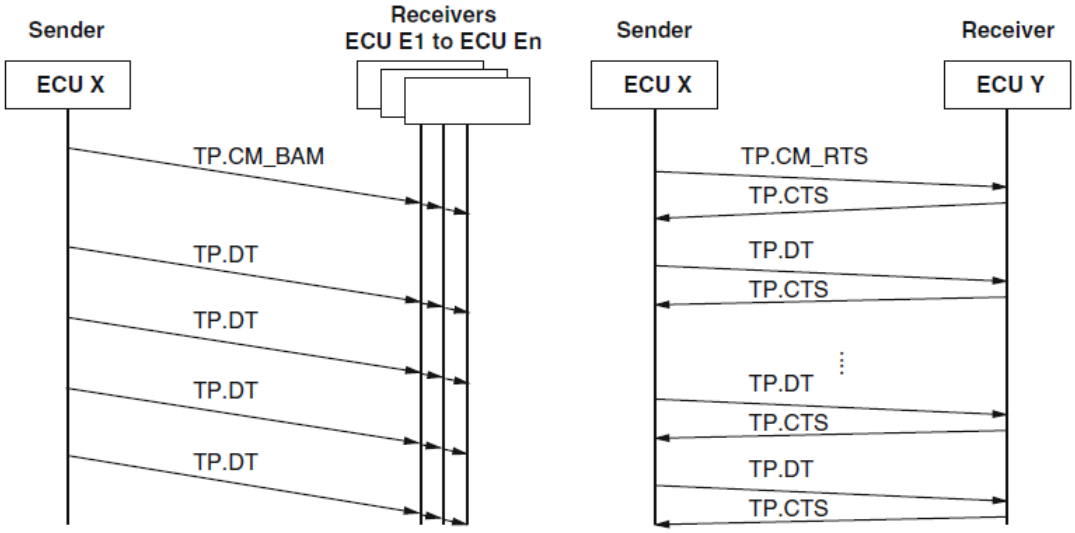


Figure 11: The message sequence for a multi-packet message. On the left the sequence for a broadcast transmission, on the right for a point-to-point transmission. [26]

single PDU of J1939 consists of the 29-bit identifier and up to 8 bytes of data. [25, 27] The structure of a PDU is illustrated in table 1.

PDU						
29-BIT IDENTIFIER						DATA
PRIORITY 3 bits	RESERVED 1 bit	DATA PAGE 1 bit	PDU FORMAT 8 bits	PDU SPECIFIC 8 bits	SOURCE ADDRESS 8 bits	 0..8 bytes
	PARAMETER GROUP NUMBER					

Table 1: A PDU of J1939. The PDU consists of the 29-bit identifier and up to 8 bytes of data. The reserved, data page, PDU format and PDU specific fields form the parameter group number of the identifier. [27]

A parameter group consists of suspect parameter numbers (SPN). A SPN is an identifier for a single parameter. For example, a parameter group for the engine temperature might have SPNs of the engine coolant temperature, engine oil temperature and fuel temperature. A SPN contains information of the data length, resolution, offset, data range and type of the parameter and the corresponding PGN. [28] Figure 12 shows an example of a parameter group named Engine Temperature containing six SPNs.

Visedo's PowerMASTER inverter doesn't implement the standard PGNs and SPNs included in the SAE J1939 standards. A set of custom messages is specified for the inverter. Figure 13 shows an example of a J1939 parameter group of the inverter taken from Visedo's communication manual. The parameter group contains two SPNs, the *cmd_mot_run* and *cmd_mot_ctrl_mode*, that combined are six bits long.

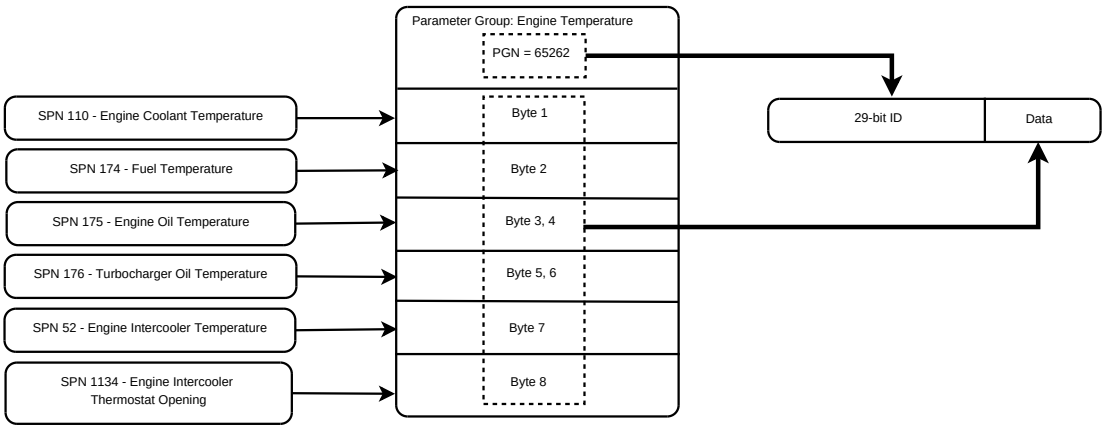


Figure 12: An example of a parameter group. The group Engine Temperature contains six SPNs that form a total of 8 bytes of data. The PGN is 65262 and is stored in the 29-bit identifier. Own illustration, based on [28].

PGN: 0xFF21
Priority: 3
Timeout: 50 ms
Source: 0x27
DLC: 8

	7	6	5	4	3	2	1	0
0	7	6	cmd_mot_run 5 msb	4 lsb	cmd_mot_ctrl_mode 3 msb	2	1	0 lsb
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	61	60	59	58	57	56

Figure 13: An example parameter group of the PowerMASTER inverter. The group contains two SPNs. The *cmd_mot_ctrl_mode* is four bits long, and is used to select the control mode for the inverter (speed control, torque control, etc.). The *cmd_mot_run* is two bits long and is used to start or stop the modulation of the inverter.

3 VISEDO'S SOFTWARE RELEASE AND TESTING PROCESS

3.1 The structure of a software release for a power converter

Currently Visedo develops a lot of custom drive systems, and therefore the software is developed according to customer needs and specifications. In result of this, different software variants are developed for the power converters. Figure 14 shows the software stack and its different layers for a power converter.

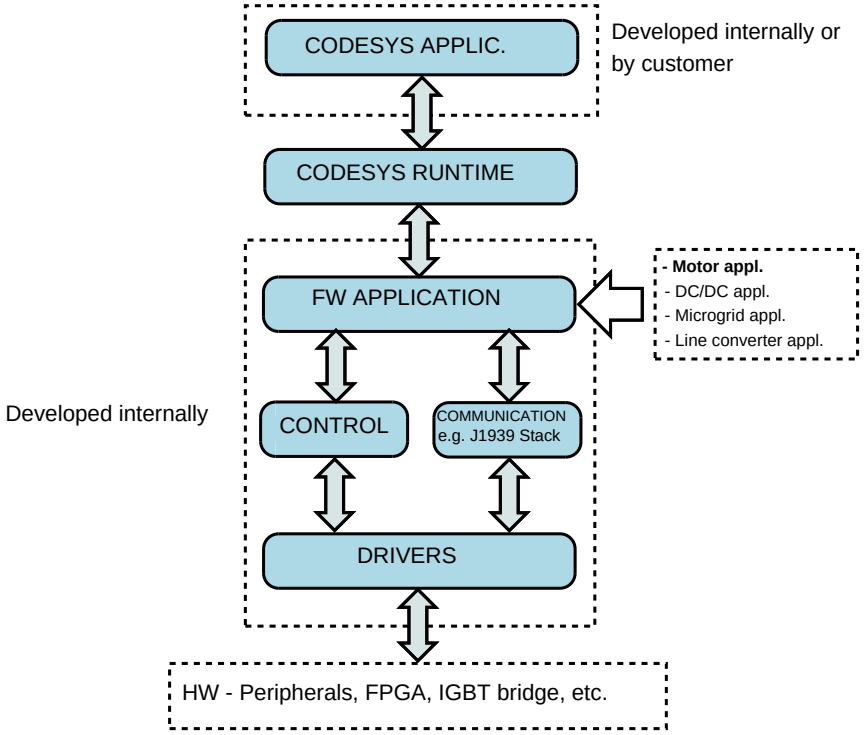


Figure 14: The different layers for a software release. The drivers, control, communication, and FW (firmware) application are developed internally at Visedo. The CODESYS application can be developed either by the customer using Visedo's base application as a reference, or by Visedo.

The software consists of the driver, control, communication, firmware (FW) application and CODESYS application layers.

- The driver layer acts as an interface between the software and hardware components of the inverter.
- The control layer contains the controls algorithms for the inverter, e.g. motor speed control, current and voltage control, etc.
- The communication layer handles the communication to and from the inverter, e.g. speed references, speed limits, run and stop commands, fault statuses, etc. Currently two communication protocols exist for the converters, the J1939 protocol and the CANopen protocol.
- The firmware application controls the converter as part of the whole drive system and commands the control layer of the software. Different applications are in use for different power converters and multiple applications can be in use simultaneously. The combination of applications is hardware based and depends on the functionality of the device. For example, for the PowerCOMBO multiconverter, a combination of a DC/DC converter and an inverter, the motor control application and the DC/DC application are in use. In the case of this thesis and the PowerMASTER, the only application implemented is the motor control application.

- The CODESYS application layer is the highest layer of the software. If the wanted functionality for the drive system can't be achieved using the FW application alone, then the CODESYS application can be used by the customer or by Visedo to develop additional features on top of the FW application. The CODESYS applications are implemented using IEC (International Electrotechnical Commission) 61131-3 Structured Text (ST) programming language. Visedo provides a base application as a reference for the customers. The CODESYS runtime is used as an interface between the FW application and the CODESYS application.

3.2 Current testing process and its problems

Currently the testing for the software releases is largely ad hoc based. Unit tests and code checking are done in the development phase, but after a software variant is released, the testing for the whole release is mainly manual and based on previous experience regarding defects. Figure 15 describes the current testing process for the software variant releases.

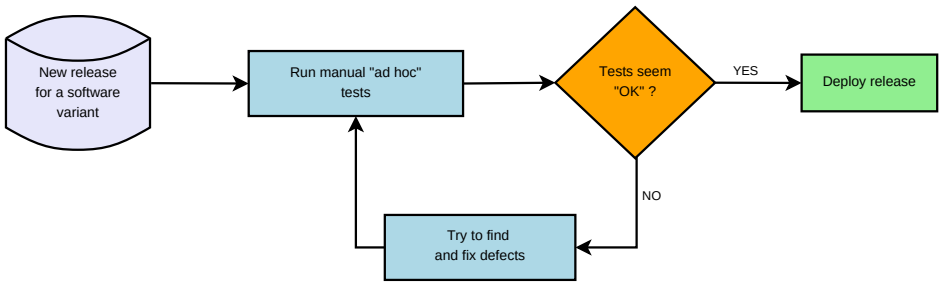


Figure 15: The current testing process for the different software variants.

After a new release for a software variant is pushed out some manual ad hoc tests are run for it. Problems with the current testing process include:

- The tests are time consuming as they have to be executed manually. For example different control values for the inverter are inputted by hand by a human operator.
- Tests can be hard to reproduce. The tests aren't predefined and planned, which can make it hard to reproduce the test runs (and the defects) in case of a failure.
- No test reports are generated. It's harder to keep track of failures and the current status of the releases.

3.3 The test framework and the intended testing process

The purpose of the test framework is to serve as a base for executing tests for the different software release variants. The testing process can be seen as a mixture of regression testing and integration testing. A tailored test set is executed for each new release of a software variant to verify that the new features or fixes implemented in any of the software layers presented in figure 14 don't break any existing functionality of the software release. The tests themselves are integration based and their purpose is to test the releases as a whole, i.e. test the integration of the software layers.

Compared to the current testing process, the tests can be automated to a larger extent with the use of automated test scripts. The framework is also used to generate and host test reports, which helps keep track of the current status of the releases. The tailored test sets make reproducing the test runs easier as the framework should be able to run the same test set for a software variant repeatedly. Figure 16 shows a flowchart of the intended testing process.

Different test scripts, that test the interaction between different layers are and will be developed. One example of these are the J1939 communication tests, that are presented in chapter 5. At the time of writing this thesis, the tests are written following the test after development process as shown in figure 7.

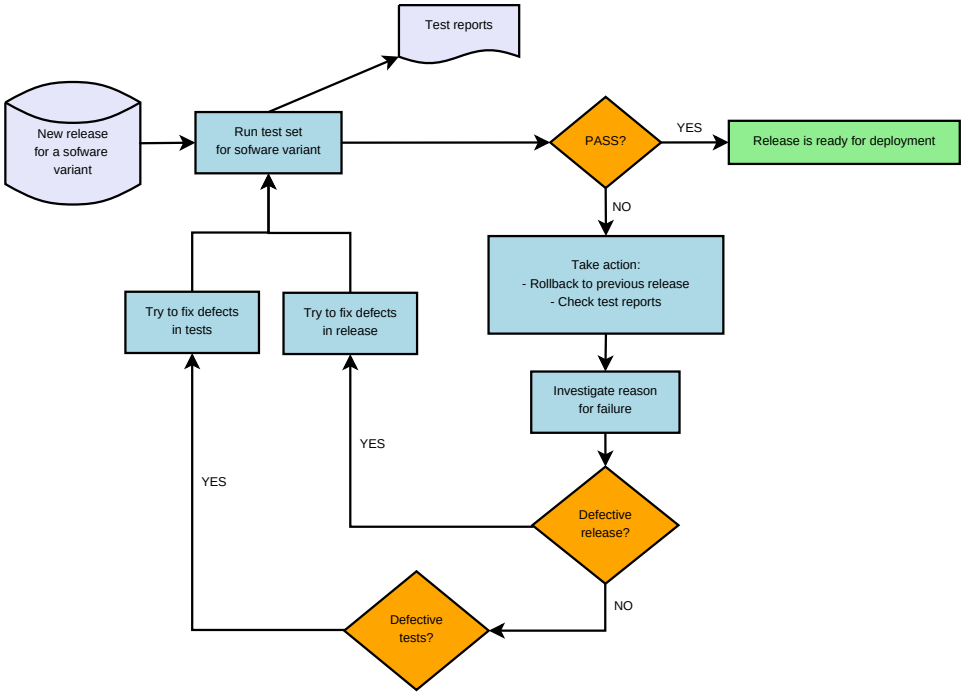


Figure 16: The intended testing process. After a new release for a software variant is pushed out, a test set tailored for that variant will be executed. If the test set passes, the release is ready for deployment. In the case of a failure, the reason should be investigated. The first assumption is that the release itself is defective. However if the defect can't be found in the release, next the tests (and the test setup) are investigated. After investigation corresponding fixes are implemented and the test set is rerun.

The selection of the test set is based on the software variant and its specific features. For example there is no sense in executing J1939 communication tests for a release, if it doesn't implement the protocol in question and instead uses e.g. CANopen. Same features can and do exist in multiple variants. This makes it possible to re-use certain tests across multiple different variants and cut the time and effort required for test development. The test sets are built from test suites, that in turn contain individual test cases of the same theme. Figure 17 shows the overall test structure for two software variants "A" and "B".

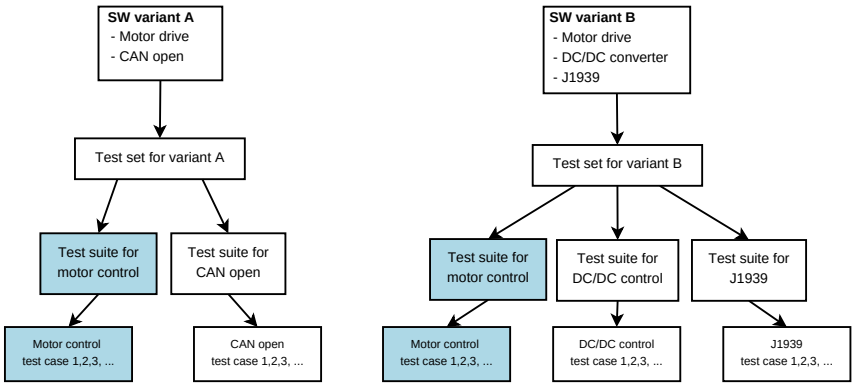


Figure 17: Overall test structure for two example software variants "A" and "B". Variant A implements the motor control application and CANopen protocol. Variant B implements the motor control and DC/DC application and the J1939 protocol. The test sets are formed according to these features. The motor control test suite, highlighted in blue, can be used for testing both variants. Note that this is just an example illustration of a test structure, and doesn't necessarily correspond to reality.

4 THE TEST FRAMEWORK

Chapter 4.1 gives an overview of the framework. Chapters 4.2 - 4.5 present the different parts of the framework in more detail. An introduction to the Ruby programming language is also given.

4.1 Overview of the framework

The testing framework uses test scripts written in Ruby to test different software releases of the power converters. One script corresponds to one test suite, that consists of test cases that test the same feature of the software (see chapter 3.3, figure 17). The core of the framework consists of Ruby's *Minitest* unit test framework and a web-based continuous integration tool *Jenkins*. Jenkins is used as the test front end and a host for the generated test reports. Figure 18 shows an overview of the testing framework.

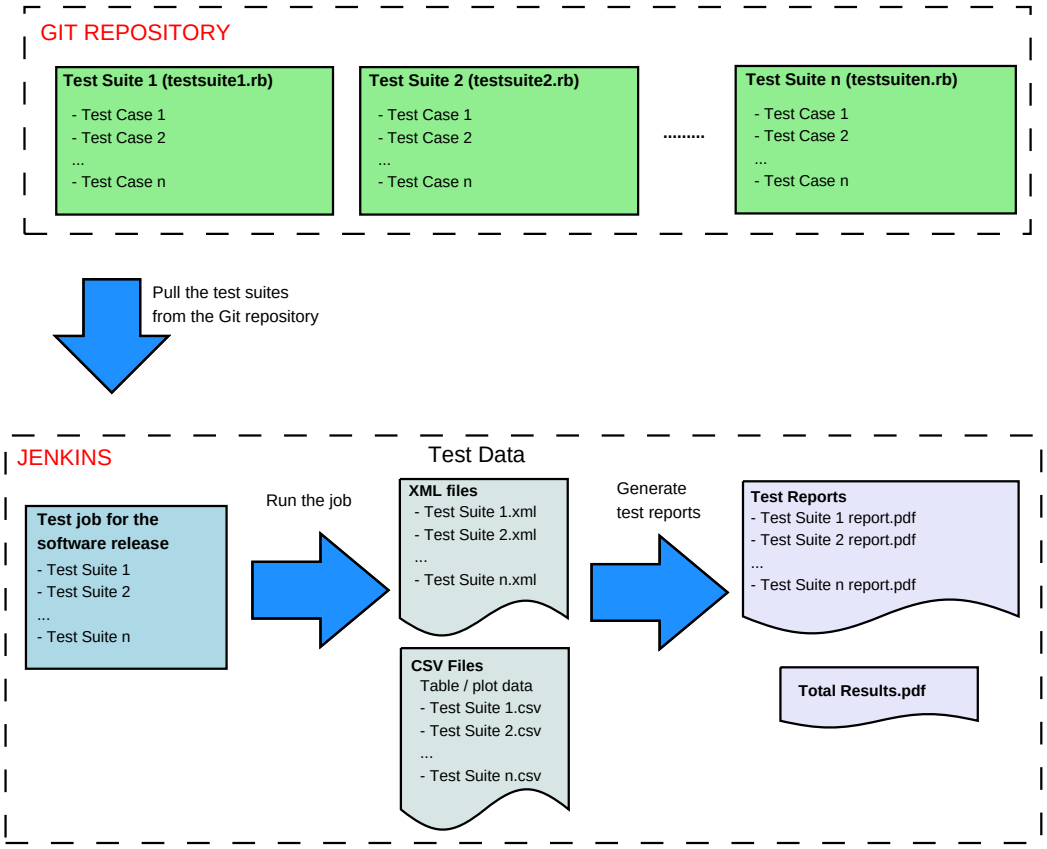


Figure 18: An overview of the testing framework. The test suites reside in Visedo's server in a Git repository. Jenkins pulls, builds and runs the suites. Test reports for each suite and a total results report are generated after the tests.

Jenkins pulls the ruby scripts (the test suites) containing the individual test cases from a Git repository residing in Visedo's server. A Jenkins job is created for the software release, that builds and runs the test suites. An XML file is generated for each test suite containing info about the executed tests (failed and succeeded assertions, test cases, etc.).

After the build, test reports (in PDF format) are generated for each test suite and saved in Jenkins' workspace. Finally a total results report is generated for the whole test. The data for the reports is parsed from the XML files. Also if the user has saved CSV-type (Comma Separated Values) table or plot data during the tests, this is parsed and the corresponding tables and plots are generated into the reports as well.

4.2 The Ruby programming language

Ruby was originally created by Yukihiro Matsumoto in Japan in 1993 and reached notable public recognition by the year 2000. Ruby is an interpreted scripting language. This means, that the source code of a Ruby program (or a script) is compiled when it's executed. In comparison in compiled languages, such as C or C++, the source code is pre-compiled into a binary format and then executed on specific microprocessor type. [29]

Some disadvantages of interpreted languages include:

- Speed. The fact, that the source code has to be interpreted at runtime results in slower execution compared to pre-compiled code. [29]
- Transparency. Everyone who uses your Ruby scripts or programs will be able to see the source code. If the developed programs are open-source this poses no problem. However for e.g. commercial applications this might be unacceptable. [29]

Advantages of interpreted languages include:

- Portability. Interpreted languages can be executed on multiple operating systems and architectures. [29]
- Real-time writing and executing of the source code. [29]

Ruby is also a purely object oriented language. Everything in Ruby is an object, strings, numbers and even truth values i.e. *true* and *false*. Every object has a class. A class has a set of methods, that can be called on objects of the given class. [30]

The main advantages of Ruby in specific are:

- Ruby is an easy to learn, fairly intuitive and clear programming language. [29]
- Ruby is extensible. New functionality can be easily added by installing different libraries in the form of Ruby gems. A gem is Ruby's own self-contained format for a programming library. [29]
- Due to Ruby being an interpreted language, programs and scripts developed with Ruby can be executed on multiple operating systems, e.g. Linux, UNIX, Windows and Mac OS X, and on multiple processor architectures. [29]

4.3 Minitest, Ci-reporter and Rake

Ruby's unit test framework Minitest [16] is used to coordinate the execution of the different test suites. Minitest is a lightweight and small unit test framework for Ruby. It has an extensive set of different assertions and is fairly fast and easy to set up. The ruby scripts (test suites) act as the test runners and contain individual test cases of the same theme. Two example test suites, *TestJ1939StandardV2inputs* and *TestJ1939StandardV2outputs*, were written and are presented in more detail in chapter 5.2.

It should be noted, that in this case Minitest isn't used for it's most common purpose, which is to run unit tests. As described before, the above test suites can be seen more as black-box integration tests. Despite this, it's still a useful tool, as the different assertions are a handy way to verify the success or failure of the test cases in any given test suite. Minitest also outputs a graphical presentation of the test run, when the test suites are executed from the command line. This can be used as a preliminary way to execute the tests without Jenkins. This can be helpful e.g. for debugging the test cases. Minitest is also required to generate the XML reports of the test runs (using the ci-reporter gem described below), which in turn serve as base for the more detailed PDF reports.

Ruby's rake utility [31] is used for calling the test suites. Rake is a make-like tool for Ruby, that can be used to specify different tasks to be run [31]. The tasks are specified in a rakefile. The code of the rakefile for running the above test suites is presented in appendix B. The code is saved with the file name "Rakefile" and the specified test suites can then be run by calling rake in the directory of the file. Figure 19 shows the output of a command line window for an example test run.

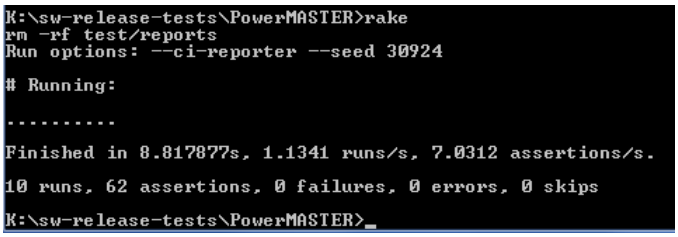


Figure 19: Output of a command line window for a test run using Minitest. The total time, test runs per second and assertions per second can be seen from the output. The total number of runs, assertions, failures, errors and skips are also presented.

To generate the XML files needed for test report generation (see figure 18, section Test Data), Ruby's ci-reporter gem for Minitest [32] is used. Ci-reporter is designed to work with tests, that use rake to run the test tasks, and hooks into different unit test frameworks (in this case Minitest) [33]. Ci-reporter is setup in the rakefile by requiring the gem and specifying the task to use ci-reporter (lines 2 and 10 in the rakefile). An example XML report from a test run of the *TestJ1939StandardV2inputs* test suite is presented in appendix D.

4.4 The test reporting system

The XML reports generated with ci-reporter contain valuable information, but the way the information is presented was insufficient for Visedo's needs. The XML reports also lack any description on what the test cases actually test and how, only names of the test cases are included. A system for creating reports in a more readable and presentable form was needed. The format was decided as PDF for its universal support. The reports were required to at least have the following information:

- The name of the executed test suite.
- List of the individual test cases in the suite.
- The date, time and author for the test run.
- A short description for each test case, that explains what it tests and how.
- Results for the test suite containing number of succeeded and failed assertions and errors.
- Description of failures, if they are present.

An example report generated from the *TestJ1939StandardV2inputs* test suite is presented in appendix E.

The test reports are generated using two scripts written in Ruby, namely *parse-test-reports.rb* and *generate-plots.rb*. The first one is the main script, used to render and save the test reports. The latter is a helper script used for generating plot images from CSV files containing plot data. These images can then be inserted into the reports using *parse-test-report.rb*.

Ruby was chosen as the programming language, because it is well suited for writing scripts and because the tests themselves and the Minitest unit test framework are also written in Ruby. Figure 20 shows a logic flowchart for the *parse-test-reports.rb* script.

The script parses the data located in the workspace directory. The data consists of the XML and CSV files (see figure 18, section Test Data). The file path to the directory is given by the user as a command line parameter. The directory is first checked for the XML files. If found, a list containing the files is created. Then the steps below are performed for each XML:

1. Parse the XML file for info of the test suite.
2. Check if table data or plot images are found for the test suite. The plot images are created from the saved plot data CSV files with the *generate-plots.rb* script.
3. Render all of the above data into a PDF and save it in the reports directory created in the workspace.

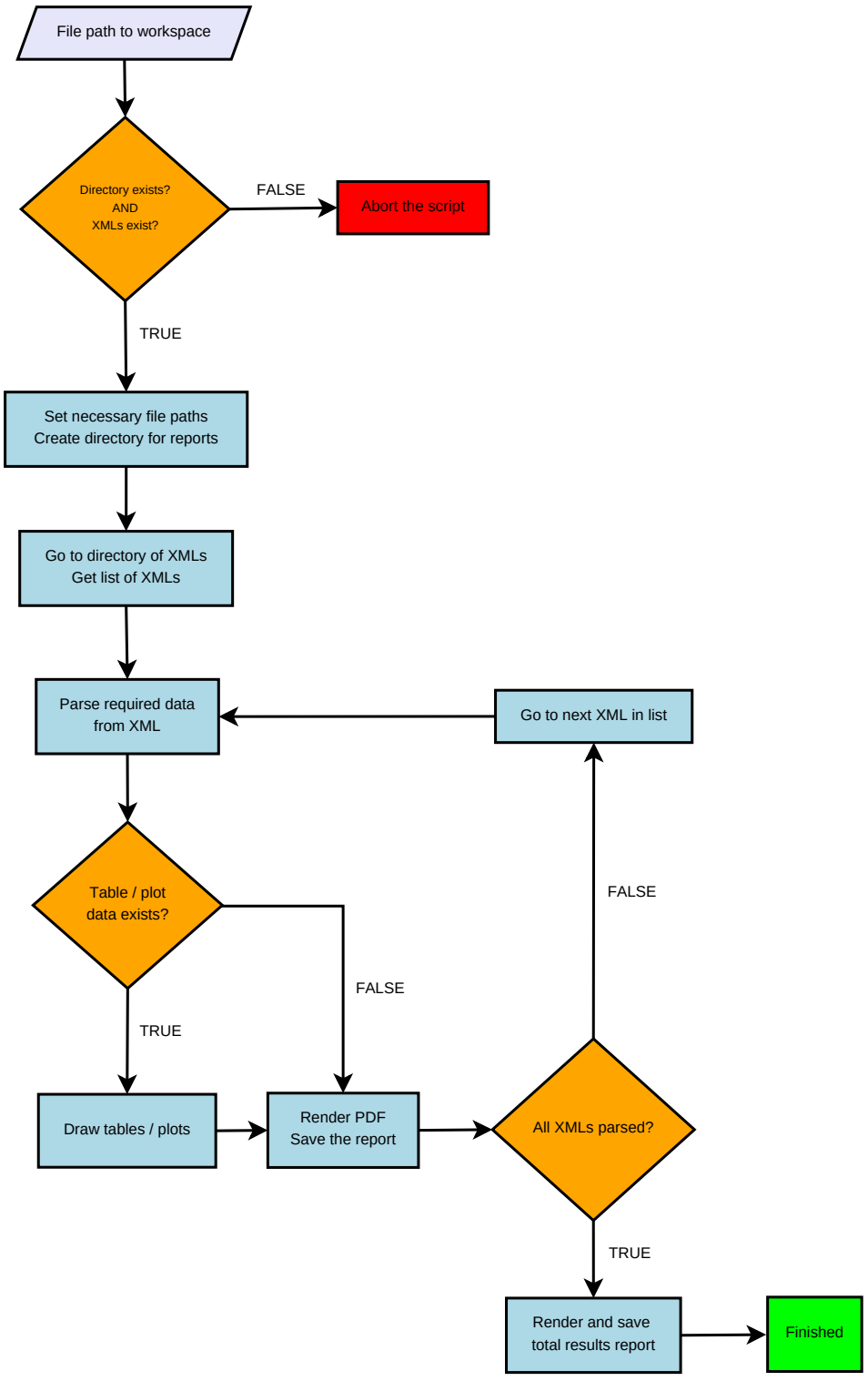


Figure 20: A logic flowchart for the *parse-test-reports.rb* script.

When all reports for individual test suites are generated, a total results report for the whole test is rendered and saved.

The scripts use four software tools — one standalone program and three Ruby gems — for achieving the wanted outcome.

1. Prawn [34]: a Ruby gem, for rendering and saving the reports in PDF format.
2. Nokogiri [35]: a Ruby gem, for parsing the XML files.
3. Gnuplot [36]: a plot drawing tool for various operating systems (e.g. Windows and Linux).
4. Ruby Gnuplot [37]: a Ruby gem, for calling gnuplot in Ruby scripts.

Prawn is a PDF generation library created for Ruby. Prawn was selected because it's lightweight and offers good functionality including tools for repeatable content (e.g. headers and footers) and PNG and JPG image embedding, which is useful for inserting plots into the reports. Nokogiri is a widely used XML parser for Ruby. It was chosen as it offers extensive functionality and has good documentation. Gnuplot was chosen as the drawing tool for plots, because it works both in Windows and Linux, and is lightweight. It is also well known and has extensive documentation.

4.5 Using Jenkins as the test front end

Jenkins [38], a web based continuous integration tool, is used as the test front end. Jenkins acts as a server, therefore allowing access through a HTTP-connection using a web browser. Jenkins can be run on multiple operating systems, e.g. Windows, Mac OS X and different Unix based systems. Different plug-ins can also be installed for Jenkins to extend it's functionality. The version of Jenkins used in this thesis is 2.0.

Continuous Integration (CI) is a software development practice, where each member of a software team integrate their work daily. The integrations are verified by an automated build and tests to detect integration errors quickly. Continuous integration systems usually look for changes in the source repository and when changes are detected, the latest source codes are pulled and compiled. After this automated tests are ran for the build. [39] Jenkins is one example of a CI-tool. However in the case of this thesis Jenkins isn't used for continuous integration. It only acts as a test front end for the regression tests.

The use of Jenkins is based on creating different jobs. A job runs different tasks in the order specified by the user. For testing the software releases a test job is created for the software variant under test, in the case of this thesis, the standard J1939 V2. Different jobs can exist for different major release families, e.g. 6.8 series, 7.0 series etc. Figure 21 shows the main view of Jenkins with two jobs for the 6.8 and 7.0 series Standard J1939 V2 software.

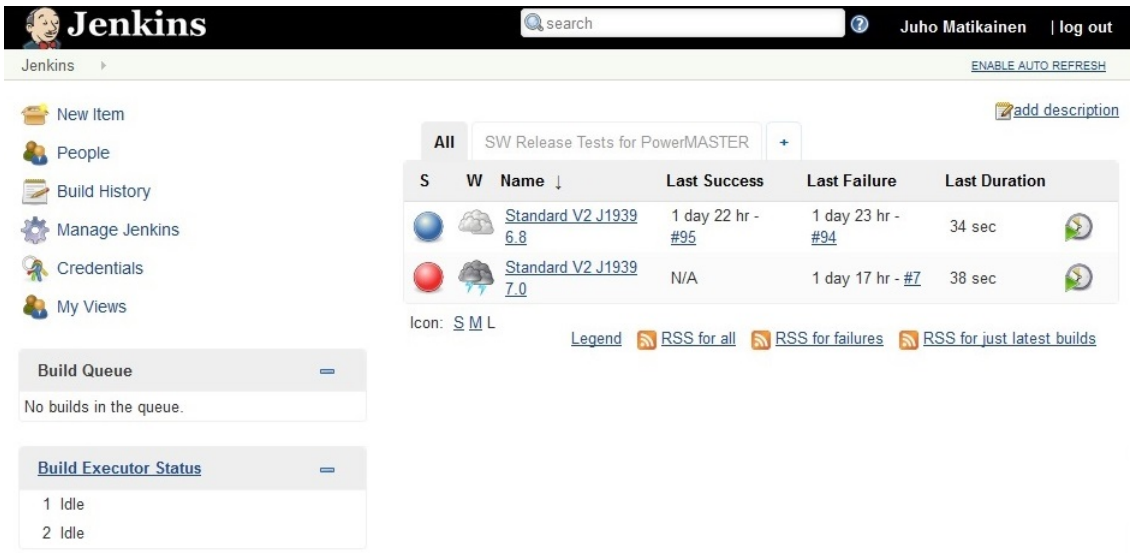


Figure 21: Jenkins' main view with two jobs: *Standard V2 J1939 6.8* and *Standard V2 J1939 7.0*.

The jobs are created as freestyle projects. A single job consists of six parts: general, source code management, build triggers, build environment, build and post-build actions. Build triggers and build environment are not discussed here, as they are currently not used in running the tests.

- General. In the general section the name and description for the job is given and a parameter is defined for the author of the tests for the test reports. A link is also created to the test reports, so that they can be easily accessed after the tests have been executed.
- Source Code Management. In this section Jenkins is told to pull changes from the Git repository containing the test suites and other required scripts.
- Build. In this section four build steps are executed in the following order:
    1. Map network drive (*map-network-drive.bat*). This is a help script used to map Visedo's network drive, so Jenkins can use it.
    2. Upload latest software (*upload-latest-software.rb*). This is a Ruby script, used for uploading the latest software for the specified software variant and release family (e.g. Standard V2 J1939 6.8) to the inverter. The script first finds the latest software release in Visedo's network drive and then uploads it to the inverter. The software variant, release family and the directory containing the software releases are given as command line parameters by the user. The file path to the latest release is saved in a text file so it can be inserted to the test reports.
    3. Update build name. The build name is renamed to include the release number of the software variant under test (e.g. 6.8.34).
    4. Run *rake*. This batch command executes the actual tests for the software by running the rakefile.
- Post-build actions. In this section the JUnit XML files are published. Also the test reports are generated by executing the *generate_report.bat* batch command. This calls the *parse-test-report.rb* and *generate-plots.rb* Ruby scripts. Finally the test reports are archived using Jenkins' build in archive artifacts function.

To achieve all of the functionality above, four plug-ins were installed for Jenkins:

- Post-Build Script plug-in. This plug-in enables the execution of scripts at the end of a build. The plug-in is used to run the *generate_report.bat* batch command.
- Sidebar Link plug-in. This plug-in is used to create a direct link to the test reports inside Jenkins' workspace.
- Nested View plug-in. This plug-in is used to create nested views for grouping different jobs together. This makes it easier to navigate Jenkins if a large number of jobs exist.
- Build Name Setter plug-in. This plug-in is used to update the build name with the release number.

Figure 22 shows the main status view of the *Standard V2 J1939 6.8* job. The test result trend shows the total number of executed test cases (in blue) and failures (in red) for different builds, a build in this case is a single test run. This data is gathered from the JUnit XML files. The last successful artifacts section contains the archived test reports from the last successful test run. Reports for two test suites (*TestJ1939StandardV2inputs* and *TestJ1939StandardV2outputs*) and a total results report can be seen. The build history contains all builds, i.e. test runs, and their results. Test reports for the latest run can be accessed from the sidebar's Test Reports link. To actually run the tests, the build with parameters option is chosen. After this the author for the tests is inputted by the user and the tests can then be executed by choosing the build option.

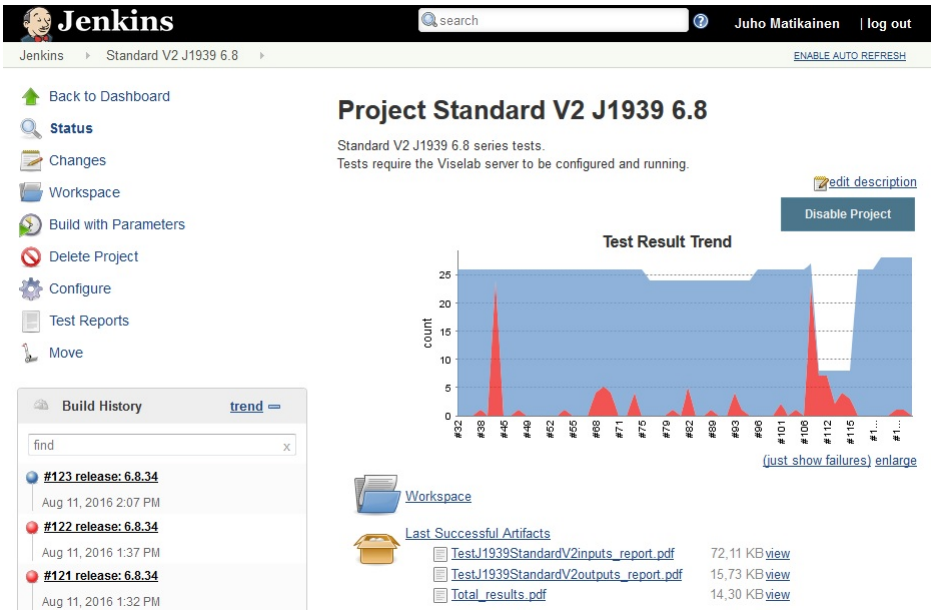


Figure 22: Status view of the *Standard V2 J1939 6.8* job.

5 EXAMPLE TESTS

For demonstrating the test framework, two example test suites were created and executed for the PowerMASTER inverter. The J1939 communication protocol was chosen as the testable feature. The purpose of these tests is to verify that software releases of the inverter handle the communication to and from the inverter correctly using the J1939 protocol. The test suites test the integration of the FW application, control, communication and driver layers of a software release (see chapter 3.1, figure 14), and the main goal for these tests is to verify that the signal flow between the layers is correct. The used test method can be seen as black- box testing, as the internal structure of the layers themselves isn't important, only the correct signal flow between them. The basic principle of a communication test using the J1939 protocol is as follows:

1. Send a message, for example the run command or a motor speed reference with CAN bus, using J1939 protocol.
2. Read the value of the corresponding signal (e.g. speed reference in rpm, run command status: false or true) from the PowerMASTER using the PDP serial bus.
3. Compare the send and read values to check whether they match or are within predefined range from each other (e.g. 0,1 rpm). This is done using the Minitest assertion functions [17], in this case *assert*, *assert_equal* and *assert_in_delta*.

Figure 23 shows a block diagram describing the different hardware and software layers and the signal flow of a J1939 communication test (e.g. sending a motor speed reference) for the PowerMASTER.

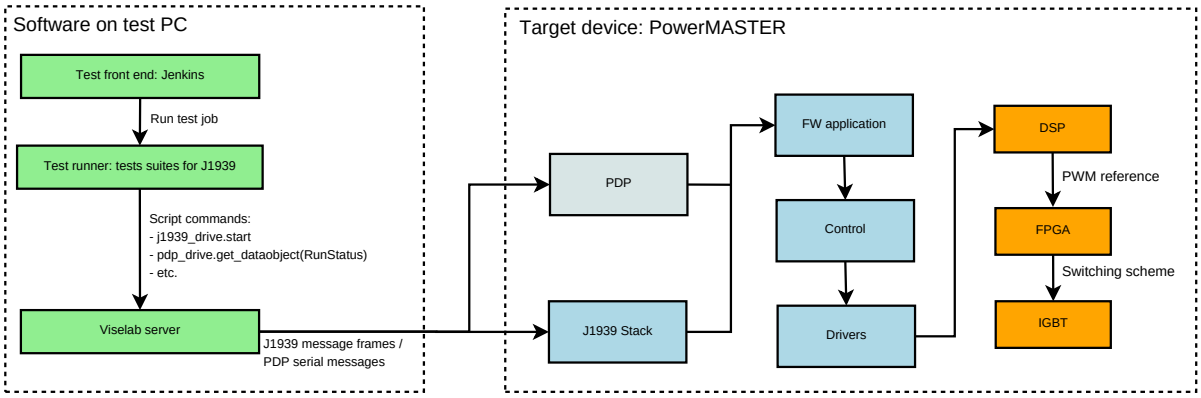


Figure 23: Different parts and signal flow of a J1939 communication test for the PowerMASTER. The blue boxes represent the different layers of the software under test. Orange boxes represent the hardware layers.

The Jenkins job starts the test run be executing the test suites. The test suites command the Viselab server — a software tool developed and maintained by Visedo — using scripts. The Viselab server converts the script commands into actual J1939 message frames or PDP serial messages. The J1939 and PDP messages are sent to the control layer of the inverter through the firmware application. The control layer commands a digital signal processor (DSP) through a driver layer and the DSP creates the PWM (Pulse Width Modulation) references for the field programmable gate array (FPGA). The FPGA in turn creates the switching scheme for the IGBT-bridge of the inverter. The PDP serial bus is used to monitor the J1939 communication. It verifies that the messages sent and read through J1939 are correct compared to what was requested. Therefore in these tests it is assumed that the communication with the PDP serial bus itself is defect free.

5.1 Different test environments and the test setup

At the time of writing this thesis three test environments exist for testing the inverter software:

1. **Simulator**. A simulator program can be used to test features without having access to an actual device.
2. **Test bench**. A test bench consisting of two motors with interconnected shafts (one is driven and the other used as a load).
3. **"Table" tests**. Test ran on the device itself, without it being connected to an actual system, e.g. a motor or electric grid.

The framework can be used to run tests in all three environments. For the communication tests, "table" testing is sufficient. It is enough for the these tests to verify that the messages end up in the control layer of the software correctly. The inverter isn't required to drive an electric motor to verify this. Other tests may require other environments. For example different control tests require feedback from an actual system and are therefore designed to be executed in the test bench.

The tests are target based, the software under test is uploaded and executed in the inverter. A host-PC is used for running the test framework, executing the tests and storing and displaying the test reports. The test setup needed for "table" tests consists of: the inverter, a host-PC with required software, a power supply for the processor PCB, a CAN to USB converter and the necessary cables. Figure 24 shows an illustration of the setup.

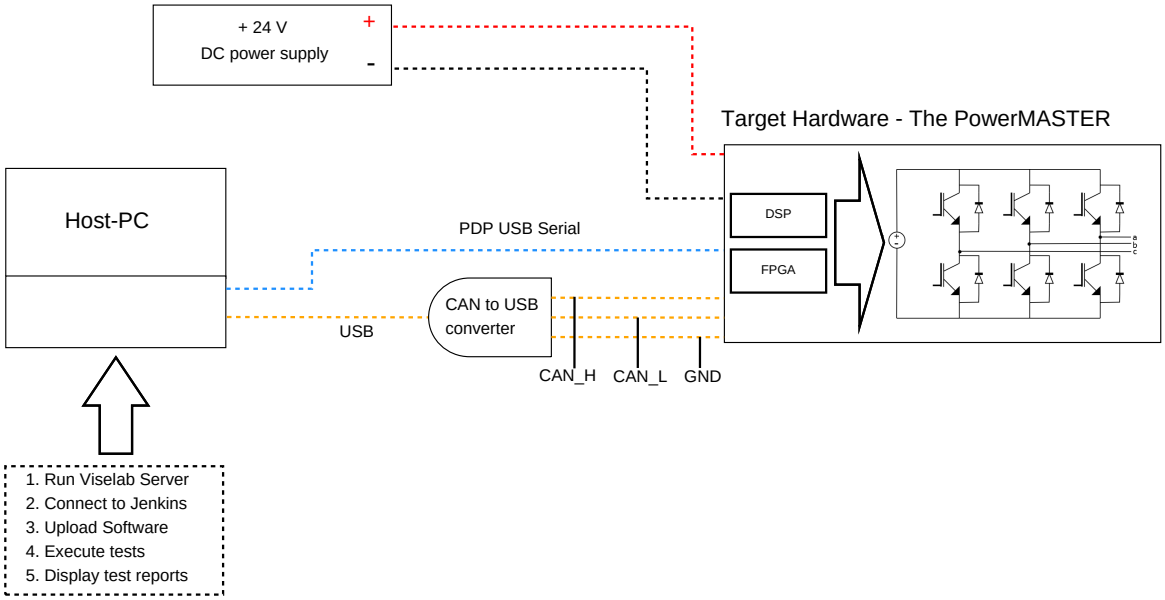


Figure 24: The test setup. The setup uses two communication buses: the CAN bus (in orange) and the PDP serial bus (in blue). The 24 V DC power supply is needed for powering the processor PCB of the inverter. A CAN to USB converter (PEAK-System PCAN-USB) is used for creating an interface between the USB ports of the PC and the CAN bus. The host-PC runs the Viselab server, connects to Jenkins, uploads the software to the PowerMASTER, executes the tests and displays the test reports.

5.2 The test suites

The test suites, *TestJ1939StandardV2inputs* and *TestJ1939StandardV2outputs*, are presented in appendix C. The first suite tests four input messages of the J1939 communication protocol: the run command, the stop command, motor speed reference and motor speed limits. The second test suite tests four output messages: the speed, torque, power and voltage measurements. These test cases are selected based on Visedo's J1939 communication specification and manual, which defines the different messages in use for the inverters. Other messages exist in the specification, but for the purposes of this thesis these eight messages were selected as examples.

The test suites consist of four parts and are executed in the following order:

1. **Before tests**. The before tests method sets up the connection to the Viselab server and sets other required parameters for the tests.

2. **Setup**. The setup method initializes the PDP and J1939 connections and performs a fault reset for the inverter.

3. **Different test cases**: "test_start", "test_stop", "test_speed_meas", "test_torque_meas", etc. These are the test cases performed for the software release.

4. **Teardown**. The teardown method stops the modulation of the inverter and closes the PDP connection.

Also some help methods are defined for use in the test cases, e.g. *ramp_ref* and *lim_test*.

A short description is written for each test case by using the *puts* command, see e.g. line 156 in *TestJ1939StandardV2inputs*. The output of the *puts* command is generated to the XML report of the test suite. The tags around the description (e.g. *info_test_start/* and */info_test_start*) are used by the *parse-test-report.rb* script to identify which description belongs to which test case. The script can then output the descriptions to the PDF reports. The *test_description* method is used to generate a description for the whole test suite in the same manner. In the *ramp_ref* method some test data is saved in CSV files for use in the test reports.

The different test cases are executed in a random order. One test case should not influence another one, i.e. the test cases should be independent of each other. To achieve this, the setup and teardown methods are executed after each test case, which resets the test environment to its original state. This functionality is provided by the Minitest framework. However it is possible to run the test cases using a specified random order using Minitest's seed option. Minitest assigns a seed for each test run, that identifies the order in which the test cases are executed. This makes it possible to replicate the test order for consequent test runs by using the same seed. The seed option is helpful in finding bugs related to the ordering of tests.

6 RESULTS

A test framework was developed for the software releases of Visedo's power converters. Ruby's Minitest unit test framework and a continuous integration tool Jenkins were used as the core parts of the framework. These tools were found to be fairly powerful and easy to configure and extend. This should help in maintaining and further developing the framework. Jenkins although originally developed to support continuous integration, was found to be suitable also as a plain test front end. The Minitest framework was found to be useful not only for unit testing, but also for executing black-box type integration tests.

The developed test framework has improved Visedo's testing process and increased confidence regarding software quality. The use of automated test scripts has decreased the need for manual ad hoc type tests and made test execution more efficient and less time consuming. The J1939 communication tests presented in this thesis have already been used to detect a bug in one of the software releases. Manual testing will still need to be carried out to support test automation, as some of the test cases are simply too difficult or impossible to automate. Tests, that are run infrequently, are also not worth automating. However in result of using automated test scripts the developers and testers now have more time to spend on the more difficult manual test cases.

Three important criteria for the framework were described in the introduction of the thesis:

1. The generation of more detailed test reports. A test reporting system, consisting of two Ruby scripts was implemented into the framework to generate PDF reports of the test runs. Compared to the previous XML reports, the PDF reports are in a more readable form and are also presentable to customers. They also contain descriptions of the test suite and the test cases, which the XML reports lacked. Test data can also be inserted into the reports in the form of tables and plots.

2. Modularity. New tests are fairly easy to add to the framework by configuring the Rakefile of the test set accordingly. However the file has to be modified by hand. Also to add a completely new test set for an untested software variant, a Jenkins job needs to be created and configured accordingly. Jenkins has the option to copy the settings from an old job when configuring a new one, which can be used to speed up this process.

3. Reproducibility. The test runs are more easily reproduced in result of the tailored test sets and the use of test scripts. The same test set can be easily rerun multiple times in the same way using Jenkins.

To deploy the framework at Visedo, a test PC was set up at Visedo's test laboratory. Jenkins and Minitest were installed and configured on this PC. The Jenkins installation at the test laboratory is currently local and therefore only accessed through the test PC. Git was installed and set up to gain access to the repository containing the test scripts and other required scripts for the framework. Ruby, necessary gems for Ruby and gnuplot were also installed.

Currently the framework only runs the J1939 communication tests. Control tests, that are more focused in the control layer of the software releases have also been developed, but these tests are still in the process of being integrated into the framework. The control tests require the test bench, described in chapter 5.1. The J1939 communication tests are also executed in the test bench for convenience, even though for these tests the test bench isn't necessarily required. All the tests in the framework are triggered manually by a tester.

Another Jenkins instance is installed in Visedo's server and has already been in use before this thesis. It is to be used in conjunction with the test framework at the test laboratory. This Jenkins runs tests for the source code of the releases, including unit tests, static tests and code checking. Test performed in the simulator are also executed in this Jenkins instance. All tests on the server side Jenkins are triggered automatically when new code is committed by the developers. The server side Jenkins is available from all computers in Visedo's local network, provided that the user has been granted access by the administrator. Figure 25 describes the current Jenkins installations in use at Visedo.

Overall the goals of the thesis were reached. The developed framework satisfied the criteria assigned for it. However the framework in itself does nothing, and the real work now lies in developing the scripts to be executed in the framework. The scripts presented in this thesis test only a small part of the releases and are not sufficient by any means. Additional test scripts need to be developed for other features. To get all of the different scripts to run in the framework at the same time may require quite a lot of work,

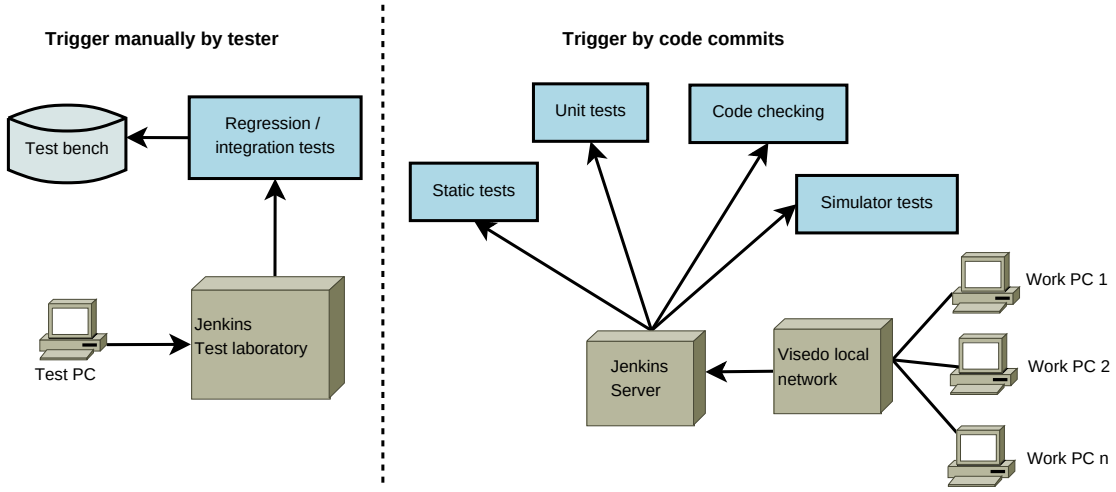


Figure 25: Jenkins installations. On the left, the installation, that is part of the framework developed in this thesis. On the right, the installation already in use.

as the complexity of the test sets increase with every added test. The test reports are also currently only preliminary and will require some work to fit the company standards. The generation of the test case descriptions is also currently quite cumbersome as it requires the use of tags inside the test scripts. Aside from developing the framework, another goal for the thesis was to serve as documentation for it. The thesis should give a satisfactory description of the framework, it's structure and use. If needed, additional documentation can be created and guidance regarding the use of the framework can be given. Future work considering the test framework and the software release testing process include:

- Maintain the framework. Proper maintenance for the framework needs to be performed, e.g. the software tools updated (if required) and bugs corrected. If needed, additional features have to be implemented.
- Write more tests and maintain existing ones. The J1939 communication tests test only one feature of the PowerMASTER's software releases. The writing and developing of test scripts needs to continue, so that a better test coverage can be achieved. This includes developing scripts to use in all test environments described in chapter 5.1. Test scripts also need to be developed and added to the framework for other product families, e.g. PowerBOOST and PowerCOMBO. Existing test scripts need to be maintained.
- Make the test execution more modular. The addition of new tests and the creation of a test job for a software variant is currently fairly easy. However the possibility to execute single hand picked test cases could be useful if some abnormal conditions are discovered in the test environment during the test run. For example some test cases failed simply because a cable was disconnected, etc. This way the whole test set wouldn't be required to be executed again, only the test cases that were affected by the abnormal condition. Although if the execution time of the test scripts is fairly low, this feature might lose it's usefulness.
- Increase the amount of automation. Some of the tests in the framework (e.g. the J1939 communication tests) could possibly be fully automated, i.e. run the test jobs in Jenkins automatically. One possibility would be to poll the release directory in Visedo's network drive and run the test job when a new release is detected. Jenkins allows to trigger the builds with a script, which could be utilized here. Another way is to have Jenkins run the test job as soon as someone commits new code for the release. This can be achieved using the poll source code management trigger of Jenkins. For some tests, monitoring of the test runs is required. Examples of these would be control tests, that require the inverter to drive an electric motor. Full automation for these type of tests isn't possible.

- Create a database for the test reports. The test reports are currently stored in Jenkins' workspace. However Jenkins isn't designed for documentation organizing purposes. A database to keep, display, sort and filter the reports could be created. This could be helpful e.g. for searching specific reports. A lot of ready made software tools exist for creating databases.
- Integrate the test reports with Visedo's quality system. One of the points of the PDF test reports was the possibility to add them into Visedo's quality system. This integration process is yet to be started and the reports will need some modifying to fit the company standards. The test case descriptions for example are currently only preliminary, e.g. information about the test fail or success criteria need to be added to the reports.

7 CONCLUSIONS

Developing a test framework and automated test scripts has increased the reliability in software quality and made test execution more efficient at Visedo. In the case of power converters that are part of an electrical drive system, software faults manifested after the commissioning of the device can be very costly. This emphasizes the need for proper testing even further.

Before the framework only manual ad hoc type testing was performed for the software releases, which was often very time consuming and inefficient. This easily resulted in the fact, that software releases were not properly tested because of tight delivery deadlines. Utilizing test automation and a test framework has helped in solving this issue. A lot of ready made testing tools exist for developing automated test systems, for example the Minitest framework and Jenkins presented in this thesis. When properly configured and utilized these tools can be seen as major help in implementing test automation. In regression testing, one of the main purposes of the test framework, reproducing the test runs is important. The tests need to be executed the same way for all releases. A framework that runs automated test scripts is helpful in achieving this, as the same set of test scripts can be rerun repeatedly. Regression tests are also by nature well suited for automation as they are run repeatedly for all new releases.

Proper test reporting was found to be an important part in testing. A clear and detailed test report provides a convenient way to track the status of the software releases. A test report should not just have the test results, but also information on what was tested and how. However the reports still need to be integrated with Visedo's quality system and modified to satisfy the company standards to gain the most benefit from them. Nonetheless the created reporting system serves as a good base for the integration process.

The test framework has increased test efficiency, but implementing and maintaining the framework isn't free. It requires resources, the test scripts and the framework have to be implemented and maintained by developers or dedicated testers. Therefore the usefulness of test frameworks and test automation in general is always case specific and financial aspects need to also be taken into consideration, even though they were not discussed in the context of this thesis. It should also be noted that not all tests can or should be automated. Manual tests are still required to support test automation. The analyzing of the test reports has to also be performed manually.

The framework developed in this thesis only runs regression tests (can also be seen as integration tests), but testing is important in all development phases of a software product. Testing in other development phases needs to also continue, and testing processes in all phases need to be developed further. Testing in general needs to be an integral part of the development process now and in the future at Visedo. Furthermore testing is not just the use of testing tools, but a development practice. It requires motivation and the right kind of attitude from software developers and testers alike.

Appendices

A An overview of inverters (DC/AC converters)

An inverter also known as a DC/AC converter, is a switch mode circuit that converts a DC-voltage input to an AC-voltage output to drive AC loads and motors or connect to the electric grid. The frequency and voltage of the AC-output can be controlled and thus used to control, for example, the torque and speed of an AC-motor. The DC-input can be supplied from e.g. batteries, solar or fuel cells, or from a DC-link, that is fed from an AC-source. [40] [41]

For three-phase inverters the operation is based on at least six semiconductor switches, some examples being the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) and the Insulated-Gate Bipolar Transistor (IGBT). [40] Figure 26 shows a three-phase inverter implemented using six IGBTs.

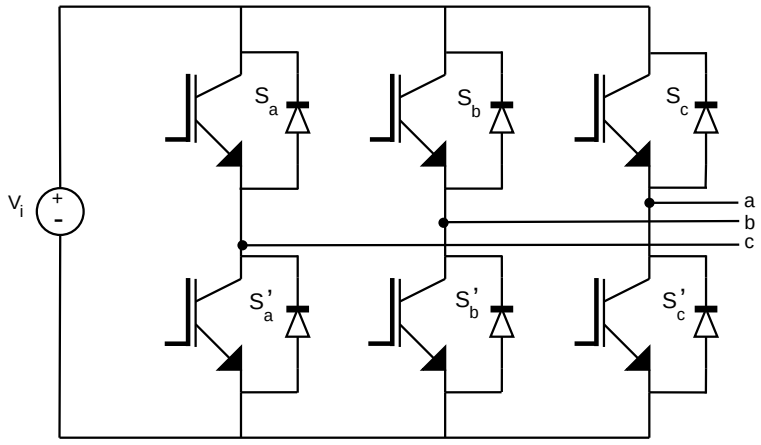


Figure 26: A simplified schematic of a three-phase inverter implemented using six IGBTs (S_a - S_c and S'_a - S'_c) as power switches. V_i is the DC-voltage input and a, b and c make the three-phase output whose voltage and frequency can be controlled. Own illustration, based on [40].

Inverters are generally classified as single-phase or three-phase inverters. Inverters can also be classified as half-bridge or full-bridge inverters. Inverters can also be divided to voltage source inverters (VSI) or current source inverters (CSI) depending on whether the input is DC-voltage or DC-current. [40] [41]

B The code for the rakefile

```
require 'rake/testtask'
require 'ci/reporter/rake/minitest'

Rake::TestTask.new do |t|
  t.test_files = FileList['../../tests/products/PowerMASTER/
      TestCaseJ1939StandardV2inputs.rb',
                          '../../tests/products/PowerMASTER/
                              TestCaseJ1939StandardV2outputs.rb']
  #  t.verbose = true
end

task :test => 'ci:setup:minitest'

desc "Run tests"
task :default => :test
```

C The test suites

C.1 Test suite: TestJ1939StandardV2inputs

```
require 'rubygems'
require 'vise_script'
require 'vise_script_utils'
gem 'minitest'
require 'minitest/autorun'
require 'csv'

class TestJ1939StandardV2inputs < Minitest::Test

  include ViseScript
  include Utils::ScriptUtils

  # Signal groups
  StartStopDo = [10100,1]
  SpeedRef = [10100,4]
  SpeedLimMin = [10202,4]
  SpeedLimMax = [10202,5]
  AppControlModeCs = [55000,1]
  UseNwSpeedLimsCs = [55000,3]
  UseNwTorqueLimsCs = [55000,4]
  UseNwPowerLimsCs = [55000,5]
  UseNwOVCUVCLimsCs = [55000,6]
  AppControlMode = [50010,10]
  SpeedLimsSource = [50010,20]
  TorqueLimsSource = [50010,21]
  PowerLimsSource = [50010,22]
  OvcLimsSource = [50010,26]
  UvcLimsSource = [50010,27]
  MotorNomSpeedRpm = [10003,1]

  AppId = [1001, 20]

  FloatDeltaLimit = 0.001

  @@before_tests_done = false

  def before_tests
    return  if @@before_tests_done
    host = ENV['VISELAB_SERVER_IP'] || '127.0.0.1'
    port = (ENV['VISELAB_SERVER_PORT'] || 3333).to_i
    @@client = ClientFactory.build(host, port)
    pdp_drive = @@client.get_device_by_name(ENV['PDP_DEVICE_NAME'] || 'Visedo Generic
        PDP Serial')
    pdp_drive.connect.wait
    app_id = pdp_drive.get_parameter(AppId)
    if app_id == 2 || app_id == 3 # Codesys application
      pdp_drive.set_parameter(UseNwSpeedLimsCs, true)
      pdp_drive.set_parameter(UseNwTorqueLimsCs, true)
      pdp_drive.set_parameter(UseNwPowerLimsCs, true)
      pdp_drive.set_parameter(UseNwOVCUVCLimsCs, true)
      pdp_drive.set_parameter(AppControlModeCs, 255).wait # Set base application
          control mode to network selected
    else
      # Set limit sources to network
      pdp_drive.set_parameter(SpeedLimsSource, 1)
      pdp_drive.set_parameter(TorqueLimsSource, 1)
      pdp_drive.set_parameter(PowerLimsSource, 1)
      pdp_drive.set_parameter(OvcLimsSource, 1)
      pdp_drive.set_parameter(UvcLimsSource, 1)
      pdp_drive.set_parameter(AppControlMode, 255).wait # Set base application control
          mode to network selected
    end

    @@before_tests_done = true
  end
```

```
63
64    def setup
65      before_tests
66      @j1939_drive = @@client.get_device_by_name(ENV['J1939_DEVICE_NAME'] || 'Visedo
            Standard V2 J1939')
67      @j1939_drive.connect.wait
68      @pdp_drive = @@client.get_device_by_name(ENV['PDP_DEVICE_NAME'] || 'Visedo Generic
            PDP Serial')
69      @pdp_drive.connect.wait
70      @pdp_drive.reset_faults.wait
71    end
72
73    def teardown
74      @j1939_drive.stop
75      # @pdp_drive.set_dataobject(StartStopDo, false).wait
76      @pdp_drive.disconnect
77      #sleep 2
78    end
79
80    # Help method for testing references (ramp)
81    def ramp_ref(ramp_min, ramp_max, steps, ref_type, unit, nom_value, ref_index)
82      reference_set = [] # Array for saving set reference to a csv file
83      reference_read = [] # Array for saving read reference to a csv file
84      reference_set[0] = "Reference set with J1939 " + "[" + unit + "]"
85      reference_read[0] = "Reference read with PDP " + "[" + unit + "]"
86      ref_read = nil
87      step = (ramp_max - ramp_min) / steps.to_f
88      @j1939_drive.ref_type = (ref_type + "_ref").upcase.to_sym
89      (steps + 1).times do |i|
90        ref_set = ramp_min + step * i
91        @j1939_drive.send(ref_type + "_ref=" , ref_set)
92        wait_until_in_delta(ref_set) { ref_read = @pdp_drive.get_dataobject(ref_index) *
              nom_value }
93        assert_in_delta(ref_set, ref_read, FloatDeltaLimit, "Failed setting #{ref_type}
              ref to #{ref_set} #{unit}.")
94        reference_set[i+1] = ref_set.round(1)
95        reference_read[i+1] = ref_read.round(1)
96      end
97      # Create directories for test data and save set and read references to csv files
            for report generation (2 tables and 2 plots)
98      if not File.directory?("test_data")
99        Dir.mkdir 'test_data'
100     end
101     Dir.chdir("test_data")
102     if not File.directory?("TestJ1939StandardV2inputs")
103       Dir.mkdir 'TestJ1939StandardV2inputs'
104     end
105     Dir.chdir("TestJ1939StandardV2inputs")
106     if ramp_max < 0
107       # Table for negative references
108       CSV.open("data_table_test_" + ref_type + "_references_negative.csv", "wb") do |
              csv|
109         csv << ["Data from test case: test_" + ref_type + "_ref. Contains the set and
                read negative references rounded to one decimal."] # table title
110         csv << reference_set # 1. column
111         csv << reference_read # 2. column
112       end
113       # Plot for negative references
114       CSV.open("data_plot_test_" + ref_type + "_references_negative.csv", "wb") do |
              csv|
115         csv << ["test_speed_ref_neg"] # plot name
116         csv << ["A plot of set and read negative speed references from test case: test
                speed ref."] # plot title
117         csv << ["Reference set with J1939 [rpm]"] # x-label
118         csv << ["Reference read with PDP [rpm]"] # y-label
119         csv << ["Speed references in rpm"] # curve title
120         reference_set.delete_at(0)
121         reference_read.delete_at(0)
122         csv << reference_set # x data
```

```
123           csv << reference_read # y data
124         end
125       else
126         # Table for positive references
127         CSV.open("data_table_test_" + ref_type + "_references_positive.csv", "wb") do |
              csv|
128           csv << ["Data from test case: test_" + ref_type + "_ref. Contains the set and
                read positive references rounded to one decimal."]
129           csv << reference_set
130           csv << reference_read
131         end
132         # Plot for positive references
133         CSV.open("data_plot_test_" + ref_type + "_references_positive.csv", "wb") do |
              csv|
134           csv << ["test_speed_ref_pos"]
135           csv << ["A plot of set and read positive speed references from test case: test
                speed ref."]
136           csv << ["Reference set with J1939 [rpm]"]
137           csv << ["Reference read with PDP [rpm]"]
138           csv << ["Speed references in rpm"]
139           reference_set.delete_at(0)
140           reference_read.delete_at(0)
141           csv << reference_set
142           csv << reference_read
143         end
144       end
145       Dir.chdir File.expand_path("..")
146       Dir.chdir File.expand_path("..")
147     end
148
149     # Help method for testing limits
150     def lim_test (lim_min, lim_max, lim_type, unit, lim_min_ref_index, lim_max_ref_index
          , nom_value)
151       lim_max_read = 0 # higher limit read through pdp
152       lim_min_read = 0 # lower limit read through pdp
153
154       @pdp_drive.set_dataobject(lim_max_ref_index, 0).wait
155       @pdp_drive.set_dataobject(lim_min_ref_index, 0).wait
156
157       @j1939_drive.send(lim_type + "_limit_higher=" , lim_max)
158       @j1939_drive.send(lim_type + "_limit_lower=" , lim_min)
159
160       wait_until_in_delta(lim_max) do
161         lim_max_read = @pdp_drive.get_dataobject(lim_max_ref_index).wait
162       end
163       wait_until_in_delta(lim_min) do
164         lim_min_read = @pdp_drive.get_dataobject(lim_min_ref_index).wait
165       end
166
167       assert_in_delta(lim_max, lim_max_read * nom_value, FloatDeltaLimit, "Failed
            setting higher #{lim_type} limit to #{lim_max} #{unit}.")
168       assert_in_delta(lim_min, lim_min_read * nom_value, FloatDeltaLimit, "Failed
            setting lower #{lim_type} limit to #{lim_min} #{unit}.")
169     end
170
171     # --------------------------------TESTS---------------------------------
172
173     ## Testsuite description
174
175     def test_description
176       puts "test_description/ This testsuite tests the input messages of the J1939
            Standard V2 communication. /test_description"
177     end
178
179     ## Tests for run and stop command
180
181     def test_start
182       puts "info_test_start/ This test case tests the setting of the run command. /
            info_test_start"
```

```
183       @j1939_drive.stop
184       assert wait_until{ !@pdp_drive.get_dataobject(StartStopDo).wait }, "stop not
              active"
185       @j1939_drive.start
186       assert wait_until{ @pdp_drive.get_dataobject(StartStopDo).wait }, "start not
              active"
187     end
188
189     def test_stop
190       puts "info_test_stop/ This test case tests the setting of the stop command. /
              info_test_stop"
191       @j1939_drive.start
192       assert wait_until{ @pdp_drive.get_dataobject(StartStopDo).wait }, "start not
              active"
193       @j1939_drive.stop
194       assert wait_until{ !@pdp_drive.get_dataobject(StartStopDo).wait }, "stop not
              active"
195     end
196
197     ## Test for setting speed references (in rpm)
198
199     def test_speed_ref
200       puts "info_test_speed_ref/ This test case tests the setting of speed references
              with a ramp input. Test pass requires the set and
201       read references to be within 0.001 units (rpm) from each other. /
              info_test_speed_ref"
202       nom_speed = @pdp_drive.get_dataobject(MotorNomSpeedRpm).wait
203       ramp_ref(0,  16000, 5, "speed", "rpm", nom_speed, SpeedRef)
204       ramp_ref(0, -16000, 5, "speed", "rpm", nom_speed, SpeedRef)
205     end
206
207     ## Test for setting speed limits (in rpm)
208
209     def test_speed_limits
210       puts "info_test_speed_limits/ This test case tests the setting of speed limits
              with one upper and one lower limit. /info_test_speed_limits"
211       lim_test(-1000, 1000, "speed", "rpm", SpeedLimMin, SpeedLimMax, 1)
212     end
213   end
```

C.2 Test suite: TestJ1939StandardV2outputs

```
1   require 'rubygems'
2   require 'vise_script'
3   require 'vise_script_utils'
4   gem 'minitest'
5   require 'minitest/autorun'
6
7   class TestJ1939StandardV2outputs < Minitest::Test
8
9     include ViseScript
10    include Utils::ScriptUtils
11
12    # Signal groups
13    SpeedLimsSource = [50010,20]
14    TorqueLimsSource = [50010,21]
15    PowerLimsSource = [50010,22]
16    OvcLimsSource = [50010,26]
17    UvcLimsSource = [50010,27]
18
19    AppId = [1001, 20]
20
21    NetworkSpeedActRpm = [2000,11]
22    NetworkTorqueActNm = [2000,12]
23    NetworkPowerActkW = [2000,22]
24    NetworkVoltageAct = [2000,10]
25
26    FloatDeltaLimit = 0.001
27
```

```
28    @@before_tests_done = false
29
30    def before_tests
31      return  if @@before_tests_done
32      host = ENV['VISELAB_SERVER_IP'] || '127.0.0.1'
33      port = (ENV['VISELAB_SERVER_PORT'] || 3333).to_i
34      @@client = ClientFactory.build(host, port)
35      pdp_drive = @@client.get_device_by_name(ENV['PDP_DEVICE_NAME'] || 'Visedo Generic
            PDP Serial')
36      pdp_drive.connect.wait
37
38      # Set limit sources to network
39      pdp_drive.set_parameter(SpeedLimsSource, 1)
40      pdp_drive.set_parameter(TorqueLimsSource, 1)
41      pdp_drive.set_parameter(PowerLimsSource, 1)
42      pdp_drive.set_parameter(OvcLimsSource, 1)
43      pdp_drive.set_parameter(UvcLimsSource, 1)
44      pdp_drive.set_parameter(AppId, 0) # Disable the application
45
46      @@teardown_counter = 0 # Counter for checking how many tests are completed
47      @@number_of_tests = methods.count{|x| x.to_s.start_with?("test")} # Count the
            total number of tests
48
49      @@before_tests_done = true
50    end
51
52    def setup
53      before_tests
54      @j1939_drive = @@client.get_device_by_name(ENV['J1939_DEVICE_NAME'] || 'Visedo
            Standard V2 J1939')
55      @j1939_drive.connect.wait
56      @pdp_drive = @@client.get_device_by_name(ENV['PDP_DEVICE_NAME'] || 'Visedo Generic
            PDP Serial')
57      @pdp_drive.connect.wait
58      @pdp_drive.reset_faults.wait
59    end
60
61    def teardown
62      @@teardown_counter = @@teardown_counter + 1
63      @j1939_drive.stop
64      # @pdp_drive.set_dataobject(StartStopDo, false).wait
65      if @@teardown_counter == @@number_of_tests # All tests completed
66        @pdp_drive.set_parameter(AppId, 2) # Enable the application
67      end
68      @pdp_drive.disconnect
69      #sleep 2
70    end
71
72    # Help method for testing measurements
73    def meas_test(meas_type, min_value, max_value, steps, network_meas_ref_index, unit)
74      step = (max_value - min_value) / steps
75      (steps + 1).times do |i|
76        value_set = min_value + step * i
77        @pdp_drive.set_dataobject(network_meas_ref_index, value_set)
78        case meas_type
79        when "speed"
80          value_meas = @j1939_drive.speed
81        when "torque"
82          value_meas = @j1939_drive.torque
83        when "power"
84          value_meas = @j1939_drive.power
85        when "voltage"
86          value_meas = @j1939_drive.voltage
87        end
88        puts "Measured #{meas_type}: #{value_meas} #{unit}"
89
90        assert_in_delta(value_meas, value_set, FloatDeltaLimit, "Measured #{meas_type}
              doesnt correspond to set value of #{value_set} #{unit}.")
91      end
```

```
92    end
93
94    # --------------------------------TESTS--------------------------------
95
96    ## Testsuite description
97
98    def test_description
99      puts "test_description/ This testsuite tests the output messages of the J1939
          Standard V2 communication. /test_description"
100   end
101
102   ## Tests for speed, torque, power and voltage measurements (application must be
        disabled for these to work)
103
104   def test_speed_meas
105     puts "info_test_speed_meas/ This test case tests the reading of the speed
          measurement. /info_test_speed_meas"
106     meas_test("speed", 0, 1000, 10, NetworkSpeedActRpm, "rpm")
107   end
108
109   def test_torque_meas
110     puts "info_test_torque_meas/ This test case tests the reading of the torque
          measurement. /info_test_torque_meas"
111     meas_test("torque", 0, 400, 10, NetworkTorqueActNm, "Nm")
112   end
113
114   def test_power_meas
115     puts "info_test_power_meas/ This test case tests the reading of the power
          measurement. /info_test_power_meas"
116     meas_test("power", 0, 4000, 10, NetworkPowerActkW, "kW")
117   end
118
119   def test_voltage_meas
120     puts "info_test_voltage_meas/ This test case tests the reading of the voltage
          measurement. /info_test_voltage_meas"
121     meas_test("voltage", 0, 500, 10, NetworkVoltageAct, "V")
122   end
123 end
```

D An example XML report from a test run

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <testsuite name="TestJ1939StandardV2inputs" tests="4" time="4.138827" failures="0"
       errors="0" skipped="0" assertions="18" timestamp="2016-08-12T14:11:03+03:00">
3    <testcase name="test_speed_ref" time="2.0858982289937558" assertions="12">
4    </testcase>
5    <testcase name="test_speed_limits" time="0.4342902459902689" assertions="2">
6    </testcase>
7    <testcase name="test_stop" time="0.4195803639886435" assertions="2">
8    </testcase>
9    <testcase name="test_start" time="0.4206930179934716" assertions="2">
10   </testcase>
11 </testsuite>
```

E An example PDF report from a test run

Presented in the next page.

VISEDO

SOFTWARE TEST REPORT

CONFIDENTIAL

Visedo Oy
Lentokentantie 44
FIN-53850 Lappeenranta
Finland

Tel: +358-50-3400 935
Email: contactus@visedo.fi
Web: www.visedo.com

Test suite: TestJ1939StandardV2inputs

INFORMATION

Tests run by: Juho Matikainen

Tests run at: 2016-09-15 at 15:43:16+03:00

Tests run for: Z:\Visedo\visedo_sw_releases\pike\releases/6.8.34/powermaster-hwrevd-6.8.34-1724be1-standard-v2-j1939-fw-ba.vfi

TEST DESCRIPTION

This testsuite tests the input messages of the J1939 Standard V2 communication.

TESTCASES (in order of execution)

Testcase 1: test_stop

This test case tests the setting of the stop command.

Testcase 2: test_start

This test case tests the setting of the run command.

Testcase 3: test_speed_limits

This test case tests the setting of speed limits with one upper and one lower limit.

Testcase 4: test_speed_ref

This test case tests the setting of speed references with a ramp input. Test pass requires the set and read references to be within 0.001 units (rpm) from each other.

TOTAL RESULTS

Runtime: 3.790379 s

Number of assertions: 18

Number of failures: 0

Number of errors: 0

Number of skipped: 0

FAILURES

No failures.

Test data

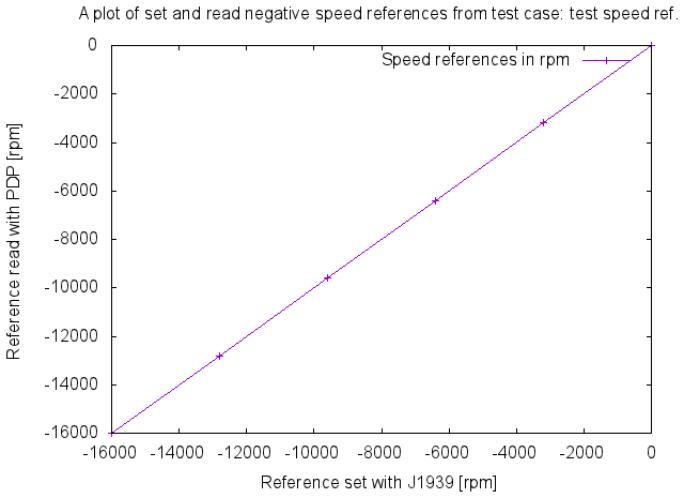
Table 1. Data from test case: test_speed_ref. Contains the set and read negative references rounded to one decimal.

Reference set with J1939 [rpm]	Reference read with PDP [rpm]
0.0	0.0
-3200.0	-3200.0
-6400.0	-6400.0
-9600.0	-9600.0
-12800.0	-12800.0
-16000.0	-16000.0

Table 2. Data from test case: test_speed_ref. Contains the set and read positive references rounded to one decimal.

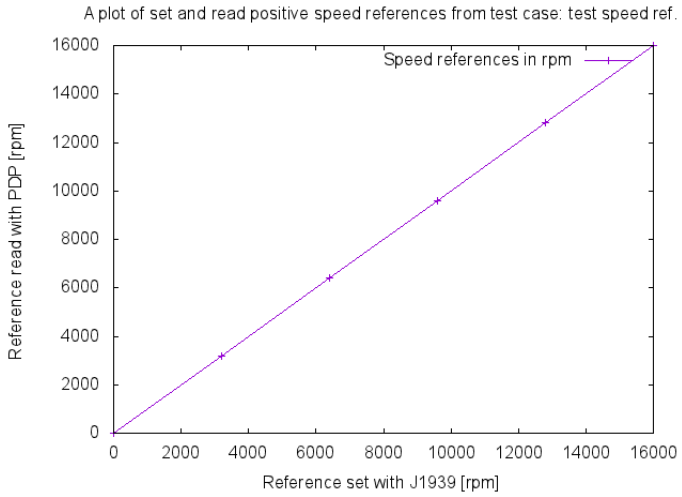
Reference set with J1939 [rpm]	Reference read with PDP [rpm]
0.0	0.0
3200.0	3200.0
6400.0	6400.0
9600.0	9600.0
12800.0	12800.0
16000.0	16000.0

Plot 1.



Visedo Oy
Lentokentantie 44
FIN-53850 Lappeenranta
Finland
Tel: +358-50-3400 935
Email: contactus@visedo.fi
Web: www.visedo.com

Plot 2.



Visedo Oy
Lentokentantie 44
FIN-53850 Lappeenranta
Finland
Tel: +358-50-3400 935
Email: contactus@visedo.fi
Web: www.visedo.com

References

[1] R. W. De Doncker. Modern electrical drives: Design and future trends. In *Power Electronics and Motion Control Conference, 2006. IPEMC 2006. CES/IEEE 5th International*, volume 1, pages 1–8, Aug 2006.

[2] International Energy Agency. Global EV Outlook, 2015. [online]. Available: `http://www.iea.org/evi/Global-EV-Outlook-2015-Update_2page.pdf`. [Accessed: May 30, 2016].

[3] Piet Cordemans, Sille Van Landschoot, Jeroen Boydens, and Eric Steegmans. *Embedded and Real Time System Development: A Software Engineering Perspective: Concepts, Methods and Principles*, chapter Test-Driven Development as a Reliable Embedded Software Engineering Practice, pages 91–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[4] Visedo Ltd. Visedo PowerMASTER M-frame datasheet, 2016. [online]. Available: `http://www.visedo.com/sites/default/files/downloads/visedo_powermaster_m-frame_datasheet_r0516.pdf`. [Accessed: May 24, 2016].

[5] Patrícia Machado, Auri Vincenzi, and José Carlos Maldonado. *Software Testing: An Overview*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Gerard O'Regan. *Software Testing Capability Maturity Model Integration*, pages 119–134. Springer International Publishing, Cham, 2014.

[7] H. m. Qian and C. Zheng. A embedded software testing process model. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1–5, Dec 2009.

[8] James W. Moore. *Knowledge Area: Software Testing*, pages 123–136. Wiley-IEEE Press, 2006.

[9] Tim A. Majchrzak. *Software Testing*, pages 11–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] Gerald D. Everett and Raymond McLeod. *Functional Testing*, pages 99–121. Wiley-IEEE Press, 2007.

[11] Jeff Tian. *Testing: Concepts, Issues, and Techniques*, pages 67–84. Wiley-IEEE Press, 2005.

[12] Lingfeng Wang and Kay CHen Tan. *Software Testing*, pages 69–87. Wiley-IEEE Press, 2006.

[13] J. Grenning. Applying test driven development to embedded software. *IEEE Instrumentation Measurement Magazine*, 10(6):20–25, December 2007.

[14] Haeng Kon Kim. *Aspect Oriented Testing Frameworks for Embedded Software*, pages 75–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[15] Gary McLean Hall. *Pro WPF and Silverlight MVVM: Effective Application Development with Model-View-ViewModel*, chapter Unit Testing, pages 145–162. Apress, Berkeley, CA, 2010.

[16] Github Inc. Github, Minitest home repository, 2016. [online]. Available: `https://github.com/seattlerb/minitest`. [Accessed: June 2, 2016].

[17] Seattlerb.org. Seattlerb.org, Minitest documentation, assertions, 2016. [online]. Available: `http://docs.seattlerb.org/minitest/Minitest/Assertions.html`. [Accessed: June 2, 2016].

[18] A. N. Seshu Kumar and S. Vasavi. *Effective Unit Testing Framework for Automation of Windows Applications*, pages 813–822. Springer India, New Delhi, 2012.

[19] Derk-Jan De Grood. *Test Automation*, pages 277–292. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[20] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist. Technical debt in test automation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 887–892, April 2012.

[21] Karl Henrik Johansson, Martin Törngren, and Lars Nielsen. *Handbook of Networked and Embedded Control Systems*, chapter Vehicle Applications of Controller Area Network, pages 741–765. Birkhäuser Boston, Boston, MA, 2005.

[22] H. Chen and J. Tian. Research on the controller area network. In *Networking and Digital Society, 2009. ICNDS '09. International Conference on*, volume 2, pages 251–254, May 2009.

[23] Robert Bosch GmbH. *Bosch Automotive Electrics and Automotive Electronics: Systems and Components, Networking and Hybrid Drive*, chapter Bus systems, pages 92–151. Springer Fachmedien Wiesbaden, Wiesbaden, 2014.

[24] Vector Informatik GmbH. Vector Informatik: Introduction to J1939, application note AN-ION-1-3100, version 1.1, 2010. [online]. Available: `http://vector.com/portal/medien/cmc/application_notes/AN-ION-1-3100_Introduction_to_J1939.pdf`. [Accessed: May 25, 2016].

[25] Kvaser. Kvaser, J1939 introduction, 2016. [online]. Available: `https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/`. [Accessed: September 2, 2016].

[26] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Higher-Level Protocols*, pages 181–214. Springer New York, New York, NY, 2012.

[27] Gangolf Feiter, Lars-Berno Fredriksson, Karsten Hoffmeister, Joakim Pauli, and Holger Zeltwanger. *Higher Level Protocols*, pages 173–254. Springer London, London, 2013.

[28] Wilfried Voss. *A Comprehensible Guide to J1939*. Greenfield: Copperhill Media Corporation, 2008.

[29] Neil Smyth. *Ruby Essentials*, chapter What is Ruby? Payload Media, 2012. [online]. Available: `http://www.techotopia.com/index.php/Ruby_Essentials`. [Accessed: July 27, 2016].

[30] Wikibooks. The Ruby Programming Wikibook, Ruby programming/overview, 2015. [online]. Available: `https://en.wikibooks.org/wiki/Ruby_Programming/Overview`. [Accessed: July 27, 2016].

[31] Github Inc. Github, Rake home repository, 2016. [online]. Available: `https://github.com/ruby/rake`. [Accessed: June 21, 2016].

[32] Github Inc. Github, Ci-reporter: Minitest, home repository, 2016. [online]. Available: `https://github.com/ci-reporter/ci_reporter_minitest`. [Accessed: June 22, 2016].

[33] Github Inc. Github, Ci-reporter, home repository, 2016. [online]. Available: `https://github.com/ci-reporter/ci_reporter`. [Accessed: June 23, 2016].

[34] Github Inc. Github, Prawn home repository, 2016. [online]. Available: `https://github.com/prawnpdf/prawn`. [Accessed: June 13, 2016].

[35] Github Inc. Github, Nokogiri home repository, 2016. [online]. Available: `https://github.com/sparklemotion/nokogiri`. [Accessed: June 13, 2016].

[36] Gnuplot. Gnuplot homepage, 2016. [online]. Available: `www.gnuplot.info`. [Accessed: June 13, 2016].

[37] Github Inc. Github, Ruby gnuplot home repository, 2016. [online]. Available: `https://github.com/rdp/ruby_gnuplot`. [Accessed: June 13, 2016].

[38] Jenkins. Jenkins home page, 2016. [online]. Available: `https://jenkins.io`. [Accessed: June 27, 2016].

[39] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE '09.*, pages 369–374, Aug 2009.

[40] Edison R. C. da Silva and Malik E. Elbuluk. *Power Electronics for Renewable and Distributed Energy Systems: A Sourcebook of Topologies, Control and Integration*, chapter Fundamentals of Power Electronics, pages 7–59. Springer London, London, 2013.

[41] Ned Mohan, Tore M. Undeland, and Willian P. Robbins. *Power Electronics: Converters, Applications, and Design, Media Enhanced Third Edition*. John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, 2003.