Expertise
and insight
for the future

Roman Chlada

Software Test Automation: Safety System of a High Speed Train

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Program of Electronics

Bachelor's Thesis

29. January 2022

Author	Roman Chlada
Title	Software Test Automation: Safety System of a High Speed Train
Number of Pages	46 pages
Date	29. January 2022
Degree	Bachelor of Engineering
Degree Program	Electronics
Professional Major	Electronics
Instructors	Iiro Koskinen, Software Engineer Janne Mäntykoski, Senior Lecturer

The purpose of this project was automating the testing of the software of an embedded system designed for the safety control of a high speed train. In safety systems, 100% of the customer requirements must be implemented and tested.

Tools for manual testing were already existing. The Robot Framework was given as the platform to run the tests. The tasks were to adapt the Robot Framework to make testing possible and efficient, to implement tests for hundreds of customer requirements and to solve challenges, which occurred during the process.

At the end of this project, over 1000 test cases are running during more than 12 hours every night. Only minor details might need to be fixed in the software and the testing.

Keywords	Embedded Systems, Software Testing, Test Automation

Contents

List of Abbreviations

Bogie	Wheelset of the locomotive
CLI	Command Line Interface
DAC	Digital to Analog Converter
GUI	Graphical User Interfaces
GW	Gateway
HAT	Hardware Attached on Top (of the Raspberry Pi)
HSA	High Speed Analog Input Module
ICD	Interface Control Document
LED	Light Emitting Diode
MIO	Modular Input/Output module
MIO_S	MIO with Safety Integrity Level 2
MIO_Sp	Primary MIO_S
MIO_Ss	Secondary MIO_S
MVB	Multifunction Vehicle Bus
PCB	Printed Circuit Board
PTI	Pt100/Pt1000 Temperature Sensor Input Module
RIOM	Remote IO Module
RIOM_S	RIOM with Safety Integrity Level 2

RIOM_Sp	Primary RIOM_S
RIOM_Ss	Secondary RIOM_S
RPI	Raspberry Pi
SDTv2	Safe Data Transmission, version 2
SSH	Secure Shell
SMI	Safety Message Identifier
SPI	Serial Peripheral Interface
STCMS	Safe Train Control and Management System
SUT	System under Test
TCN	Train Communication Network
TCMS	Train Control and Management System
TRDP	Train Real Time Data Protocol
VCU	Vehicle Control Unit
VCU_S	VCU with Safety Integrity Level 2
VCU_Sp	Primary VCU_S
VCU_Ss	Secondary VCU_S
WTB	Wire Train Bus

Definitions

Safety/safe: In this paper the term safety/safe is used to denote the safety of the functionality. The safety of the functioning is ensured by enhancing reliability and reducing the failure rate.

Security/secure: Unlike safety, the term security/secure refers in this paper entirely to the accessibility of a system. The access of a secure system is made as difficult as possible for unauthorized people.

1 Introduction

Establishing test automation requires not only a notable amount of expertise, but also a rare resource in modern industry: time. This increases the temptation to postpone building up test automation and to continue manual testing – despite of its known disadvantages.

Manual testing is error prone and finally more time consuming than automated testing. Manual testing can be started easily, while it might take weeks or months until the first automated test is running. But once automated testing is established, the test runs much faster than manual testing can ever be performed.

This paper shows the establishing of test automation for an embedded system monitoring safety relevant data in a train. The following chapter, chapter 2, provides a generic overview over safety systems in trains, starting with the definition of safety systems, then showing the hardware used typically and its functionality. In chapter 3 two examples for the architecture of safety systems in trains are presented.

The project of test automation is described in chapters 4, 5 and 6. Chapter 4 describes the system under test. In chapter 5, the tools are introduced that were designed earlier for manual testing. Chapter 6 shows the process of establishing automated testing.

2 Safety Systems in Trains

2.1 Safety Systems and Safety Software Development

Safety Systems are used in different industries like car industry, aviation, or space industries. Also in trains safety systems are needed. Different contexts require different levels of safety. Those levels are defined by standards, they are called: Safety Integrity Levels (SIL). No system can work without failures, but failures can be systematically reduced. The more is at stake the less failures can be accepted. The four Safety Integrity Levels are distinguished by the expectable failure rate – as can be seen in table 1. In train industry SIL-2 is widely used.

Table 1. SIL table: Probability of failures per hour and risk reduction factor [1].

SIL	Probability of dangerous failure per hour	Risk Reduction Factor
1	10^{-5} - 10^{-6}	100.000 - 1.000.000
2	10^{-6} - 10^{-7}	1.000.000 - 10.000.000
3	10^{-7} - 10^{-8}	10.000.000 - 100.000.000
4	10^{-8} - 10^{-9}	100.000.000 - 1.000.000.000

The train industry specific standard EN 50128 [2] describes the software development. It uses the V-model to describe the development process – see figure 1. The 'V' starts with an overview over the needed features. It continues with breaking down all that is required to increasingly detailed levels, until implementation is possible. The writing of the source code is at the corner of the 'V'. After that the system is built up, tested, and validated. Practically, no software development ever worked by following the 'V' only once. Customer requirements might cover only parts of the entire system in the beginning, they are clarified or even changed later. Testing reveals bugs and maybe also flaws in the software design. Therefore, at least the steps from the Software Component Design Phase to the Software Integration Phase are iterated over and over again; often even the Software Requirement Phase has to be re-opened. For integration testing, that means, that it is repeated many times. This is, where test automation helps tremendously.

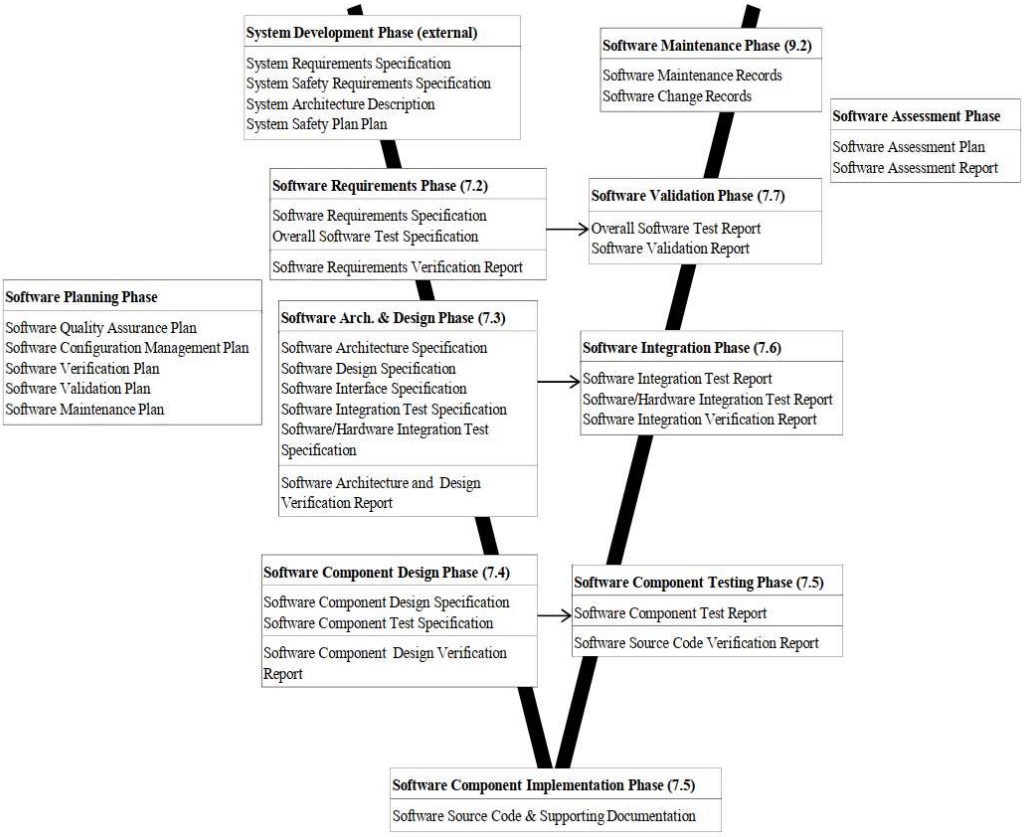


Figure 1. Illustrative Development Lifecycle – V-model [2, 23].

2.2 Safe Train Control and Management Systems

A Safe Train Control and Management System (STCMS) can be designed in many ways, depending on the needs of the customer and the type of the train: A high speed train travelling with more than 300 km/h will need more safety related monitoring than a metro train that reaches at maximum 80 km/h.

STCMS controls so-called safety functions. The standard EN-50128 defines a safety function as “a function that implements a part or whole of a safety requirement” [2, 13]. A safety requirement is the necessity to mitigate a certain safety hazard that cannot be tolerated. All means to provide this mitigation can be understood as one safety function.

For example, one of the best known safety functions in trains is the passenger alarm. The *safety requirement* is that passengers must have the possibility to stop the train. It

is necessary to mitigate the following safety hazard: An event happening in the passenger area might require the stopping of the train, but it cannot be noticed by the train driver or train personnel.

The *safety function* implementing this requirement is the passenger alarm handle in every passenger cabin and everything that connects this handle and the brakes of the train. Among those things connecting the passenger alarm handle and the brakes is STCMS. It reads as a digital input the state of the passenger alarm handle and sets the outputs activating the brake if the safety function is not overridden.

Examples of safety functions used in this project can be found in chapter 4.2. The hardware used typically for performing different safety functions is described in the following chapters: 2.3, 2.4 and 2.5

2.3 Communication

Aside of train specific protocols that are used on commonly used busses (e.g. TRDP on ethernet), there are two types of busses, which are designed and certified especially for trains. They are the Multifunctional Vehicle Bus (MVB) and the Wire Train Bus (WTB). MVB and WTB together are one valid way to form the so-called Train Communication Network (TCN), where WTB has the role of the train backbone and MVB works as consist network [3, 19-20] as shown in figure 2.

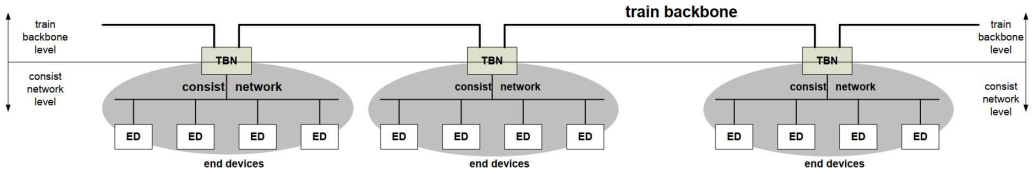


Figure 2. TCN according to the standard EN 61375 [3, 13].

2.3.1 Multifunction Vehicle Bus, MVB

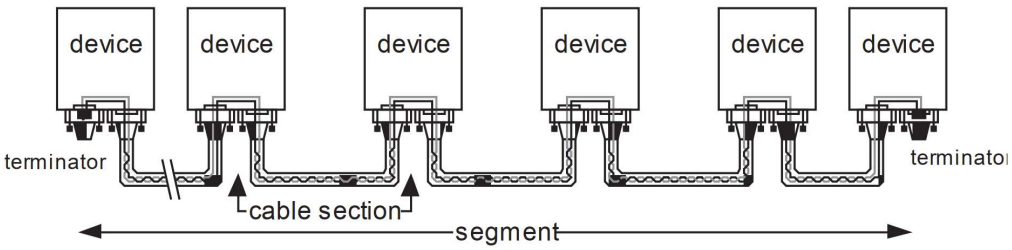


Figure 3. MVB configuration according to the standard EN 61375 [4, 51 (Sic: The last letter of the word 'terminator' on the right side of the picture is cut like that in the standard)].

To ensure reliability, the MVB uses four wires: a redundant pair of two wires, used to transmit a differential signal. The bus is very robust. It is connected as a chain with terminations at both ends of the chain – as shown in figure 3.

The MVB needs one bus master, all other devices must be configured as slaves. Both master and slaves, can transmit and receive frames. Data is sent from the source and read at the sink. The master is needed to hold all ports used in the bus. Typically the ports used for sourcing and sinking the messages are configured statically, the configuration is read at startup and no port can be added or removed during run time.

With the static configuration, the MVB is used for a group of devices, that is not changed during run time. Usually, this is the devices inside of one vehicle of the train, a locomotive, or a coach.

A bridge (BG) is used to connect two separate chains of MVB. As one MVB can have only one master, the master is also a vulnerability of the bus, because the whole bus will be stop functioning when the master is lost. For this reason it is better to separate long chains to several parts, where each part has its own master.

2.3.2 Wire Train Bus, WTB

The WTB contains like the MVB four wires – a redundant pair of two wires used to transmit a differential signal. Also WTB is connected as a chain. The same cables can be used for MVB and for WTB connections. It is specified by the standard EN 61375-2-1 "a serial data communication bus designed primarily, but not exclusively, for

interconnecting consists which are frequently coupled and uncoupled, as is the case of international UIC trains" [5, 13].

The configuration differs completely to MVB. WTB is designed to establish and recognize new nodes during run time. This makes it useful for the communication between the locomotive and a variable number of coaches or several train units, that is: one or two locomotives and coaches. Whenever a coach is connected to the train, the new node is found, and the communication is established. WTB does not use terminations, because nodes can be added or removed at any time.

A gateway (GW) combines MVB and WTB by translating the MVB messages to WTB and vice versa (see figure 4).

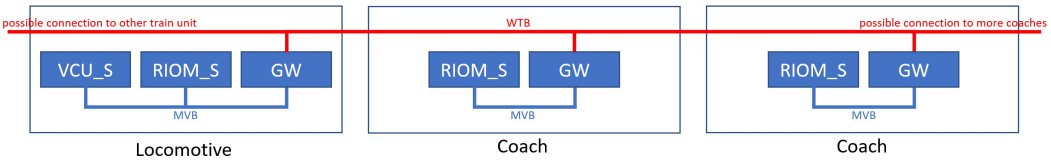


Figure 4. TCN realization with WTB and MVB.

2.3.3 Safe Data Transmission Version 2, SDTv2

SDTv2 is the certified protocol to be used for communication of safety systems in trains. It "provides a safe communication path between a source of safety related (vital) data (SDSRC) and one or several sinks of those data (SDSINK)" [6, 162]. SDTv2 is described in detail in the standard EN 61375, part 2-3, Annex B [6, 162-184]. It defines the footer of the message, comprising a version number, a life sign and the checksum created over the message content (including the life sign) and the so-called Safety Message Identifier (SMI). The SMI is unique for every communication channel of the bus. By recalculating the checksum, the sink device can validate the message content and the source of the message. To transmit a valid message, the source device needs to use the correct port and must create the checksum over the correct SMI.

SDTv2 defines the handling of communication loss. An old message whose life sign is not changed anymore can be used for a configurable amount of bus cycles (typically eight cycles). When the message is updated again and life sign is changing, a certain

amount of fresh frames must be received (typically 1000 frames) before the message is accepted and the information may be used.

2.4 Units

Electronic units used in trains normally come in racks with several slots, where different modules can be inserted. The modules in one rack communicate via busses in the back-plane of the rack. For safety systems so called double racks are used. In a double rack two electrically separated devices can be placed. Each of them has an independent power supply. A redundant pair of units can be in the same rack.

2.4.1 Safe Vehicle Control Unit, VCU_S

The VCU_S (see figure 5) is the center and the head of the STCMS. [7; 8.] With its digital inputs and outputs it is connected directly to the driver's desk and to the Brake Control Unit (BCU). VCU_S reads driver's inputs such as traction handle positions and overrides and controls the warning lamps in driver's desk. It can request an emergency brake from the BCU. VCU_S is located in the locomotive.

Figure 5. Example for a redundant pair of VCU_S with the modules (left to right): Central processing unit, MVB, three modules for PT sensor reading, Power supply [7].

2.4.2 Safe Remote Input/Output Module, RIOM_S

RIOM_S [9] – or Remote Input/Output Unit, as it called by some vendors [10] – is a unit located for functional reasons in a different place than VCU_S. It can be located in a

locomotive or in a coach. It might be necessary to be close to sensors to avoid signal deterioration or to be in coaches to monitor locally safety related functions such as door control. RIOM_S reads analog signals from sensors and local digital inputs. It processes the readings and reports them to VCU_S, where the decisions are made about protective actions like a brake activation.

2.4.3 Redundancy

A way to reduce the failure rate of a system is redundancy. Redundancy means, that two identical units perform the same tasks simultaneously. VCU_S reads both units of a redundant pair. In normal operation the message of the primary unit is considered, and the data of the secondary unit is discarded. In case of a failure of the primary unit, VCU_S can instantly start using the messages from the values coming from the secondary unit.

For this purpose two separate chains of MVB are needed. VCU_S reads both busses, all other devices are connected only to one of them, to the primary or the secondary bus. At a time, only one VCU_S writes to both busses. The other one runs all functions without writing anything to the bus and monitors the functioning of the primary unit. Whenever a failure is noticed, the primary unit is silenced, and the secondary unit takes over. The type of redundancy of VCU_S is called *dynamic redundancy* (see figure 6).

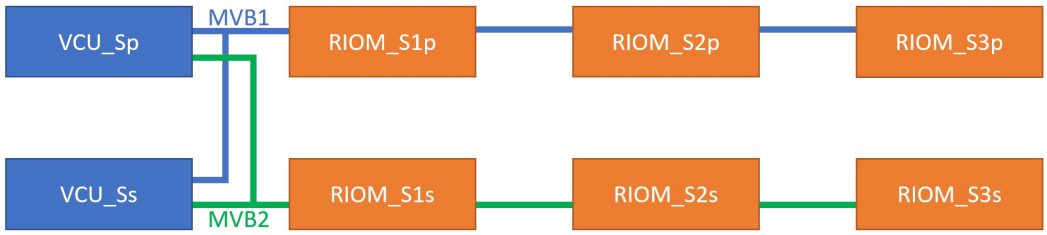


Figure 6. Scheme of Redundancy.

2.5 Modules

2.5.1 Central Processing Unit

In this module an operating system can be expected. For this purpose, a version of embedded Linux can be used. It collects the data from the other modules and reads their

diagnostics. It takes care of all network levels higher than the physical level. The incoming frames are checked for their SDTv2 compliance, and the frames are prepared to be transmitted on the bus. The incoming messages are parsed, and the information is used to make decisions. An example for this type of unit can be found at EKE-Electronics [11].

2.5.2 MVB/WTB Module

The bus modules take care of the physical layer of the network communication. They provide received frames to the application at the central processing unit and transmit the frames coming from there to the bus. LEDs on the outside panel of the module inform about the correct traffic on the bus or eventually about failure. Diagnostics about the bus and the module itself are supplied. [12; 13.]

2.5.3 Input/Output Modules

Digital Input/Output modules are widely used to read digital inputs and to write digital outputs. [14.]

Special modules are used to read PT sensors, temperature sensors made from platinum. Due to the naturally quite linear increase of the resistivity of platinum, PT sensors are known to be very exact. Most common are two types of platinum temperature sensors: PT100 and PT1000, where the number indicates the resistance of the sensor at 0 ºC in Ohm. [15.]

Other analog input modules are used for other sensors, providing current or voltage defined signals. Various sensors can be read by these modules, such as pressure or acceleration sensors. [16.]

Especially for the use in SIL-2 systems, it is important, that diagnostics about each input and output channel and about the state of the module are supplied.

3 Examples for STCMS Architectures and Test Systems

There is two different train types: trains with a variable consist and trains with a fixed consist. Variable consist trains comprise as many coaches as necessary for a certain

route; coaches can be added or removed whenever needed. A fixed consist is defined at production. The number of coaches cannot be changed later.

A test system for a certain STCMS needs to contain a representative amount of devices. There must be at least one example of each safety unit used in the train. Additional devices that are not under test themselves might be needed for instance for the communication between the safety units. Furthermore, testing units monitor the communication and simulate units that are not available in the test system.

3.1 STCMS in a Variable Consist Train

TCN was designed to handle variable consists. Figure 7 shows a train with the TCN architecture as described by the standard [3, 13]: WTB goes throughout the whole train, MVB is used for communication inside of one vehicle. This train has the following units:

Locomotive:

- One redundant pair of VCU_S
- One redundant pair of RIOM_S, locomotive type
- Non-safety units for communication: GW (WTB – MVB)

Coaches:

- One redundant pairs of RIOM_S coach type in every coach
- Non-safety units for communication: GW (WTB – MVB)

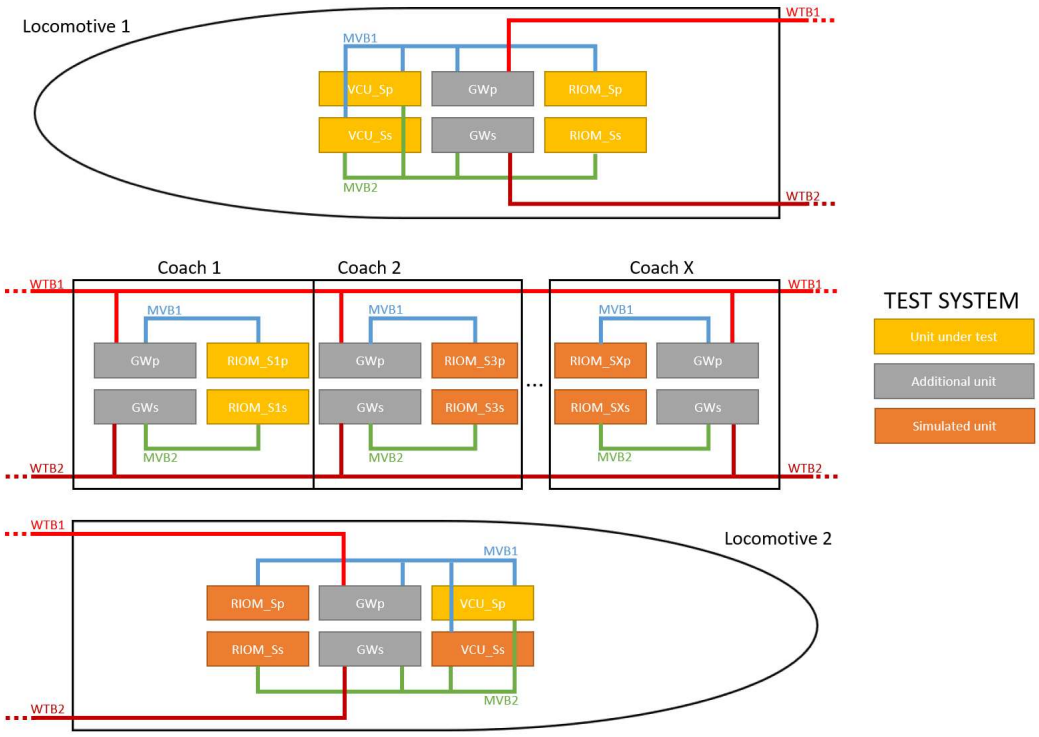


Figure 7. STCMS in a variable consist train.

Figure 8 presents a test system suitable for testing the functionality of the STCMS for a variable consist train.

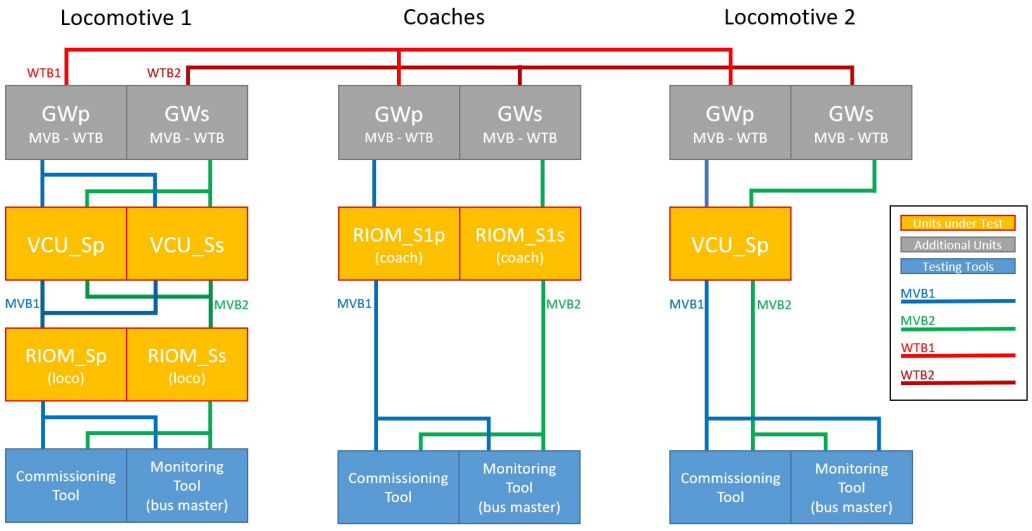


Figure 8. Test system for STMCS in a variable consist train.

First Locomotive

- One redundant pair of VCU_S: To test the dynamic redundancy of VCU_S at least one redundant pair is needed

- One redundant pair of RIOM_S, locomotive type: To test the failure one unit of a redundant pair, at least one pair of RIOM_S is needed.

Additional unit, which is not within the scope of testing: GW

Coaches

- One redundant pair of RIOM_S, coach type: To test the failure one unit of a redundant pair, at least one pair of RIOM_S is needed

Additional unit, which is not within the scope of testing: GW

Additional units, which are not within the scope of testing: two bridges (combining the separate MVB in the locomotives with the MVB of the coaches)

Second Locomotive

- One piece of VCU_S: The locomotive to locomotive communication must be tested.

Additional unit, which is not within the scope of testing: GW

3.2 STCMS in a Fixed Consist Train

WTB is an open bus. It collects data form and about all active nodes. The amount of active nodes can change any time, without disturbing the functionality of the bus. This is not needed in a fixed consist. Fixed consist trains can be designed using MVB only.

MVB depends entirely on one and only one bus master device. This is one of the various reasons, why it is advisable to split the MVB chain into several parts with a master device

in each part. In case of a failure of one bus master, the functionality of the other part is not interrupted.

Two separate MVB chains can be connected by a so-called MVB bridge (BG). An MVB bridge is a device to two MVB chains, copying data frames from one MVB chain to another.

Figure 9 shows a fixed consist train with two locomotives and a certain number of coaches between the two locomotives. There is two separate MVB chains: one for each locomotive and one for all coaches. They are connected by two MVB bridges.

This train has the following units:

Locomotive:

- One redundant pair of VCU_S
- One redundant pair of RIOM_S, locomotive type

Coaches:

- One redundant pairs of RIOM_S coach type in every coach
- Non-safety units for communication: BG (MVB locomotive to MVB coaches)

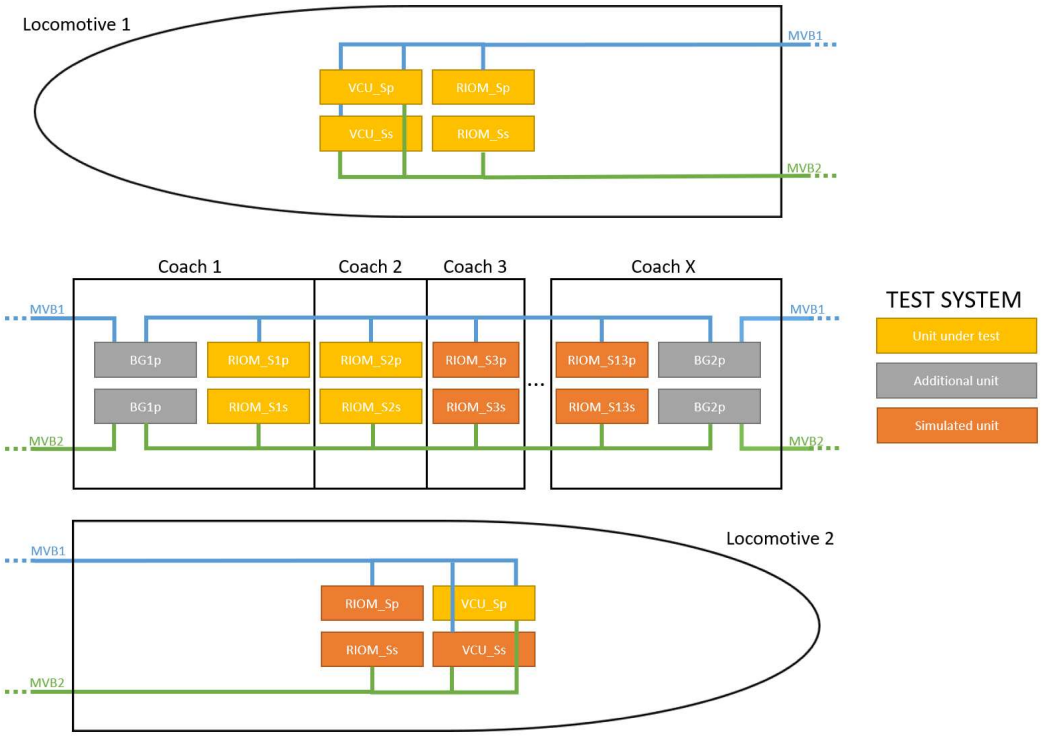


Figure 9. STCMS in a fixed consist train.

The test system for a fixed consist train is demonstrated in figure 10.

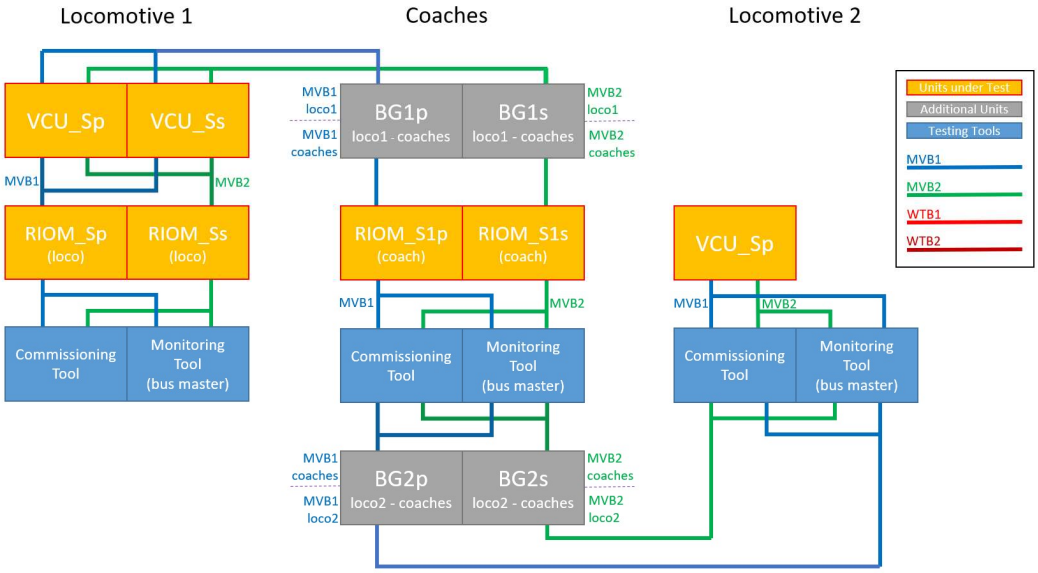


Figure 10. Test system for STMCS in a fixed consist train.

First Locomotive:

- One redundant pair of VCU_S
- One redundant pair of RIOM_S, locomotive type

Coaches

- One redundant pair of RIOM_S, coach type

Additional units, which are not within the scope of testing: two bridges (combining the separate MVB in the locomotives with the MVB of the coaches)

Second Locomotive

- One piece of VCU_S

4 System Under Test

4.1 Software: Safety Applications

The software under test comprises three different applications, run on the three different units that STCMS consists of. They collect digital and analog inputs, they communicate via WTB and MVB. The application of VCU_S sets the most crucial digital outputs of the train functionality, such as:

- warning lights in the driver's desk
- activation of the motors
- activation of the emergency brake
- enabling the doors

4.2 Examples of Safety Functions

4.2.1 Safety Functions Depending on Digital Inputs Only

Following safety functions depend only on digital inputs:

- Passenger Alarm: Whenever a passenger alarm is triggered, STCMS needs to stop the train. However, stopping the train is more urgent, when the train is in a station or leaving a station, because there is a high probability, that the passenger alarm was activated due to a door problem: somebody might be stuck in the door for instance. The driver has a pushbutton to delay the braking due to a passenger alarm for a brief time (< 1min). STCMS has to make sure, that this delay happens only, when the train is outside of a station. [18.]

- Parking Brake Applied in Motion: If the parking brake is active or activated, when the train is moving, STCMS must stop the train. [18.]

- Fire Alarm in Locomotives: If a fire is detected in any locomotive of the train, a warning lamp is lit in the driver's desk. [18.]

4.2.2 Safety Functions Using Analog Inputs

Following safety functions use digital inputs and analog signals coming from various sensors.

- Rolling Temperature Monitoring: The temperatures of the bearings of the wheelsets are monitored. In case of overheating three alarm levels are activated. When the third level is reached, STCMS must stop the train. [18.]

- Rolling Stability Monitoring: The accelerations of lateral vibrations of the wheelsets are monitored. If these accelerations come close to 1g (9.81 m/s2), the danger of derailment of the train becomes too big. Therefore, the train needs to be stopped before it gets that far. STCMS raises a speed limit when a certain threshold of acceleration values is surpassed. After another limit value is reached, STCMS must stop the train. [18.]

5 Existing Tools for Manual Testing

5.1 Testing Scope and Testing Method

The certification process of safety systems demands that 100 % of the customer requirements are implemented and tested. All test cases need to be specified in documents, all specified tests must be run, and all of them have to pass. This is the final scope of testing after the implementation is fully developed.

The customer requirements of the high speed train project are extremely detailed. Every data set transmitted or received by any safety unit is defined bit per bit. Additional to the communication between the safety units there is an exuberant amount of signals sent from the safety units to other non-safety units, which are feeding information to the driver's display and collecting diagnostics data for the maintenance personnel. All these data are sent to the MVB.

Monitoring the MVB communication of the system provides therefore a detailed knowledge about the processes going on in the system. A software bug, that never becomes visible in any MVB message is highly unlikely.

5.2 Reading the Bus and Writing to the Bus: Tools and GUIs

Before the high speed train project, the tools used to monitor the bus traffic at MVB were only command line interfaces. Writing a command and a configured port number of a certain data set to the Linux shell would print the content of the dataset in hexadecimal values to the screen. The MVB traffic of this project consists of 234 configured data sets with up to 200 mostly one bit signals. It is difficult enough and certainly error prone to monitor the changes of single bits from hexadecimal values. With these vast amounts of signals it was not promising to test even the simplest safety function with the existing CLIs.

Before test automation was started, tools for manual testing were created. Those tools do not belong to the scope of the here described test automation project, but a certain understanding of them is vital for describing the test automation process.

A simple application was created running on hardware similar to VCU_S that would provide the data sets at all configured ports to an ethernet multicast group. The ethernet data could be easily read by a PC application that also provided the graphical user interface to present all data from the MVB in human readable form.

In this GUI (see figure 11) the data sets were not named with their port number but with the names of the sourcing and the sinking device (e.g. VCU_S To RIOM_S). All signals already had human readable names defined by the customer in the so-called interface control document (ICD). These names are displayed together with their values in primary and secondary MVB. The system from the hardware connected to and reading the MVB up to the GUI was called Monitoring Tool.

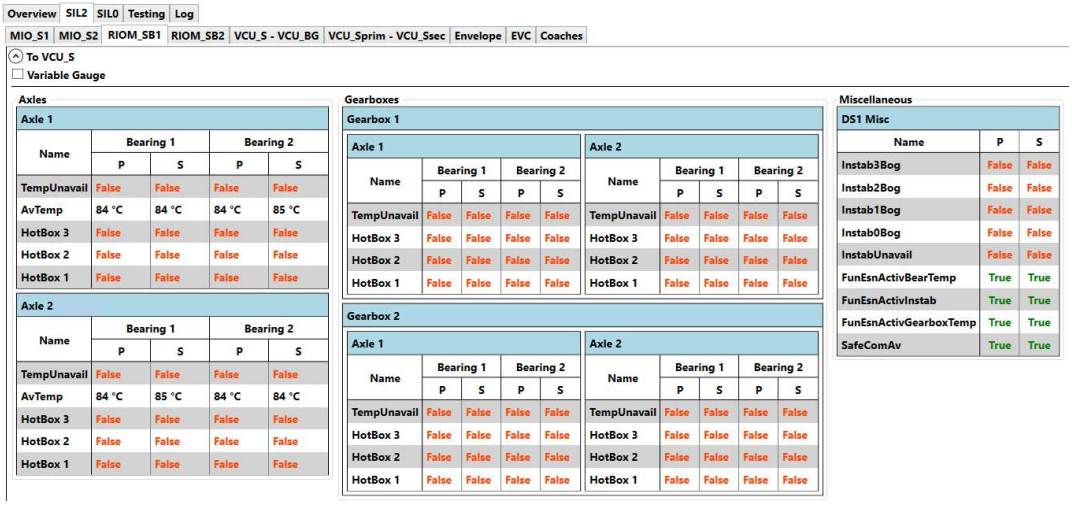


Figure 11. Example of a data set displayed in the Monitoring Tool.

As the test system does not comprise all units used in a train, at least the data sets of the non-existing units should exist in the bus. The Commissioning Tool was created to provide the simulated units, especially the coaches 3 to 12 where no hardware exists in the test system. In the GUI (see figure 12) the user can set the values and send the data sets to an ethernet multicast group. On another hardware similar to VCU_S an application reads from the multicast group and transmits the data to the MVB.

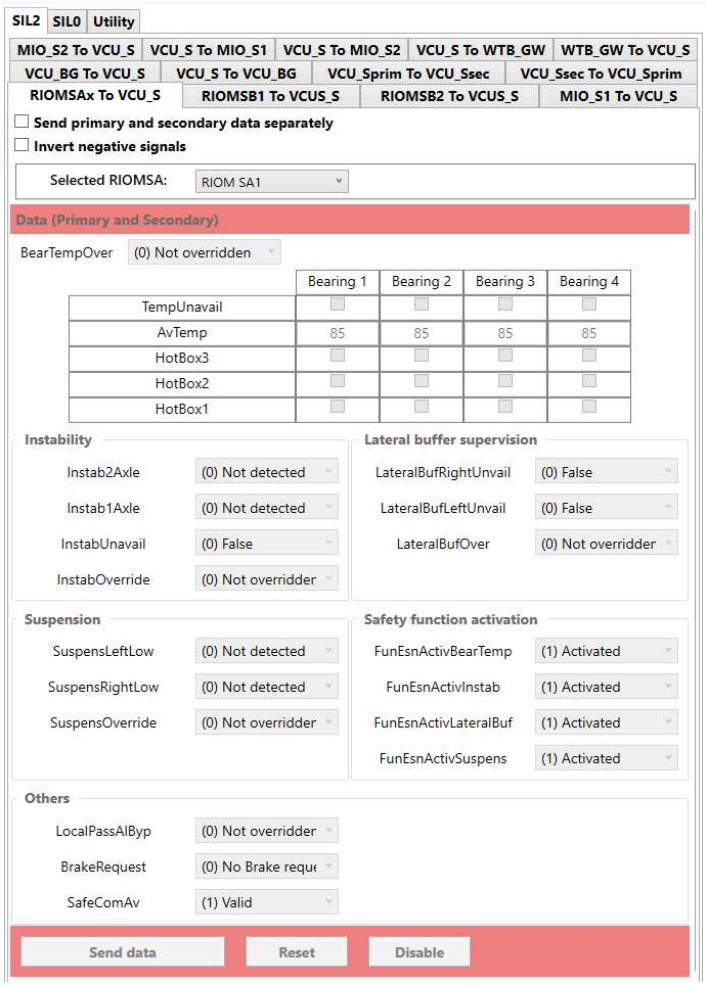


Figure 12. Example of a data set displayed in the Commissioning Tool.

5.3 Simulating Inputs

Digital inputs are simulated with simple switches collected to switchboxes, where each switch connects the digital input pin either to the specified voltage or to ground. LEDs integrated in the switchbox show the states of the digital outputs.

Simulating analog inputs is a bit more challenging. A temperature sensor changes its resistance with the temperature therefore it can be simulated with a potentiometer. Increasing the resistance will be read by the Pt100/Pt1000 Temperature Sensor Input module (PTI) as increase of the temperature. One major problem of this temperature sensor simulation is the accuracy. As long as only one threshold is monitored, it might be not too difficult to find a position of the potentiometer below and another above this threshold.

With different alarm levels and several thresholds it becomes much more difficult to find reliably the different positions below, above and in between those thresholds.

Testing became practically impossible due to another functionality. By nature temperatures do not change quickly. This fact is frequently used for evaluation of temperature sensors. A so-called gradient error is raised, whenever the measured temperature is changing too quickly. When the gradient error is raised, the temperature sensor is discarded due to this failure. Manually testing a software evaluating the sensor constantly for a correct temperature gradient means to turn the potentiometer evenly and slowly. With some practice even that might be possible for one simulated sensor. But there are numerous sensors used, that should change their value equally. If one sensor behaves notably differently than the others, this is regarded to be again another error and a failure of the sensor. Manual testing was not possible and new solutions had to be found.

In other projects inputs to the High Speed Analog module (HSA) – an EKE product reading current and voltage signals – are simulated with function generators. The function generators used do not provide more than two outputs. The high speed train project requires three different signals, which are fed to all together 14 input pins of six HSA modules. Such a solution would be quite costly using the same function generators. Otherwise, reach for finding more suitable products would have been necessary. The decision was made to develop a special tool for simulating the inputs to the HSA module.

6 Establishing Test Automation

6.1 The Robot Framework

The Robot Framework is a generic open source framework used for test automation. It was originally designed at Nokia Networks and open sourced in 2008. [19.]

The framework is coded with Python. It offers some default libraries and can be extended by project specific Python libraries. The tests are run by scripts with the file extension *.robot, which call functions defined in the included libraries. The functions are called keywords. The Robot Framework reacts to Python exceptions [20]: The test passes, when during its execution no Python exception is raised; if any exception – built in or customer exception – is raised, the test is regarded failed.

In the Robot scripts blanks are allowed within a keyword. To make the ending of one and the beginning of another keyword clear the separator '|' can be used. Following Python syntax, the indentation is decisive. If the separator '|' is used, the indentation is written like that: '| |' – followed by the keyword. Comments are started with '#'.

The Robot script requires the sections mentioned in listing 1:

```
| *** Settings *** |
| *** Test Cases *** |
```

Listing 1. Headers for settings and test case sections.

In the 'Settings' section the used Python libraries, the test suite setup and teardown are defined. Each test case in the 'Test cases' section starts with a title. These titles are treated as keywords themselves, therefore all test case titles must be unique in a script. After the title, the keywords are following, that define the test and the expected result. All lines belonging to one test case must be indented.

Optionally, there can be at least two more sections as shown in listing 2:

```
| *** Variables *** |
| *** Keywords *** |
```

Listing 2. Headers for variables and keywords sections.

The section 'Variables' offers room to define variables used in this test suite. The keywords used in the tests can be defined either in a library that was mentioned in the 'Settings' section or in the special 'Keywords' section. The keywords defined in the 'Keywords' section are higher level functions using functions from the libraries.

A test suite defined in a Robot script is started with a command line in a command prompt window at the PC where the Robot Framework and its libraries are installed. It is started with the command 'robot'.

This command requires at least one argument, which is the name of the Robot script file or a folder, where at least one Robot script file is located. In case a folder name is used as argument, all Robot scripts located in the folder will be executed. The command line can be extended by a vast variety of options. For instance:

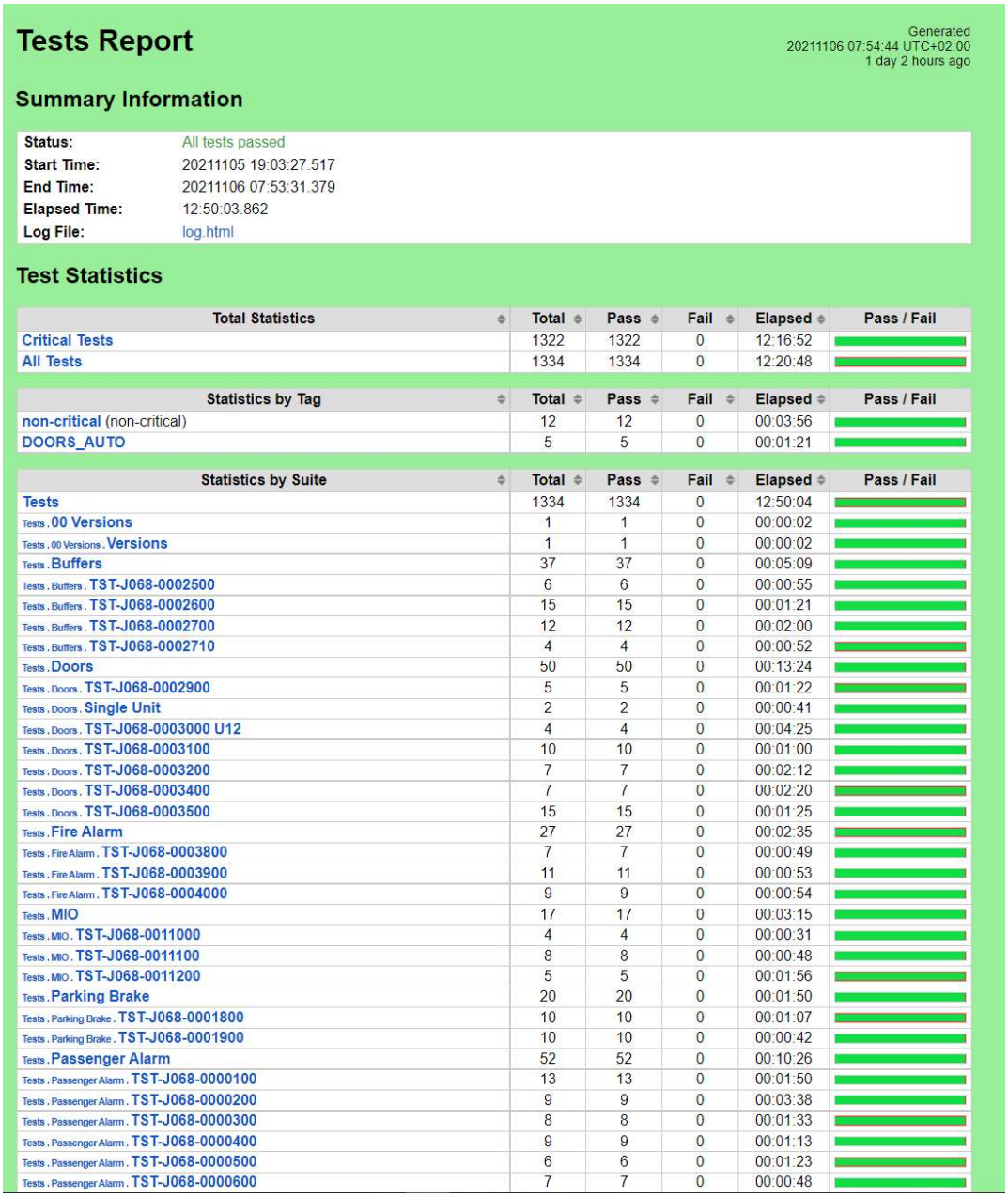
- values can be passed to variables defined in the script: -v <variable_name>:<value>

- a single test case of the script can be run: -t <test_case_title>

- especially tagged test cases can be excluded from being run by adding: --exclude <tag_name>

From the test case titles defined in the script and the test results, the Robot Framework creates test reports that will be stored after running the test suite in the same directory where the Robot Framework was started with the *Robot* command. A part of a test report is shown in figure 13.

Additional information can be gathered on three levels: the default 'LOG' level, the 'DEBUG' level with more information and the 'TRACE' level including all available information. All levels include time information with the resolution of single milliseconds. The additional levels can be activated for a test suite by adding the argument "-L DEBUG" or "-L TRACE" to the command line starting the test script.



Tests Report

Generated
20211106 07:54:44 UTC+02:00
1 day 2 hours ago

Summary Information

Status:	All tests passed
Start Time:	20211105 19:03:27.517
End Time:	20211106 07:53:31.379
Elapsed Time:	12:50:03.862
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1322	1322	0	12:16:52	
All Tests	1334	1334	0	12:20:48	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
non-critical (non-critical)	12	12	0	00:03:56	
DOORS_AUTO	5	5	0	00:01:21	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Tests	1334	1334	0	12:50:04	
Tests . 00 Versions	1	1	0	00:00:02	
Tests . 00 Versions . Versions	1	1	0	00:00:02	
Tests . Buffers	37	37	0	00:05:09	
Tests . Buffers . TST-J068-0002500	6	6	0	00:00:55	
Tests . Buffers . TST-J068-0002600	15	15	0	00:01:21	
Tests . Buffers . TST-J068-0002700	12	12	0	00:02:00	
Tests . Buffers . TST-J068-0002710	4	4	0	00:00:52	
Tests . Doors	50	50	0	00:13:24	
Tests . Doors . TST-J068-0002900	5	5	0	00:01:22	
Tests . Doors . Single Unit	2	2	0	00:00:41	
Tests . Doors . TST-J068-0003000 U12	4	4	0	00:04:25	
Tests . Doors . TST-J068-0003100	10	10	0	00:01:00	
Tests . Doors . TST-J068-0003200	7	7	0	00:02:12	
Tests . Doors . TST-J068-0003400	7	7	0	00:02:20	
Tests . Doors . TST-J068-0003500	15	15	0	00:01:25	
Tests . Fire Alarm	27	27	0	00:02:35	
Tests . Fire Alarm . TST-J068-0003800	7	7	0	00:00:49	
Tests . Fire Alarm . TST-J068-0003900	11	11	0	00:00:53	
Tests . Fire Alarm . TST-J068-0004000	9	9	0	00:00:54	
Tests . MIO	17	17	0	00:03:15	
Tests . MIO . TST-J068-0011000	4	4	0	00:00:31	
Tests . MIO . TST-J068-0011100	8	8	0	00:00:48	
Tests . MIO . TST-J068-0011200	5	5	0	00:01:56	
Tests . Parking Brake	20	20	0	00:01:50	
Tests . Parking Brake . TST-J068-0001800	10	10	0	00:01:07	
Tests . Parking Brake . TST-J068-0001900	10	10	0	00:00:42	
Tests . Passenger Alarm	52	52	0	00:10:26	
Tests . Passenger Alarm . TST-J068-0000100	13	13	0	00:01:50	
Tests . Passenger Alarm . TST-J068-0000200	9	9	0	00:03:38	
Tests . Passenger Alarm . TST-J068-0000300	8	8	0	00:01:33	
Tests . Passenger Alarm . TST-J068-0000400	9	9	0	00:01:13	
Tests . Passenger Alarm . TST-J068-0000500	6	6	0	00:01:23	
Tests . Passenger Alarm . TST-J068-0000600	7	7	0	00:00:48	

Figure 13. Upper part of a test report: All 1334 tests cases are passed; running them took almost 13 hours - this is done usually during the night, in this case 19:00 in the evening to almost 8:00 in the morning.

6.2 Replacing the GUIs with TestingCLI

For manual testing graphical user interfaces are invincibly more practical than command line interfaces. For test automation it is the other way round: It is a lot easier to enter lines to a CLI than to address fields of a GUI. Therefore the existing tools – the PC

applications controlling Monitoring and Commissioning Tool – were combined to one command line interface, called: TestingCLI.

TestingCLI is able to connect to a certain part of the test system (first or second locomotive, coaches), it can create and read MVB messages. For establishing the connection and for signal setting it returns ‘Success’ or ‘Error’. If a signal is read, TestingCLI returns the read values of MVB1 and MVB2 or ‘Error’. Logs are gathered that contain additional information about the errors.

6.3 Network Controlled Input Simulators

TestingCLI made it possible to automate reading MVB and creating messages to the bus. But it did not solve the question, how to automate digital and analog inputs. Special devices were needed, that would be connected to the PC running the Robot Framework, receiving commands from there and providing inputs to the test system.

The Raspberry Pi 3 provided the needed features: enough memory and processing power to run an application controlling the needed channels and producing different types of signals and an ethernet socket to connect the device to the private network. Moreover, Raspberry Pi offers the option to connect a special PCB on top of the general purpose IO pins. This special board is called HAT – Hardware Attached to Top. There is boxes available, that can house a RPI with the HAT.

It seemed reasonable to design small boards:

- one that would be able to feed digital inputs at the specified voltage and also to read the outputs of the DIO modules,
- another one that provides adjustable resistances to simulate a PT100/PT1000 sensor,
- finally a board with DACs to create designed voltage or current signals.

Applications run on the RPI should control the HATs and take care of the communication with the Robot Framework.

6.3.1 DIO Simulator

The simplest task was the DIO simulator. Regulators needed to boost 5 V delivered from RPI to 24 V for the inputs of the DIO module. The outputs of the DIO module needed to be stepped down to 5 V, which are readable for the RPI. The applications takes care of the setting and the reading and the network communication with the Robot Framework.

6.3.2 PTI Simulator

The PTI simulator was developed in another thesis project at Metropolia by Oleg Languev. [21.] It was a crucial part for test automation.

There are digital potentiometers available that provide the needed resolution. Calibration was needed to guarantee the defined accuracy of +/- 1° C. This accuracy was regarded to be good enough for testing purposes. The calibration process changes the resistance of the digital potentiometer gradually, crosschecks the resistance with the read temperature value at the PTI module, and saves the state of the potentiometer for each temperature to make in available for reproducing it.

The PTI simulator does not need to evaluate the PTI module and its firmware. They are generic and they were tested and certified earlier. It is therefore allowed to calibrate the simulator with the PTI module, and an external calibration tool is not needed.

6.3.3 HSA Simulator

As mentioned in 4.4 the high speed train project requires three different analog inputs to the HSA module. All of them are defined by the current. There are two types of acceleration sensors, one providing a rectified signal, the other a swinging signal with an offset of 12 mA (the middle of the used range of HSA, 4 – 20 mA), both band pass filtered before being fed to HSA with cutoff frequencies of 3 Hz and 9 Hz and with a center frequency of 6 Hz. The third signal comes from a pressure sensor. [18, #1634.]

Although the lateral accelerations of the axles of a train most definitely do not produce such ideal wave forms, the two acceleration sensor signals can be simulated as sinusoidal signals. In the MVB messages the RMS of these signals is sent. From sinusoidal wave forms it is easy to find the RMS value by multiplying the amplitude with the square

root of two. The applications of the two types of RIOM_S calculate the RMS value from the acceleration sensor inputs. To monitor the correct calculation, it is feasible use a wave form whose RMS value is not too difficult to find.

The HSA module has an input resistance of 120 Ω, therefore current inputs between 0 and 24 mA can be simulated by voltage controlled signals between 0 and 1200 mV. The DAC MCP4822 is controlled via SPI. It has two output channels that are chosen by the most significant bit of a 16 bit value. The 12 least significant bits define the output voltage between 0 and 4095 mV, so every bit resembles 1 mV, which makes it fairly easy to manage. This DAC was chosen for the RPI HAT.

The application was designed using the C programming language. To ensure the real time functioning of the application, it seemed reasonable to fork the application process to a parent and a child process. RPI 3 has four cores. If ever the operating system allows it, it can run two processes truly in parallel. The parent process reads UDP messages or optionally a small command line interface and prepares the signal by calculating 1000 values for each output channel, which are saved to shared memory. The child process updates every millisecond the output value of all channels by sending them via SPI to the DACs. To avoid publishing values while they are written, two sets of values are used at shared memory. One is reserved for the child process. The other one is used by the parent process for writing. Whenever the parent process finishes writing, it commands the child process to switch the sets and take the new values into use.

6.4 Data Flow in the Test System

The data flow in the test system is described in the following figure 14.

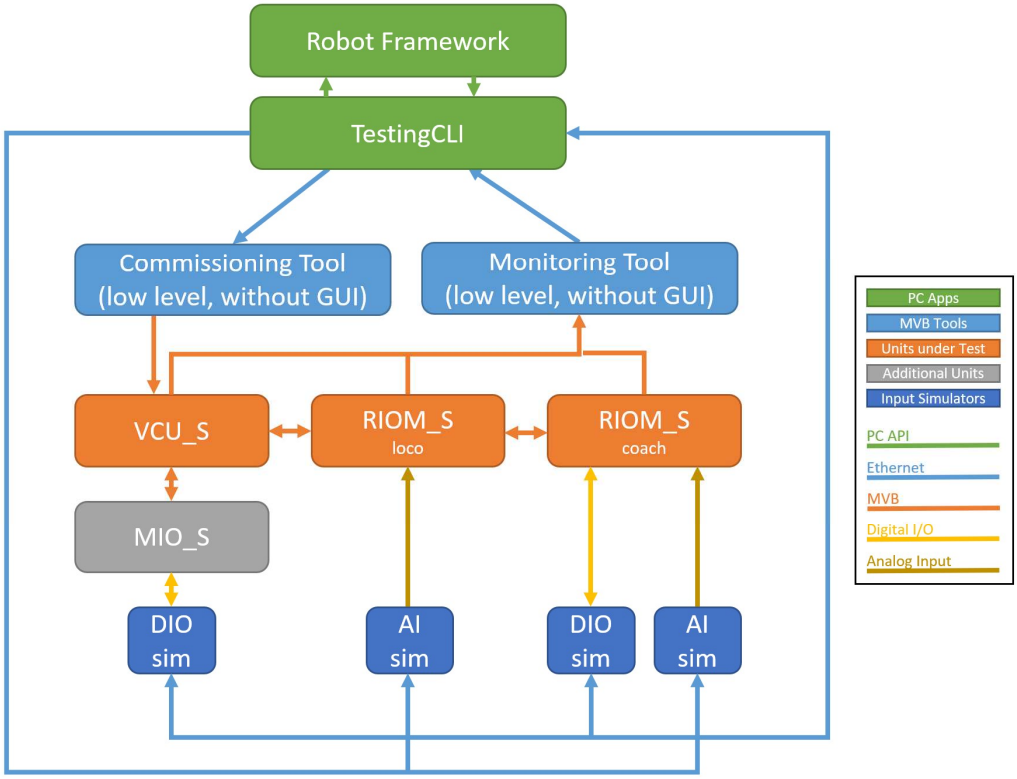


Figure 14. Data flow in the test system.

Input to the test system:

- On the PC, the Robot Framework runs the client application TestingCLI and feeds to TestingCLI the names of the signals, which need to be sent or read, according to the Robot script.

- TestingCLI addresses the Commissioning Tool and the input simulators via ethernet and provides the information of the signals needed for the test to them.

- The Commissioning Tool creates messages to the MVB and simulates to VCU_S the units, which are not available in the test system, and controls their messages to VCU_S.

- The input simulators deliver the needed digital and analog inputs to the modules of the units.

Output from the test system:

- The DIO simulator reads the digital outputs from the DIO module and reports their values to TestingCLI via ethernet.
- The Monitoring Tool reads all messages available on the MVB.
- TestingCLI polls via ethernet from the Monitoring Tool the signals needed for testing and reports their values to the Robot Framework
- The Robot Framework compares the received values with the expected values from the script and creates the test report.

6.5 Automating the Test Procedures – Writing the Scripts

As mentioned above in 4.1, the certification process demands that integration tests cover 100% of the requirements. These tests need to be designed, documented, and reviewed. To achieve the certificate all these tests naturally need to pass.

Before the automation, the test procedures were defined at least at a draft level. They were initially designed for manual testing and also tried out as far as possible manually. When the test automation was built up to a certain level, the writing of the test scripts was started. It was in the beginning mostly a translation of the test specifications to Robot and json files readable for the Robot Framework. The integration test specifications were done individually for each safety function. The first ones that could be automated were the safety functions depending on digital inputs only. Some of them are quite simple – like the Parking Brake Applied in Motion, the Fire Alarm, or the Brake Pipe Monitoring. They were quite naturally the ones to start with.

The structure of these test procedures is simple and does not require much of the logic offered by the Robot Framework: one signal or a group of signals is set and immediately or after the ending of a timer a group of signals is checked.

One test case or test step in the Robot script typically looks like shown in listing 3 (lines staring with '#' are comments explaining the code line by line):

```
# Title of the test case or test step.
| TST-J068-0000700 Step 6 - Set traction handle to MINIMUM |

# A json file is defined, that contains the signals, which need to be set.
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 | ${PH1} |
| | ... | 0700_06_set.json | confPath=${CONFP_TR07} | accFailures=${true} |

# Wait. In this case just as long all signals are set and read (max. 1 s).
| | Sleep | ${set_wait} |

# A json file is defined, that contains the signals
# that need to be checked and their expected values.
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 | ${PH1} |
| | ... | 0700_06_check.json | confPath=${CONFP_TR07} | accFailures=${true} |
```

Listing 3. Typical test case in a Robot script.

Note: '| ... |' indicates the continuation of the line above (the indentation '| |' needs to be repeated).

A vital role in testing has the validation of timers. Many events will not trigger a reaction immediately, when they emerge, but only when the new state perseveres for a certain time period. For instance, it is not the single spike of a temperature higher than a threshold that is regarded to be an overheating, but it is a measured temperature that is higher than the threshold for several seconds. Likewise, an alarm will be deactivated only, when the measured value is lower that the alarm threshold for another defined time interval.

For testing that means, that two checks will be needed to evaluate the correct implementation of the timers after an event triggering a reaction of the system has been created. The first check before the end of the timer verifies that the event was noticed and there is no reaction yet, the second check after the end of the timer makes sure, that the correct reaction to the event is set.

The variables need to be defined: The timer tDEACT_Instab [s] for the deactivation of an alarm was defined by the customer, another timer 'delay' is defined for the testing purpose to define how long before and after the expiration of the customer timer the checks are done – as can be seen in listing 4:

```
| ${tDEACT_Instab} | 20 |
| ${delay} | 3 |
```

Listing 4. Variable definitions: timers.

Listing 5 shows the use of the timers, comments – starting with '#' – enlighten some details:

```
# Test step, using the timers:
| TST-J068-0004210 Step  4 - Wait > tDEACT |

# Wait until timer is almost finished and check that signals are not changed:
# Alarm is not yet deactivated.
| | Sleep | ${${tDEACT_Instab} - ${delay}} |
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 | ${PH1} |
| | ... | Level0_SB1p_check.json | confPath=${CP42} | accFailures=${true} |
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 | ${PH1} |
| | ... | Level0_B1_VCUS_check.json | confPath=${CP42} | accFailures=${true} |

# Wait until timer is definitely finished and check that signals are changed:
# Alarm is deactivated.
| | Sleep | ${2 * ${delay}} |
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 | ${PH1} | |
| ... | NoLevel_check.json | confPath=${CP42} | accFailures=${true} |
```

Listing 5. Example for timer use with the sleep function.

Reading one signal from the MVB1 and MVB2 takes between 30 and 35 ms. Therefore, about 30 signals can be read during one second.

Ideally, a test would not check only the signals changing during a procedure, but also the signals remaining the same. This way unintended side effects would be caught immediately. The system under test uses hundreds of signals, so it is impossible to read them all before and after the expiration of a timer. Practical choices have to be made and the side effects need to be caught with smoke tests in the beginning and the end of the test suite. "Smoke tests are a subset of test cases that cover the most important functionality of a component or system, used to aid assessment of whether main functions of the software appear to work correctly" [22].

6.6 Selected Challenges and Their Solutions

6.6.1 Accept Failures

By default, the Robot Framework stops a test suite at the first failure. The BuiltIn library of the Robot Framework offers the keyword shown in listing 6:

```
| Run Keyword And Continue On Failure |
```

Listing 6. Keyword used to continue test run after a failure.

This prevents the test from being stopped after a failure. But it enables only, that the following line of the Robot script is executed. The ‘Test Tool V2’, defined in the project specific Python library, raises an exception, every time a read value does not meet the expected value. In one line of the Robot script as many signals can be checked as defined in the json file. Yet only the first exception and therefore only the first failing signal would be reported.

Especially during software development, it seemed more helpful to collect information about all failing features than just stopping at the first found failure. The first found failure might have been known already and of minor importance. This made it advisable to collect a list of failures rather than to stop the test execution with the first failure.

A special flag – accFailures – was created to the ‘Test Tool V2’: If set ‘true’, it makes sure, a whole json file is executed, collecting all failures, and raising the exception only in the end of the function – as can be seen in listing 7:

```
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 | ${PH1} |
| | ... | 0700_06_check.json | confPath=${CONFP_TR07} | accFailures=${true} |
```

Listing 7. Robot script line with the accFailures flag (bold print).

6.6.2 The Opened Test Tool

TestingCLI addresses signals within an MVB data set. Those signals are mostly one bit wide, sometimes 2, 8 or more bits – according to the definition of the ICD. Once a data set is created by the Commissioning Tool, it is repeated infinitely with out a change. TestingCLI can set and clear each signal independently. But when an instance of TestingCLI is started, it has no knowledge about the signals, that were set already. Therefore, all signals, which are not explicitly set, are cleared. During run time, TestingCLI keeps track of the signals that were set and keeps them at their state, when other signals of the same data set are changed.

‘Test Tool V2’ starts a new instance of TestingCLI every time it is called. Inevitably, it clears each time all signals, which are not set by the test data. A simplified example

shows the problem – see figure 15. The signal indicating, if the locomotive is occupied, that is: if the driver is present and using the locomotive, another signal telling, if the passenger alarm is activated and the signal containing information about the opening of the doors are in the same data set. When we want to test the passenger alarm activation, we might inadvertently get the locomotive unoccupied and open the doors, because those signals are cleared.

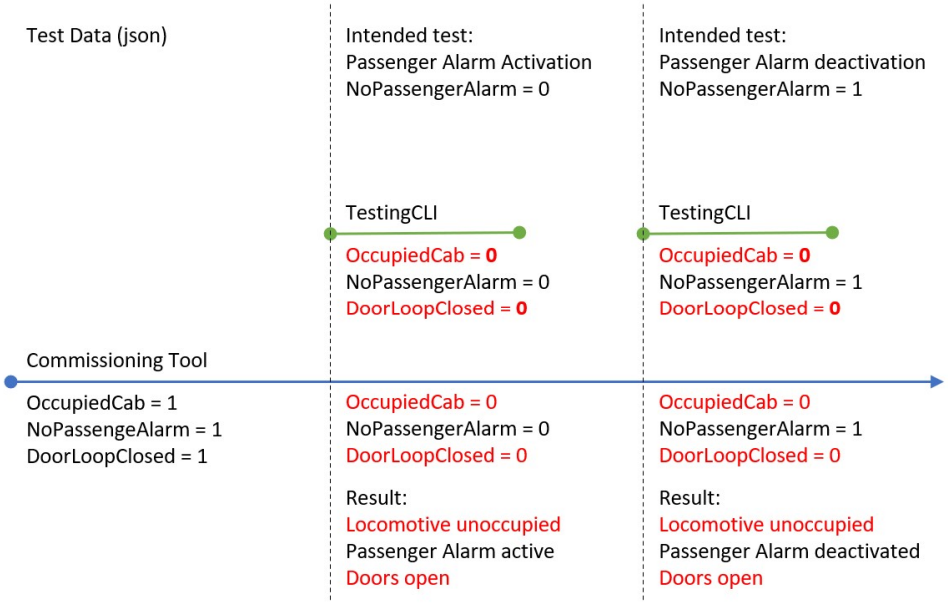


Figure 15. Side effects due to new sessions of TestingCLI with each command.

The issue can be fixed in a simple way by setting the signals, which should remain set, each time any signal in a data set is changed. This is both, tedious and error prone. This solution requires the tester to keep track manually of signals that were set before, when ever one signal must change its value – see figure 16.

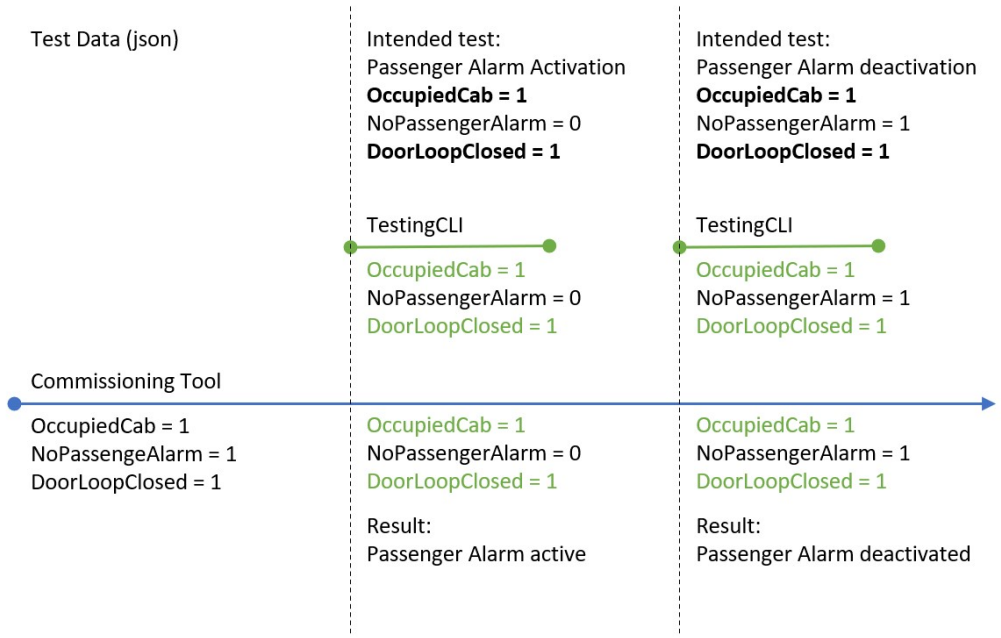


Figure 16. First fix: Test data contains all signals, which must not be changed.

Moreover, this solution did not use TestingCLI to its full potential. TestingCLI keeps track of the signals set during one session, so the data sets used can be initialized in the beginning, then only the signals that are needed for testing are addressed.

The ’Opened Test Tool V2’ made it possible to start an instance of TestingCLI in the beginning of the test suite at the suite setup and close it at teardown. An initializing json file takes care once for the signals that need to be set but are not needed for the particular test. This is shown in figure 17.

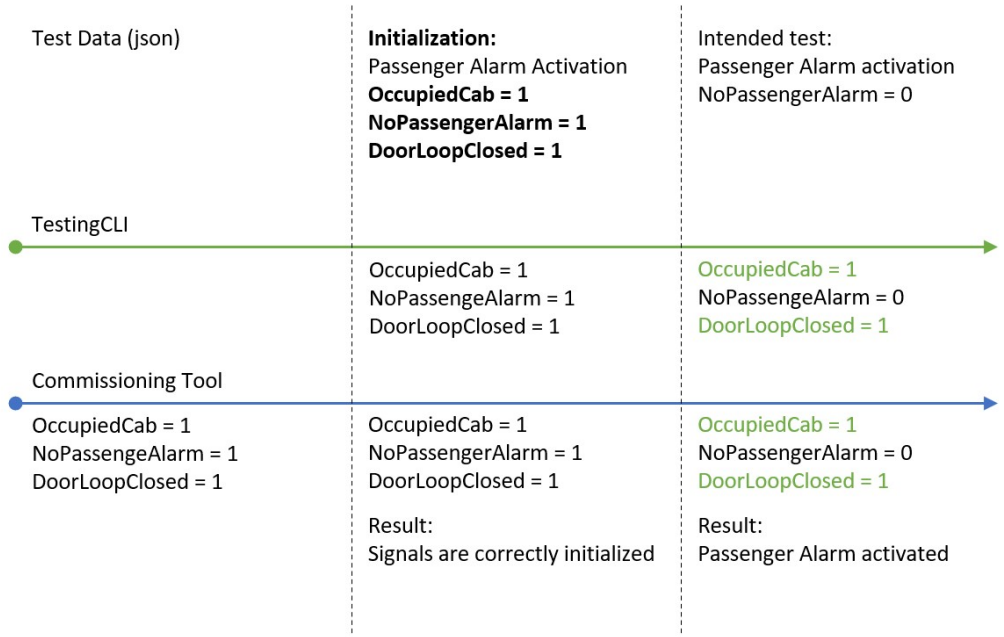


Figure 17. Real fix: TestingCLI is started in the beginning of the test suite and initialized.

Together with the 'Opened Test Tool' multiple sessions were introduced. The train and therefore also the test system has three separate MVB chains: one for each locomotive and one for all coaches. An instance of the TestingCLI is able to establish connections to as many simulators as needed, but it can monitor only one MVB. Therefore, up to three instances of the TestingCLI are needed to monitor the whole test system. Listing 8 shows how multiple sessions with connections to all needed devices are established and closed – comments inside the code explain more details:

```
| *** Settings *** |
| Library | Test_Library |
| Suite Setup | Open Many Test Tools |
| Suite Teardown | Close Many Test Tools |

| *** Variables *** |
# Connection to Monitoring/Commissioning Tool, Locomotive 1:
| ${CONN_PH1} | connect 239.0.0.100 50020 239.0.0.100 50030 8 |

# Connections to DIO simulators, Locomotive 1:
| @{MIOS2} | mioconnect VCU_S_TO_MIO_S2_P 10.0.0.56:50000 |
| ... | mioconnect VCU_S_TO_MIO_S2_S 10.0.0.57:50000 |
| ... | mioconnect MIO_S2_TO_VCU_S_DS1_P 10.0.0.56:50000 |
| ... | mioconnect MIO_S2_TO_VCU_S_DS1_S 10.0.0.57:50000 |

# Connection to Monitoring/Commissioning Tool, Coaches:
| ${CONN_CH} | connect 239.0.0.100 50021 239.0.0.100 50031 8 |

# Connection to HSA simulator, Coaches:
```

```
| @{SIM} | deviceconnect HSAsim2 10.0.0.4:700 |

# Connections to DIO simulator, Coaches:
| ... | deviceconnect DIOsim1 10.0.0.52:50000 |
| ... | deviceconnect DIOsim2 10.0.0.53:50000 |

# Connection to Monitoring/Commissioning Tool, Locomotive 2:
| ${CONN_PH2} | connect 239.0.0.100 50022 239.0.0.100 50032 8 |

| *** Keywords *** |

# Keyword called at Suite Setup to establish three sessions,
# that is three instances of TestingCLI with connections
# to the three different MVBs:
| Open Many Test Tools |
| | ${session1}= | Open Test Tool | toolPath=${TOOL_PATH} |
| | ... | initCmd=${CONN_PH1} |

# The 'extraCmds' accept the list of simulators,
# that one instance of TestingCLI should connect to:
| | ... | extraCmds=${MIOS2} |
| | Set Suite Variable | ${PH1} | ${session1} |
| | ${session2}= | Open Test Tool | toolPath=${TOOL_PATH} | initCmd=${CONN_CH}
|
| | ... | extraCmds=${SIM} |
| | Set Suite Variable | ${CH} | ${session2} |
| | ${session3}= | Open Test Tool | toolPath=${TOOL_PATH} |
| | ... | initCmd=${CONN_PH1} |
| | Set Suite Variable | ${PH2} | ${session3} |

#Keyword called at Suite Teardown to close all three instances of TestingCLI:
| Close Many Test Tools |
| | Close Test Tool | ${PH1} |
| | Close Test Tool | ${CH} |
| | Close Test Tool | ${PH2} |
```

Listing 8. Establishing various connections in multiple sessions of TestingCLI with the Robot script.

6.6.3 Acceleration RMS

In the locomotives the RMS value of the measured acceleration data is calculated over 500 m of movement. There is no sliding window, the value is updated every 500 m. As figure 18 shows, the maximum travelling distance until the acceleration RMS is fully updated is a bit less than 1 km, whereas the minimum travelling distance for the update is slightly more than 500 m.

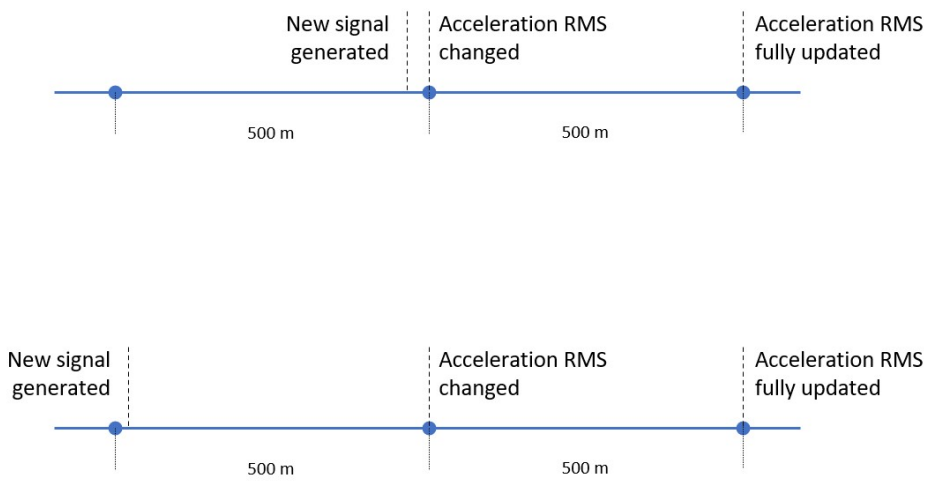


Figure 18. Different Acceleration RMS updates with a new signal.

The discrepancy between two redundant acceleration sensors is evaluated with a defined difference of the RMS value. The discrepancy error is raised, when a discrepancy between two sensors is noticed over a certain time (10 s). To test the correct functioning of the discrepancy error:

- it is necessary to create the input, which will be measured as discrepant,
- then to wait that the RMS value is fully updated,
- when the RMS value is fully updated, the timer starts for the raising of the discrepancy error,
- then check before the timer ends, that the discrepancy error is not set,
- and after the timer ends, that the discrepancy error is set.

To perform these two checks – before and after the end of the timer – is it crucial to know, when the timer starts, that is: when the RMS value is updated. With a speed of

180 km/h, travelling 500 m takes 10 s. Therefore, the full RMS update might take from a bit more than 10 s up to almost 20 s (see Fig. 14).

The solution is obviously to check repeatedly, if the RMS reached the desired value, and, when it has reached this value, to start the actual test. A loop stopped under a certain condition would be a while loop, but the Robot Framework offers only for loops, written with a syntax similar to Python – as shown in listing 9:

```
for index in range (max_value):
```

Listing 9. For loop according to Python programming language.

This makes the script slightly more complicated because a for loop is ended when the index reaches the defined maximum value. So, another condition must be defined, that aborts the loop prematurely. But with for loops it is much harder to create accidentally infinite loops.

Listing 10 shows the use of the for loop in practice; comments inside the code explain more details:

```
# Test case title:
| TST-J068-0004300 Step 4 - TARp Discrepancy, TARs Zero |

# Json file contains commands addressing the simulator
# to provide the needed signals:
| | Run Opened Test Tool V2 | ${PH1} | 4300_04_HSA.json |
| | ... | confPath=${CONF_PATH_4300} |

# Wait minimum time for RMS update (10 s).
| | Sleep | ${rms_update} |

# Start loop to check repeatedly if RMS is updated.
# The maximum value for the index (max_for) depends on the waiting time
# between the repetitions of the loop (delay_for):
# max_for * delay_for = 10 [s]
# (After 20 s the RMS value has to be updated.)
| | FOR | ${index} | IN RANGE | ${max_for} |

# Assign return value of function 'Single Signal Check' to variable RMS
| | | ${RMS} = | Single Signal Check | ${PH1} |
| | | ... | read RIOM_SB1_TO_VCU_PH_DS2 AccelerationRms |
| | | ... | expected_string=17 [01] |

# Check if RMS is updated and ...
| | | Run keyword if | "${RMS}" == "OK" |
# ... exit loop if RMS is updated or ...
| | | ... | Exit For Loop |

# ... wait a bit longer.
| | | ... | ELSE |
| | | ... | Sleep | ${delay_for} |
| | END |
```

```
# Run actual test: Wait until the timer is almost finished
# and check if signals are not changed yet.
| | Sleep | ${${tUNAVAIL_Instab} - ${delay}} |
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 |
| | ... | ${PH1} | 4300_04a_check.json | confPath=${CONF_PATH_4300} |
| | ... | accFailures=${true} |

# Wait until the timer is finished and a bit more
# and check if the signals are correctly changed.
| | Sleep | ${2 * ${delay}} |
| | Run Keyword And Continue On Failure | Run Opened Test Tool V2 |
| | ... | ${PH1} | 4300_04b_check.json | confPath=${CONF_PATH_4300} |
| | ... | accFailures=${true} |
```

Listing 10. Test step with a loop waiting for the acceleration RMS update.

6.6.4 TestStatus

So far, the described examples used only the Robot scripting language to define the test performance. The Robot scripting language is a simplified programming language, which offers principally the possibility to define everything, what can be programmed.

Writing more complex logic with the Robot scripting language ends up with two issues:

- The code gets long and difficult to maintain. There is no IDE, which would support Robot scripting language, so typos can be found only manually or by running the script. Bugs are extremely hard to detect.

- Every step defined in the Robot scripting language is also reported in the test logs. For instance, every repetition of a loop is documented with all steps performed inside the loop. This produces easily enormous amounts of irrelevant data and makes the test logs hard to read.

These issues can be easily avoided by using Python instead of the Robot scripting language. The Robot Framework requires to define the test cases in the Robot script, but all parts of one test case can be defined in Python libraries. Any test suite can use various Python libraries.

Defining a special Python library seemed advisable for testing the values of the signal TestStatus. This signal is used by the application of VCU_S to report the status of brake tests performed periodically. TestStatus is an eight bit signal with valid values from zero (default) to ten with the following definitions:

- 0: Idle, no test run
- 1: preconditions check (e.g.: no speed, no known internal errors of the system)
  displayed for 200 ms
- 2: test run
  displayed for 2 s
- 3: test finished
  displayed for 200 ms
- 4 – 10: test aborted due to different reasons
  (e.g.: 4 – train has speed, 8 – internal error)
  displayed for 5 s

Figure 19 shows the normal brake test run:

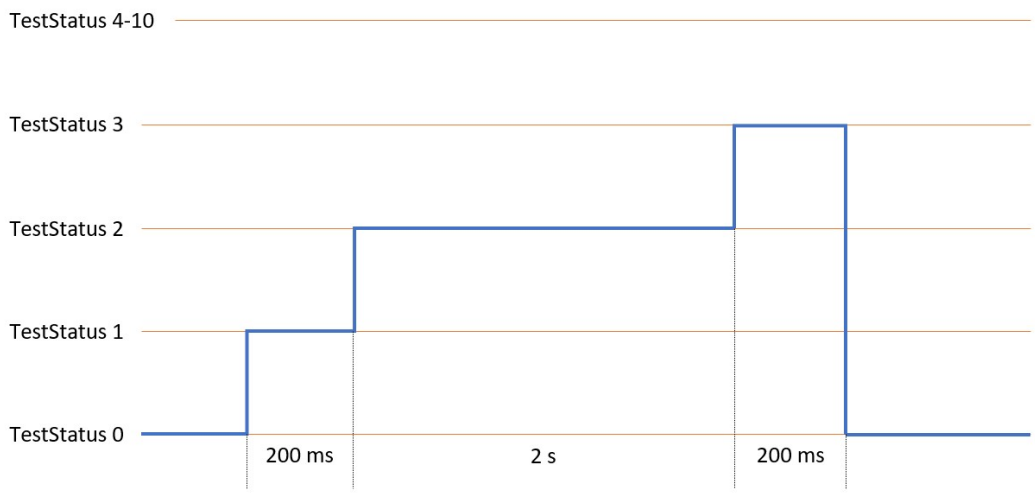


Figure 19. Normal test run.

In figure 20 the test is aborted after checking the preconditions, that is: the preconditions needed to start the brake test were not fulfilled.

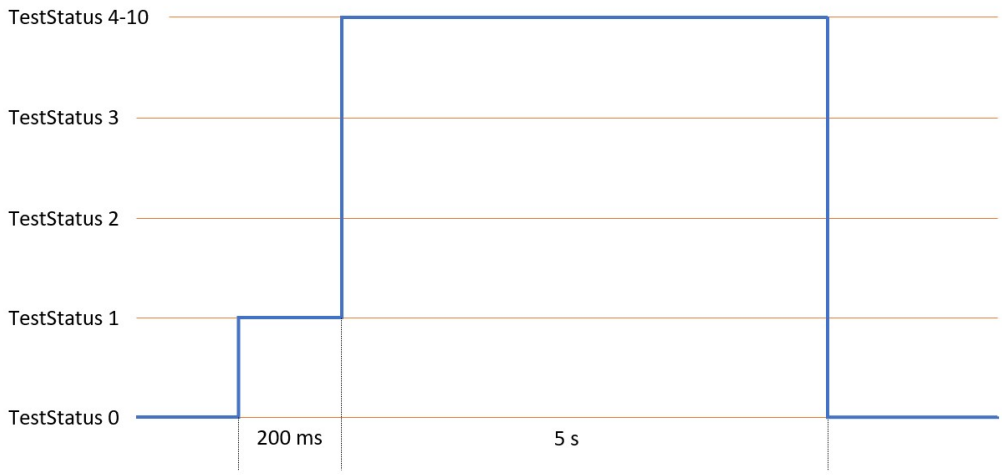


Figure 20. Test aborted at preconditions check.

In the case shown on figure 21, the preconditions were violated, after the brake test was started.

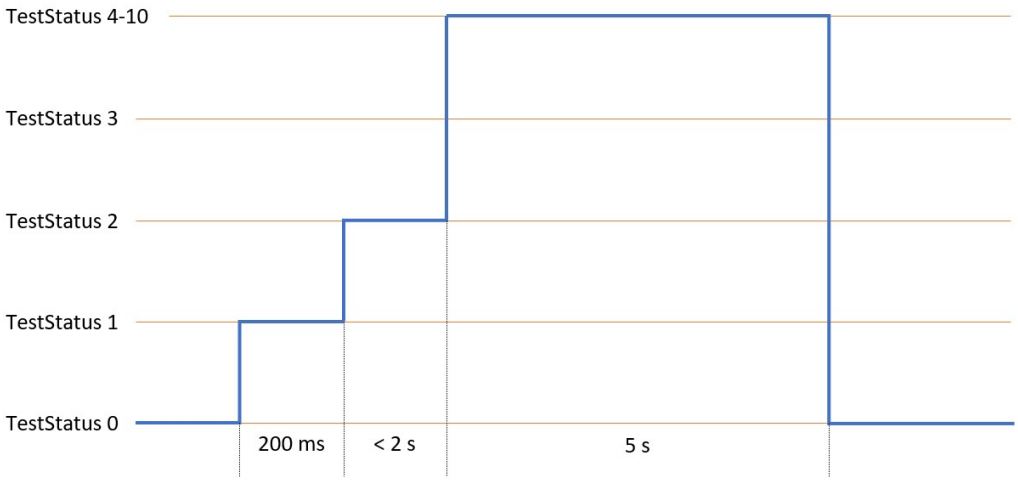


Figure 21. Test aborted during test run.

The various transitions can be tested easily, by running the same brake test over and over again, validating each time a different transition. The test cases for the values of TestStatus must raise an exception, when

- the expected value is not found at all,
- the timing of the expected value is not correct (too long, too short),
- the next value, following the expected value, is not correct.

Furthermore it should report

- the values found if the expected value was not found; to keep the list short, repeated value can be filtered out (this shows, if the test was not at all run, if it was aborted or – if it should be aborted – that it was run without being aborted),
- the time when the brake test is initiated, the time when the expected value is found (this is used to compare the brake test run with the internal logs of the VCU_S application),
- the measured duration, the expected value is displayed (to adapt the accuracy of the validation of the timing),
- the next value, following the expected value.

Waiting for a signal can be done in Python much more elegant than in the Robot scripting language (compare the for loop in the previous section 5.6.3) – see listing 11:

```
start_time = time.time()
while read_teststatus != teststatus and read_time < MAX_TIME:
       read_teststatus = _read_mvb1 (locomotive, READ_TESTSTATUS)
       read_time = time.time() - start_time
```

Listing 11. Loop waiting for a value with additional timeout.

First the start time of the loop is taken. The loop ends, when the expected value is found, or due to timeout. If the loop ends due to timeout, an exception is raised – as shown in listing 12:

```
if read_time >= MAX_TIME:
    raise TestStatusError ("TestStatus %s not found, found: %s" %
       (teststatus, _find_different (all_read)))
```

Listing 12. Raising an exception because of timeout.

Listing 13 shows, that if the expected value is found, the time is reported:

```
else:
    LogDoc ("%s  TestStatus %s found after %.3f s" %
       (time.asctime()[11:19], teststatus, read_time))
```

Listing 13. Reporting success when value is found.

Next, the duration is measured, how long the expected value is displayed. Additionally, it is checked that only the expected value and the value following the expected value are displayed – as can be seen in listing 14:

```
start_time = time.time()
while read_teststatus != next_teststatus and read_time < MAX_TIME:
    read_teststatus = _read_mvb1 (locomotive, READ_TESTSTATUS)
    if (read_teststatus != teststatus and
       read_teststatus != next_teststatus):
         error_count += 1
         error_teststatus.append (read_teststatus)

    read_time = time.time() - start_time
```

Listing 14. Loop measuring time.

If an unexpected value is found, an exception is raised. This is shown in listing 15:

```
if error_count > 0:
    raise TestStatusError ("Found unexpected Teststatus %d times; found:
       %s" % (error_count, _find_different (error_teststatus)))
```

Listing 15. Raise of an exception if incorrect value is found.

Note, that the function _find_different () filters out values, which are repeatedly found. If the test should have been aborted, but was normally run, it will return the list: [0, 1, 2, 3, 0].

An exception is raised also, when the TestStatus value remains unchanged, and the following value is never reached – see listing 16:

```
if read_time >= MAX_TIME:
    raise TestStatusError ("TestStatus %s did not change to %s" %
       (teststatus, next_teststatus))
```

Listing 16. Raise of an exception due to timeout.

When the following value is reached correctly, it is reported. Finally, the measured time is evaluated – as listing 17 shows:

```
else:
    LogDoc ("%s  TestStatus changed from %s to %s after %.3f s" %
       (time.asctime()[11:19], teststatus, next_teststatus, read_time))
    EvalTime (expected_time, read_time)
```

Listing 17. Reporting success when next value is found.

7 Conclusion

The High Speed Train project was started with manual testing. Manual test tools were created. At first the test procedures were designed to be performed manually. Only at a later phase of the project it was decided to introduce test automation. This made testing much more reliable because repeating manual tests is by nature error prone. And it made more thorough testing possible because testers were able to concentrate on improving the tests instead of performing them over and over again. But there is always room for improvement.

Aside of many details, there are two principal things, that should be improved in a new test automation project:

- **Do not automate manual testing, build test automation:** It seemed reasonable for the project not to start over again and use the existing tools. The drawback was the rather complicated access to the system under test: The Robot Framework controls a PC application; the PC application controls the MVB Tools – only the MVB Tools are directly connected to the system under test. The additional PC application is not needed and can be the reason for failures. The Robot Framework's access to the system under test should be as direct as possible. An optimized test system is shown in figure 22.

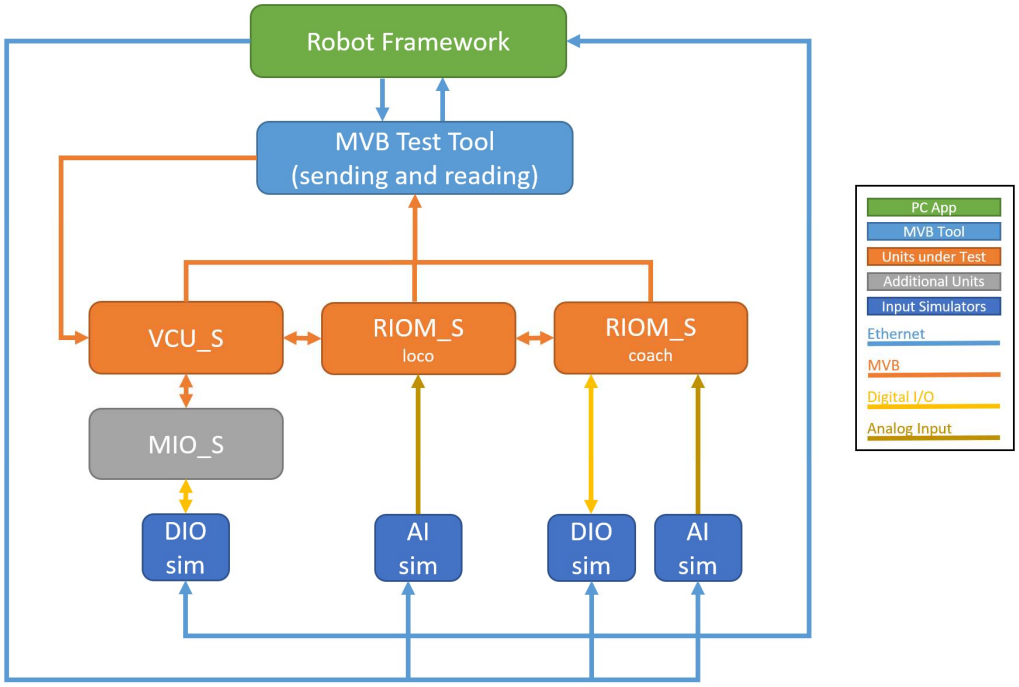


Figure 22. Test system, optimized for working with the Robot Framework (see also figure 14).

- **More Python, less Robot scripting:** Python code is a bit more difficult to write than Robot scripting language, but it is a lot easier to read, review and to maintain. All logic should be done in Python, the Robot script should be used only to define the test procedure. Defining parts of a test case in Python and giving them speaking names (e.g. OccupyCab, EnableTraction, Speed) makes the Robot script more readable. Carefully named exceptions can help to answer the most crucial question, which comes with every failed test: Is the failure on the testing side or really a failure of the SUT?

Finally, I would like to add, that working with safety systems is fulfilling work. Safety systems are in use, where human lives are at stake. Everybody working with safety systems is contributing to avoiding accidents and to make life safer.

References

1 Wikipedia.org [online]. Safety integrity level. (6. June 2021). URL: https://en.wikipedia.org/wiki/Safety_integrity_level (Accessed 20. October 2021).

2 European Committee for Electrotechnical Standardization. EN 50128:2011. Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Brussels: CENELEC; 2011.

3 European Committee for Electrotechnical Standardization. EN 61375-1:2012. Electronic railway equipment - Train communication network (TCN) - Part 1: General architecture. Brussels: CENELEC; 2012.

4 European Committee for Electrotechnical Standardization. EN 61375-3-1:2012. Electronic railway equipment - Train communication network (TCN) - Part 3-1: Multifunction Vehicle Bus (MVB). Brussels: CENELEC; 2012.

5 European Committee for Electrotechnical Standardization. EN 61375-2-1:2012. Electronic railway equipment - Train communication network (TCN) - Part 2-1: Wire Train Bus (WTB). Brussels: CENELEC; 2012.

6 European Committee for Electrotechnical Standardization. EN 61375-2-3:2015. Electronic railway equipment - Train communication network (TCN) - Part 2-3: TCN communication profile. Brussels: CENELEC; 2015.

7 EKE-Electronics.com [online]. Vehicle Control Unit (VCU). 19 February 2020. URL: https://www.eke-electronics.com/vehicle-control-unit-vcu (Accessed 20. October 2021).

8 HASLERRail.com [online]. Vehicle Control Units (VCU). NA. URL: https://www.haslerrail.com/products-solutions/obe/control-protection-and-i-o/vehicle-control-units-vcu (Accessed 20. October 2021).

9 EKE-Electronics.com [online]. Remote I/O Module (RIOM). 10. August 2020. URL: https://www.eke-electronics.com/riom-remote-i-o-module (Accessed 20. October 2021).

10 HASLERRail.com [online]. Remote I/O Units. NA. URL: https://www.haslerrail.com/products-solutions/obe/control-protection-and-i-o/remote-i-o-units (Accessed 20. October 2021).

11 EKE-Electronics.com [online]. Central Processing Unit with Serial Links (CPS, CPF). 18. May 2020. URL: https://www.eke-electronics.com/cpu-with-serial-links (Accessed 20. October 2021).

12 EKE-Electronics.com [online]. Multifunction Vehicle Bus Interface Module (MVB). 10. August 2020.
URL: https://www.eke-electronics.com/multifunction-vehicle-bus-mvb (Accessed 20 October 2021).

13 EKE-Electronics.com [online]. Wire Train Bus Interface Module (WTB, WTF). 10. August 2020.
URL: https://www.eke-electronics.com/wire-train-bus-wtb (Accessed 20. October 2021).

14 EKE-Electronics.com [online]. Digital Input/Output Module (DIO). 20. July 2020.
URL: https://www.eke-electronics.com/digital-input-ouput-module (Accessed 20. October 2021).

15 EKE-Electronics.com [online]. Pt100/Pt1000 Temperature Sensor Input Modules (PTI). 11. August 2020.
URL: https://www.eke-electronics.com/pt100-temperature-sensor-module (Accessed 20. October 2021).

16 EKE-Electronics.com [online]. High Speed Analogue Input Module (HSA). 10. August 2020.
URL: https://www.eke-electronics.com/high-speed-analogue-module (Accessed 20. October 2021).

17 Customer Specification. Architecture_TCMS_v15. [Restricted, not open to the public]. 2021.

18 Customer Specification. SW0261_SW_REQ_Export_BS012. [Restricted, not open to the public]. 2021.

19 Robotframework.org [online]. Introduction. NA.
URL: https://robotframework.org/#introduction (Accessed 20. October 2021).

20 Python.org [online]. 8. Errors and Exceptions. NA.
URL: https://docs.python.org/3/tutorial/errors.html (Accessed 20. October 2021).

21 Languev, O. Simulator of resistive temperature sensor. BE Thesis. Metropolia University of Applied Science; 2019.

22 Wikipedia.org [online]. Smoke testing (software). (16. July 2021).
URL: https://en.wikipedia.org/wiki/Smoke_testing_(software) (Accessed 7. November 2021).